数据结构模板

C 语言

云南民族大学　数学与计算机科学学院

杨德轩

前言

本模板是一本关于数据结构的参考手册。作为一本简要示例数据结构的模板，它的目的是将一些常见的数据结构进行归类、统一并呈现给读者。

编者认为用伪代码来描述算法相当于纸上谈兵，因此本模板通篇使用 C99 标准的 C 语言代码编写，并且尽我可能的加了一些注释来帮助读者理解代码，模版内的全部代码都分别在 DevC++及 Visual Studio 2017 中测试运行通过，这些代码是开源的，读者可以直接在非正式场合中使用这些代码，但如果想在以盈利的目的下使用本模板中的代码请与我联系。

源代码可以到本人的 Github 上下载，下载地址：

https://github.com/YoungTakhin/Data-Structure-with-C/archive/master.zip

本书 Github 页面：

https://github.com/YoungTakhin/Data-Structure-with-C

因编者水平有限，模板中的 bug 在所难免，望读者批评指正并帮助我完善这套数据结构模板，请与我联系。

本模板约定

标题部分

一级标题使用黑体三号字加粗；
二级标题使用黑体四号字加粗；
三级标题使用黑体小四号字加粗。

文字部分

汉语使用宋体五号字；
英文使用 Consolas 五号等宽字体。

代码部分

汉语使用宋体 11.5 号字；
英文使用 Consolas11.5 号等宽字体；
并且为了方便读者阅读，代码部分还使用了不同的颜色对代码进行了高亮处理。

联系方式

QQ:　　850189787
Github:　　https://github.com/YoungTakhin

计划

本模板后续计划补充如下数据结构:

双向链表
一些模式匹配算法

二叉树三叉链式结构
一些图的其他基本操作
图的以邻接表构造
图的关键路径算法
线索二叉树
B 树
红黑树

致谢

在本模板的编写过程中，云南民族大学数学与计算机科学学院的王康、龙品池同学帮助我进行代码的调试和测试，在此表示衷心的感谢！

杨 德 轩
2018 年 8 月

目录

1. 线性表

1.1. 顺序表

1.1.1. 定义

ADT List {

数据对象：$D = \{a_i \mid a_i \in ElemSet, i = 1, 2, \cdots, n; n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, 3, \cdots, n\}$

基本操作：

`InitList(&L)`

操作结果：构造一个空的顺序表 `L`。

`ClearList(&L)`

初始条件：顺序表 `L` 已存在；

操作结果：将 `L` 置空。

`ListEmpty(L)`

初始条件：顺序表 `L` 已存在；

操作结果：若 `L` 为空表，则返回 `TRUE`，否则返回 `FALSE`。

`ListLength(L)`

初始条件：顺序表 `L` 已存在；

操作结果：返回 `L` 中数据元素个数。

`GetElem(L, i, &e)`

初始条件：顺序表 `L` 已存在，且 `1 ≤ i ≤ ListLength(L)`；

操作结果：用 `e` 保存 `L` 中第 `i` 个数据元素的值。

`LocateElem(L, e)`

初始条件：顺序表 `L` 已存在；

操作结果：返回 `L` 中第 `1` 个值与 `e` 相同的元素的位置。若不存在这样的元素，则返回值为 `0`。

`PriorElem(L, cur_e, &pre_e)`

初始条件：顺序表 `L` 已存在；

操作结果：若 `cur_e` 是 `L` 的数据元素，且不是第一个，则用 `pre_e` 返回其前驱，函数返回 `OK`，否则函数返回 `ERROR`，`pre_e` 无定义。

`NextElem(L, cur_e, &next_e)`

初始条件：顺序表 `L` 已存在；

操作结果：若 `cur_e` 是 `L` 的数据元素，且不是第最后一个，则用 `next_e` 返回其后继，函数返回 `OK`，否则函数返回 `ERROR`，`next_e` 无定义。

`ListInsert(&L, i, e)`

初始条件：顺序表 `L` 已存在，且 `1 ≤ i ≤ ListLength(L) + 1`；

操作结果：在 `L` 中第 `i` 个位置之前插入新的数据元素 `e`，`L` 的总长度加 `1`。

`ListDelete(&L, i, &e)`

初始条件：顺序表 `L` 已存在并非空，且 `1 ≤ i ≤ ListLength(L)`；

操作结果：删除 `L` 的第 `i` 个数据元素，`L` 的总长度减 `1`，用 `e` 保存删除的数据元素。

TraverseList(L)
初始条件：顺序表 L 已存在；
操作结果：对顺序表 L 进行遍历，遍历时对 L 的每个结点访问一次并输出每个结点的数据元素。
}

1.1.2. 表示

```c
#include <stdio.h>

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Statu 是函数返回的数据类型，其值为状态代码
typedef int ElemType; // ElemType 为数据类型，类型根据实际情况而定，这里假设为 int

/* 定义顺序表 */
typedef struct {
	ElemType data[MAXSIZE]; // 存储数据元素
	int length; // 储存顺序表长度
}SqList;
```

1.1.3. 实现

```c
Status InitList(SqList *L) {
	L->length = 0;
	return OK;
}

Status ClearList(SqList *L) {
	L->length = 0;
	return OK;
}

Status ListEmpty(SqList L) {
	if(L.length == 0) {
		return TRUE;
	}
	else {
		return FALSE;
	}
}
```

```c
int ListLength(SqList L) {
	return L.length;
}

Status GetElem(SqList L, int i, ElemType *e) {
	if(L.length == 0 || i < 1 || i > L.length) {
		return ERROR;
	}
	*e = L.data[i - 1];
	return OK;
}

int LocateElem(SqList L, ElemType e) {
	int i;
	if(L.length == 0) {
		return ERROR;
	}
	for(i = 0; i < L.length; i++) {
		if(L.data[i] == e) {
			break;
		}
	}
	if(i >= L.length) {
		return ERROR;
	}
	return i + 1;
}

Status PriorElem(SqList L, ElemType e, ElemType *pre_e) {
	int i;
	i = LocateElem(L, e);
	if(i == 1 || i == 0) {
		return ERROR;
	}
	else {
		GetElem(L, i - 1, pre_e);
		return OK;
	}
}

Status NextElem(SqList L, ElemType e, ElemType *next_e) {
	int i;
	i = LocateElem(L, e);
	if(i == ListLength(L) || i == 0) {
		return ERROR;
	}
	else {
```

```
                GetElem(L, i + 1, next_e);
                return OK;
        }
}

Status ListInsert(SqList *L, int i, ElemType e) {
        int k;
        /* 顺序表已经满 */
        if(L->length == MAXSIZE) {
                return ERROR;
        }
        /* 当 i 比第一位置小或者比最后一位置后一位置还要大时 */
        if(i < 1 || i > L->length + 1) {
                return ERROR;
        }
        /* 若插入数据位置不在表尾 */
        if(i <= L->length) {
                /* 将要插入位置之后的数据元素向后移动一位 */
                for(k = L->length - 1; k >= i - 1; k--) {
                        L->data[k + 1] = L->data[k];
                }
        }
        /* 将新元素插入 */
        L->data[i - 1] = e;
        L->length++;
        return OK;
}

Status ListDelete(SqList *L, int i, ElemType *e) {
        int k;
        /* 顺序表为空 */
        if(L->length == 0) {
                return ERROR;
        }
        /* 删除位置不正确 */
        if(i < 1 || i > L->length) {
                return ERROR;
        }
        *e = L->data[i - 1];
        /* 如果删除不是最后位置 */
        if(i < L->length) {
                /* 将删除位置后继元素前移 */
                for(k = i; k < L->length; k++) {
                        L->data[k - 1] = L->data[k];
                }
        }
        L->length--;
        return OK;
```

```
}

Status TraverseList(SqList L) {
	int i;
	for(i = 0; i < L.length; i++) {
		printf("%d ", L.data[i]);
	}
	printf("\n");
	return OK;
}
```

1.1.4. 测试

```
int main(int argc, char** argv) {
	SqList L; // 声明顺序表 L
	ElemType e; // 声明数据元素 e
	Status i; // 声明状态码 i
	int j, k; // 声明临时变量

	i = InitList(&L);
	printf("初始化 L 后表长：%d\n", ListLength(L));
	for(j = 1, e = 1; j <= 5; j++, e++) {
		i = ListInsert(&L, j, e);
	}
	printf("在 L 的表头依次插入 1～5 后：");
	TraverseList(L);
	printf("L 表长：%d \n", ListLength(L));

	i = ListEmpty(L);
	printf("L 是否空：%d (1:是 0:否)\n", i);

	i = ClearList(&L);
	printf("清空 L 后：L 表长：%d\n", ListLength(L));

	i = ListEmpty(L);
	printf("L 是否为空：%d (1:是 0:否)\n", i);

	for(j = 1, e = 1; j <= 10; j++, e++) {
		ListInsert(&L, j, e);
	}
	printf("在 L 的表尾依次插入 1～10 后：");
	TraverseList(L);
	printf("L 表长：%d \n", ListLength(L));

	ListInsert(&L, 1, 0);
	printf("在 L 的表头插入 0 后：");
	TraverseList(L);
```

```
printf("L 表长: %d \n", ListLength(L));

GetElem(L, 5, &e);
printf("第 5 个元素的值为 %d\n", e);

j = 4;
k = LocateElem(L, j);
if(k) {
        printf("值为%d 的元素位置在%d\n", j, k);
}
else {
        printf("没有值为%d 的元素\n", j);
}

k = ListLength(L); // k 为表长
for(j = k + 1; j >= k; j--) {
        i = ListDelete(&L, j, &e); // 删除第 j 个位置的数据
        if(i == ERROR) {
                printf("删除第%d 个数据失败 \n", j);
        }
        else {
                printf("删除第%d 个的元素值为: %d\n", j, e);
        }
}

printf("依次输出 L 的元素:  ");
TraverseList(L);

j = 5;
ListDelete(&L, j, &e); // 删除第 5 个位置的数据
printf("删除第%d 个的元素值为: %d\n", j, e);
printf("依次输出 L 的元素:  ");
TraverseList(L);

j = 5;
i = PriorElem(L, j, &e);
if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是第一个元素，没有前驱\n", j, j);
}
else {
        printf("%d 的前驱是%d\n", j, e);
}

j = 999;
i = PriorElem(L, j, &e);
if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是第一个元素，没有前驱\n", j, j);
}
```

```
        else {
                printf("%d 的前驱是 %d\n", j, e);
        }

        j = 5;
        i = NextElem(L, j, &e);
        if(i == 0) {
                printf("没有值为%d 的元素 或 %d 是最后一个元素，没有后继\n", j, j);
        }
        else {
                printf("%d 的后继是%d\n", j, e);
        }

        j = 1000;
        i = NextElem(L, j, &e);
        if(i == 0) {
                printf("没有值为%d 的元素 或 %d 是最后一个元素，没有后继\n", j, j);
        }
        else {
                printf("%d 的后继是%d\n", j, e);
        }

        ClearList(&L);

        return 0;
}
```

1.2. 链表

1.2.1. 定义

ADT List {

数据对象：$D = \{a_i \mid a_i \in ElemSet, i = 1, 2, \cdots, n; n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, 3, \cdots, n\}$

基本操作：

InitList(&L)

操作结果：构造一个空的链表 L。

ClearList(&L)

初始条件：链表 L 已存在；

操作结果：将 L 置空。

ListEmpty(L)

初始条件：链表 L 已存在；

操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。

ListLength(L)

初始条件：链表 L 已存在；
操作结果：返回 L 中数据元素个数。
GetElem(L, i, &e)
初始条件：链表 L 已存在，且 1 ≤ i ≤ ListLength(L)；
操作结果：用 e 保存 L 中第 i 个数据元素的值。
LocateElem(L, e)
初始条件：链表 L 已存在；
操作结果：返回 L 中第 1 个值与 e 相同的元素的位置。若不存在这样的元素，则返回值为 0。
PriorElem(L, cur_e, &pre_e)
初始条件：链表 L 已存在；
操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回其前驱，函数返回 OK，否则函数返回 ERROR，pre_e 无定义。
NextElem(L, cur_e, &next_e)
初始条件：链表 L 已存在；
操作结果：若 cur_e 是 L 的数据元素，且不是第最后一个，则用 next_e 返回其后继，函数返回 OK，否则函数返回 ERROR，next_e 无定义。
ListInsert(&L, i, e)
初始条件：链表 L 已存在，且 1 ≤ i ≤ ListLength(L) + 1；
操作结果：在 L 中第 i 个位置之前插入新的数据元素 e，L 的总长度加 1。
ListDelete(&L, i, &e)
初始条件：链表 L 已存在并非空，且 1 ≤ i ≤ ListLength(L)；
操作结果：删除 L 的第 i 个数据元素，L 的总长度减 1，用 e 保存删除的数据元素。
TraverseList(L)
初始条件：链表 L 已存在；
操作结果：对链表 L 进行遍历，遍历时对 L 的每个结点访问一次并输出每个结点的数据元素。
}

1.2.2. 表示

```c
#include <stdio.h>
#include <stdlib.h> // malloc、free

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Statu 是函数返回的数据类型，其值为状态代码
typedef int ElemType; // ElemType 为数据类型，类型根据实际情况而定，这里假设为 int

/* 定义结点 */
typedef struct Node {
        ElemType data;
        struct Node *next;
```

```
}Node;

typedef struct Node *LinkList; // 定义链表
```

1.2.3. 实现

```
Status InitList(LinkList *L) {
	*L = (LinkList)malloc(sizeof(Node)); // 产生头结点,并使 L 指向此头结点
	if(!(*L)) {
		return ERROR; // 内储分配失败
	}
	(*L)->next = NULL; // 指针域为空
	return OK; // 内存分配成功
}

Status ClearList(LinkList *L) {
	LinkList p, q;
	p = (*L)->next; // p 指向第一个结点
	/* 没到表尾 */
	while(p) {
		q = p->next;
		free(p);
		p = q;
	}
	(*L)->next = NULL; // 头结点指针域为空
	return OK;
}

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

int ListLength(LinkList L) {
	int i = 0; //计数器
	LinkList p = L->next; // p 指向第一个结点
	while(p) {
		i++;
		p = p->next;
	}
	return i;
}
```

```c
Status GetElem(LinkList L, int i, ElemType *e) {
	int j; //计数器
	LinkList p;	// 声明结点 p
	p = L->next; // 让 p 指向链表 L 的第一个结点
	j = 1;
	/* p 不为空或者计数器 j 还没有等于 i 时，循环继续 */
	while(p && j < i) {
		p = p->next; // 让 p 指向下一个结点
		++j;
	}
	if(!p || j > i) {
		return ERROR; // 第 i 个元素不存在
	}
	*e = p->data; // 取第 i 个元素的数据
	return OK;
}

int LocateElem(LinkList L, ElemType e) {
	int i = 0;
	LinkList p = L->next;
	while(p) {
		i++;
		/* 找到这样的数据元素 */
		if(p->data == e) {
			return i;
		}
		p = p->next;
	}
	return 0;
}

Status PriorElem(LinkList L, ElemType e, ElemType *pre_e) {
	int i;
	i = LocateElem(L, e);
	if(i == 1 || i == 0) {
		return ERROR;
	}
	else {
		GetElem(L, i - 1, pre_e);
		return OK;
	}
}

Status NextElem(LinkList L, ElemType e, ElemType *next_e) {
	int i;
	i = LocateElem(L, e);
	if(i == ListLength(L) || i == 0) {
		return ERROR;
```

```
        }
        else {
                GetElem(L, i + 1, next_e);
                return OK;
        }
}

Status ListInsert(LinkList *L, int i, ElemType e) {
        int j;
        LinkList p, s;
        p = *L;
        j = 1;
        /* 寻找第 i 个结点 */
        while(p && j < i) {
                p = p->next;
                ++j;
        }
        if(!p || j > i) {
                return ERROR; // 第 i 个元素不存在
        }
        s = (LinkList)malloc(sizeof(Node)); // 生成新结点
        s->data = e;
        s->next = p->next; // 将 p 的后继结点赋值给 s 的后继
        p->next = s; // 将 s 赋值给 p 的后继
        return OK;
}

Status ListDelete(LinkList *L, int i, ElemType *e) {
        int j;
        LinkList p, q;
        p = *L;
        j = 1;
        /* 遍历寻找第 i 个元素 */
        while(p->next && j < i) {
                p = p->next;
                ++j;
        }
        /* 第 i 个元素不存在 */
        if(!(p->next) || j > i) {
                return ERROR;
        }
        q = p->next;
        p->next = q->next; // 将 q 的后继赋值给 p 的后继
        *e = q->data; // 将 q 结点中的数据赋值给 e
        free(q); // 让系统回收此结点，释放内存
        return OK;
}
```

```
Status TraverseList(LinkList L) {
	LinkList p = L->next;
	while(p) {
		printf("%d ", p->data);
		p = p->next;
	}
	printf("\n");
	return OK;
}
```

1.2.4. 测试

```
int main(int argc, char** argv) {
	LinkList L; //声明链表 L
	ElemType e; //声明数据元素 e
	Status i; //声明状态码
	int j, k; //声明临时变量

	i = InitList(&L);
	printf("初始化 L 后表长：%d\n", ListLength(L));
	for(j = 1, e = 1; j <= 5; j++, e++) {
		i = ListInsert(&L, j, e);
	}
	printf("在 L 的表头依次插入 1～5 后：");
	TraverseList(L);
	printf("L 表长：%d \n", ListLength(L));

	i = ListEmpty(L);
	printf("L 是否空：%d (1:是 0:否)\n", i);

	i = ClearList(&L);
	printf("清空 L 后：L 表长：%d\n", ListLength(L));

	i = ListEmpty(L);
	printf("L 是否为空：%d (1:是 0:否)\n", i);

	for(j = 1, e = 1; j <= 10; j++, e++) {
		ListInsert(&L, j, e);
	}
	printf("在 L 的表尾依次插入 1～10 后：");
	TraverseList(L);
	printf("L 表长：%d \n", ListLength(L));

	ListInsert(&L, 1, 0);
	printf("在 L 的表头插入 0 后：");
	TraverseList(L);
	printf("L 表长：%d \n", ListLength(L));
```

```
GetElem(L, 5, &e);
printf("第 5 个元素的值为 %d\n", e);

j = 4;
k = LocateElem(L, j);
if(k) {
        printf("值为%d 的元素位置在%d\n", j, k);
}
else {
        printf("没有值为%d 的元素\n", j);
}

k = ListLength(L); // k 为表长
for(j = k + 1; j >= k; j--) {
        i = ListDelete(&L, j, &e); // 删除第 j 个位置的数据
        if(i == ERROR) {
                printf("删除第%d 个数据失败 \n", j);
        }
        else {
                printf("删除第%d 个的元素值为: %d\n", j, e);
        }
}

printf("依次输出 L 的元素:  ");
TraverseList(L);

j = 5;
ListDelete(&L, j, &e); // 删除第 5 个位置的数据
printf("删除第%d 个的元素值为: %d\n", j, e);
printf("依次输出 L 的元素:  ");
TraverseList(L);

j = 5;
i = PriorElem(L, j, &e);
if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是第一个元素，没有前驱\n", j, j);
}
else {
        printf("%d 的前驱是%d\n", j, e);
}

j = 999;
i = PriorElem(L, j, &e);
if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是第一个元素，没有前驱\n", j, j);
}
else {
```

```
        printf("%d 的前驱是 %d\n", j, e);
    }

    j = 5;
    i = NextElem(L, j, &e);
    if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是最后一个元素，没有后继\n", j, j);
    }
    else {
        printf("%d 的后继是%d\n", j, e);
    }

    j = 1000;
    i = NextElem(L, j, &e);
    if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是最后一个元素，没有后继\n", j, j);
    }
    else {
        printf("%d 的后继是%d\n", j, e);
    }

    ClearList(&L);

    return 0;
}
```

1.3. 静态链表

1.3.1. 定义

ADT List {

数据对象：$D = \{a_i \mid a_i \in ElemSet,\ i = 1, 2, \cdots, n;\ n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i = 2, 3, \cdots, n\}$

基本操作：

InitList(L)

操作结果：构造一个空的静态链表 L。

ClearList(L)

初始条件：静态链表 L 已存在；

操作结果：将 L 置空。

ListEmpty(L)

初始条件：静态链表 L 已存在；

操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。

ListLength(L)

初始条件：静态链表 L 已存在；

操作结果：返回 L 中数据元素个数。

GetElem(L, i, &e)

初始条件：静态链表 L 已存在，且 1 ≤ i ≤ ListLength(L)；

操作结果：用 e 保存 L 中第 i 个数据元素的值。

LocateElem(L, e)

初始条件：静态链表 L 已存在；

操作结果：返回 L 中第 1 个值与 e 相同的元素的位置。若不存在这样的元素，则返回值为 0。

PriorElem(L, cur_e, &pre_e)

初始条件：静态链表 L 已存在；

操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回其前驱，函数返回 OK，否则函数返回 ERROR，pre_e 无定义。

NextElem(L, cur_e, &next_e)

初始条件：静态链表 L 已存在；

操作结果：若 cur_e 是 L 的数据元素，且不是第最后一个，则用 next_e 返回其后继，函数返回 OK，否则函数返回 ERROR，next_e 无定义。

ListInsert(L, i, e)

初始条件：静态链表 L 已存在，且 1 ≤ i ≤ ListLength(L) + 1；

操作结果：在 L 中第 i 个位置之前插入新的数据元素 e，L 的总长度加 1。

ListDelete(L, i)

初始条件：静态链表 L 已存在并非空，且 1 ≤ i ≤ ListLength(L)；

操作结果：删除 L 的第 i 个数据元素，L 的总长度减 1。

TraverseList(L)

初始条件：静态链表 L 已存在；

操作结果：对静态链表 L 进行遍历，遍历时对 L 的每个结点访问一次并输出每个结点的数据元素。

}

1.3.2. 表示

```c
#include <stdio.h>

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 1000 // 存储空间初始分配量

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef int ElemType; // ElemType 类型根据实际情况而定，这里假设为 int

/* 定义静态链表 */
typedef struct {
        ElemType data; // 储存数据元素
        int cur;  // 游标(Cursor)，为 0 时表示无指向
} Component, StaticLinkList[MAXSIZE];
```

```c
int ListLength(StaticLinkList L); //返回表长，供其他函数调用
Status ListDelete(StaticLinkList L, int i); //删除数据元素，供其他函数调用


 /* 若备用空间链表非空，则返回分配的结点下标，否则返回 0 */
int Malloc_SSL(StaticLinkList space) {
	/* 当前数组第一个元素的 cur 存的值就是要返回的第一个备用空闲的下标 */
	int i = space[0].cur;
	if (space[0].cur) {
		space[0].cur = space[i].cur; // 由于要拿出一个分量来使用了，所以就将其下一个分量做备用
	}
	return i;
}

/* 将下标为 k 的空闲结点回收到备用链表 */
void Free_SSL(StaticLinkList space, int k) {
	space[k].cur = space[0].cur; // 把第一个元素的 cur 值赋给要删除的分量 cur
	space[0].cur = k; // 把要删除的分量下标赋值给第一个元素的 cur
}
```

1.3.3. 实现

```c
Status InitList(StaticLinkList space) {
	int i;
	for(i = 0; i < MAXSIZE - 1; i++) {
		space[i].cur = i + 1;
	}
	space[MAXSIZE - 1].cur = 0; // 目前静态链表为空，最后一个元素的 cur 为 0
	return OK;
}

Status ClearList(StaticLinkList L) {
	int k;
	for(k = ListLength(L); k > 0; k--) {
		ListDelete(L, k);
	}
	return 1;
}

Status ListEmpty(StaticLinkList L) {
	int j = 0;
	int i = L[MAXSIZE - 1].cur;
	while(i) {
		i = L[i].cur;
		j++;
	}
```

```
        if(j == 0) {
                return TRUE;
        }
        return FALSE;
}

int ListLength(StaticLinkList L) {
        int j = 0;
        int i = L[MAXSIZE - 1].cur;
        while(i) {
                i = L[i].cur;
                j++;
        }
        return j;
}

Status GetElem(StaticLinkList L, int i, ElemType *e) {
        int j;
        int k = 1;
        for(j = L[MAXSIZE - 1].cur; j>0; k++) {
                if(k == i) {
                        *e = L[j].data;
                        return OK;
                }
                j = L[j].cur;

        }
        return ERROR;
}

int LocateElem(StaticLinkList L, ElemType e) {
        int j;
        int k = 1;
        for(j = L[MAXSIZE - 1].cur; j>0; k++) {
                if(e == L[j].data) {
                        return k;
                }
                j = L[j].cur;
        }
        return ERROR;
}

Status PriorElem(StaticLinkList L, ElemType e, ElemType *pre_e) {
        int i;
        i = LocateElem(L, e);
        if(i == 1 || i == 0) {
                return ERROR;
        }
```

```
        else {
                GetElem(L, i - 1, pre_e);
                return OK;
        }
}

Status NextElem(StaticLinkList L, ElemType e, ElemType *next_e) {
        int i;
        i = LocateElem(L, e);
        if(i == ListLength(L) || i == 0) {
                return ERROR;
        }
        else {
                GetElem(L, i+1, next_e);
                return OK;
        }
}

Status ListInsert(StaticLinkList L, int i, ElemType e) {
        int j, k, l;
        k = MAXSIZE - 1; // 注意 k 首先是最后一个元素的下标
        if(i < 1 || i > ListLength(L) + 1)
                return ERROR;
        j = Malloc_SSL(L); // 获得空闲分量的下标
        if(j) {
                L[j].data = e; // 将数据赋值给此分量的 data
                /* 找到第 i 个元素之前的位置 */
                for(l = 1; l <= i - 1; l++) {
                        k = L[k].cur;
                }
                L[j].cur = L[k].cur; // 把第 i 个元素之前的 cur 赋值给新元素的 cur/
                L[k].cur = j; // 把新元素的下标赋值给第 i 个元素之前元素的 cur
                return OK;
        }
        return ERROR;
}

Status ListDelete(StaticLinkList L, int i) {
        int j, k;
        if(i < 1 || i > ListLength(L)) {
                return ERROR;
        }
        k = MAXSIZE - 1;
        for(j = 1; j <= i - 1; j++) {
                k = L[k].cur;
        }
        j = L[k].cur;
        L[k].cur = L[j].cur;
```

```
        Free_SSL(L, j);
        return OK;
}

Status TraverseList(StaticLinkList L) {
        int i = L[MAXSIZE - 1].cur;
        while(i) {
                printf("%d ", L[i].data);
                i = L[i].cur;
        }
        printf("\n");
        return OK;
}
```

1.3.4. 测试

```
int main(int argc, char** argv) {
        StaticLinkList L; // 声明静态链表 L
        Status i; // 声明状态码 i
        ElemType e; // 声明数据元素 e
        int j, k; // 声明临时变量

        i = InitList(L);
        printf("初始化 L 后表长：%d\n", ListLength(L));
        for(j = 1, e = 1; j <= 5; j++, e++) {
                i = ListInsert(L, j, e);
        }
        printf("在 L 的表头依次插入 1～5 后：");
        TraverseList(L);
        printf("L 表长：%d \n", ListLength(L));

        i = ListEmpty(L);
        printf("L 是否空：%d (1:是 0:否)\n", i);

        i = ClearList(L);
        printf("清空 L 后：L 表长：%d\n", ListLength(L));

        i = ListEmpty(L);
        printf("L 是否为空：%d (1:是 0:否)\n", i);

        for(j = 1, e = 1; j <= 10; j++, e++) {
                ListInsert(L, j, e);
        }
        printf("在 L 的表尾依次插入 1～10 后：");
        TraverseList(L);
        printf("L 表长：%d \n", ListLength(L));
```

```
ListInsert(L, 1, 0);
printf("在 L 的表头插入 0 后：");
TraverseList(L);
printf("L 表长：%d \n", ListLength(L));

GetElem(L, 5, &e);
printf("第 5 个元素的值为 %d\n", e);

j = 4;
k = LocateElem(L, j);
if(k) {
	printf("值为%d 的元素位置在%d\n", j, k);
}
else {
	printf("没有值为%d 的元素\n", j);
}

k = ListLength(L); // k 为表长
for(j = k + 1; j >= k; j--) {
	i = ListDelete(L, j); // 删除第 j 个位置的数据
	if(i == ERROR) {
		printf("删除第%d 个数据失败\n", j);
	}
	else {
		printf("删除第%d 个的元素\n", j);
	}
}

printf("依次输出 L 的元素：  ");
TraverseList(L);

j = 5;
ListDelete(L, j); // 删除第 5 个位置的数据
printf("删除第%d 个的元素\n", j);
printf("依次输出 L 的元素：  ");
TraverseList(L);

j = 5;
i = PriorElem(L, j, &e);
if(i == 0) {
	printf("没有值为%d 的元素 或 %d 是第一个元素，没有前驱\n", j, j);
}
else {
	printf("%d 的前驱是%d\n", j, e);
}

j = 999;
i = PriorElem(L, j, &e);
```

```
    if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是第一个元素，没有前驱\n", j, j);
    }
    else {
        printf("%d 的前驱是 %d\n", j, e);
    }

    j = 5;
    i = NextElem(L, j, &e);
    if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是最后一个元素，没有后继\n", j, j);
    }
    else {
        printf("%d 的后继是%d\n", j, e);
    }

    j = 1000;
    i = NextElem(L, j, &e);
    if(i == 0) {
        printf("没有值为%d 的元素 或 %d 是最后一个元素，没有后继\n", j, j);
    }
    else {
        printf("%d 的后继是%d\n", j, e);
    }

    ClearList(L);

    return 0;
}
```

2. 栈

2.1. 顺序栈

2.1.1. 定义

ADT Stack {

数据对象：$D = \{a_i \mid a_i \in ElemSet, i = 1, 2, \cdots, n; n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, 3, \cdots, n\}$

约定 a_n 端为栈顶，a_1 端为栈底。

基本操作：

`InitStack(&S)`

操作结果：构造一个空顺序栈 S。

`ClearStack(&S)`

初始条件：顺序栈 S 已存在；

操作结果：将 S 清为空栈。

`StackEmpty(S)`

初始条件：顺序栈 S 已存在；

操作结果：若 S 为空栈，则返回 TRUE，否则返回 FALSE。

`StackLength(S)`

初始条件：顺序栈 S 已存在；

操作结果：返回 S 中数据元素个数。

`GetTop(S, &e)`

初始条件：顺序栈 S 已存在且非空；

操作结果：用 e 返回 S 的栈顶元素，不修改栈顶指针。

`Push(&S, e)`

初始条件：顺序栈 S 已存在；

操作结果：插入数据元素 e 到栈顶。

`Pop(&S, &e)`

初始条件：顺序栈 S 已存在且非空；

操作结果：删除 S 的栈顶数据元素，并用 e 返回其值。

`StackTraverse(S)`

初始条件：顺序栈 S 已存在且非空；

操作结果：从栈底到栈顶依次对 S 中每个数据元素进行访问并输出。

}

2.1.2. 表示

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

```c
/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Statu 是函数返回的数据类型，其值为状态代码
typedef int SElemType; // SElemType 为数据类型，类型根据实际情况而定，这里假设为 int

/* 定义顺序栈 */
typedef struct {
	SElemType data[MAXSIZE]; // 储存数据元素
	int top; // 用于栈顶指针
}SqStack;
```

2.1.3. 实现

```c
Status InitStack(SqStack *S) {
	S->top = -1;
	return OK;
}

Status ClearStack(SqStack *S) {
	S->top = -1;
	return OK;
}

Status StackEmpty(SqStack S) {
	if(S.top == -1) {
		return TRUE;
	}
	return FALSE;
}

int StackLength(SqStack S) {
	return S.top + 1;
}

Status GetTop(SqStack S, SElemType *e) {
	if(S.top == -1) {
		return ERROR;
	}
	*e = S.data[S.top];
	return OK;
}
```

```c
Status Push(SqStack *S, SElemType e) {
	/* 栈满 */
	if(S->top == MAXSIZE - 1) {
		return ERROR;
	}
	S->top++; // 栈顶指针增加 1
	S->data[S->top] = e; // 将新插入元素赋值给栈顶空间
	return OK;
}

Status Pop(SqStack *S, SElemType *e) {
	if(S->top == -1) {
		return ERROR;
	}
	*e = S->data[S->top]; // 将要删除的栈顶元素赋值给 e
	S->top--; // 栈顶指针减 1
	return OK;
}

Status StackTraverse(SqStack S) {
	int i = 0;
	while(i <= S.top) {
		printf("%d ", S.data[i++]);
	}
	printf("\n");
	return OK;
}
```

2.1.4. 测试

```c
int main(int argc, char** argv) {
	SqStack s; // 声明顺序栈 s
	SElemType e; // 声明数据元素 e
	int j; // 声明临时变量

	if(InitStack(&s) == OK) {
		for(j = 1; j <= 10; j++) {
			Push(&s, j);
		}
	}
	printf("栈中元素依次为:  ");
	StackTraverse(s);

	Pop(&s, &e);
	printf("弹出的栈顶元素 e=%d\n", e);
	printf("栈是否为空: %d(1:空 0:否)\n", StackEmpty(s));
```

```
    GetTop(s, &e);
    printf("栈顶元素 e=%d\n", e);
    printf("栈的长度为 %d\n", StackLength(s));

    ClearStack(&s);
    printf("清空栈后，栈是否为空：%d(1:空 0:否)\n", StackEmpty(s));

    return 0;
}
```

2.2. 两栈共享空间

2.2.1. 定义

ADT Stack {

数据对象：$D = \{a_i \mid a_i \in ElemSet, i = 1, 2, \cdots, n; n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, 3, \cdots, n\}$

约定 a_n 端为栈顶，a_1 端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空栈 S。

ClearStack(&S)

初始条件：栈 S 已存在；

操作结果：将 S 清为空栈。

StackEmpty(S)

初始条件：栈 S 已存在；

操作结果：若 S 为空栈，则返回 TRUE，否则返回 FALSE。

StackLength(S)

初始条件：栈 S 已存在；

操作结果：返回 S 中数据元素个数。

GetTop(S, &e, n)

初始条件：栈 S 已存在且非空；

操作结果：用 e 返回 n 的栈顶元素，不修改栈顶指针。

Push(&S, e, n)

初始条件：栈 S 已存在；

操作结果：插入数据元素 e 到 n 的栈顶。

Pop(&S, &e, n)

初始条件：栈 S 已存在且非空；

操作结果：删除 n 的栈顶数据元素，并用 e 返回其值。

StackTraverse(S)

初始条件：栈 S 已存在且非空；

操作结果：从栈底到栈顶依次对 S 中每个数据元素进行访问并输出。

}

2.2.2. 表示

```c
#include <stdio.h>

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Statu 是函数返回的数据类型，其值为状态代码
typedef int SElemType; // SElemType 为数据类型，类型根据实际情况而定，这里假设为 int

/* 定义两栈共享空间结构 */
typedef struct {
        SElemType data[MAXSIZE];
        int top1; // 栈 1 栈顶指针
        int top2; // 栈 2 栈顶指针
}SqDoubleStack;
```

2.2.3. 实现

```c
Status InitStack(SqDoubleStack *S) {
        S->top1 = -1;
        S->top2 = MAXSIZE;
        return OK;
}

Status ClearStack(SqDoubleStack *S) {
        S->top1 = -1;
        S->top2 = MAXSIZE;
        return OK;
}

Status StackEmpty(SqDoubleStack S) {
        if(S.top1 == -1 && S.top2 == MAXSIZE) {
                return TRUE;
        }
        return FALSE;
}

int StackLength(SqDoubleStack S) {
```

```
        return (S.top1 + 1) + (MAXSIZE - S.top2);
}

Status GetTop(SqDoubleStack S, SElemType *e, int stackNumber) {
        if(stackNumber == 1) {
                if(S.top1 == -1) {
                        return ERROR;
                }
                else {
                        *e = S.data[S.top1];
                }
                return OK;
        }
        else {
                if(S.top2 == -1) {
                        return ERROR;
                }
                else {
                        *e = S.data[S.top2];
                }
                return OK;
        }
}

Status Push(SqDoubleStack *S, SElemType e, int stackNumber) {
        /* 栈已满，不能再入栈了 */
        if(S->top1 + 1 == S->top2) {
                return ERROR;
        }
        /* 栈 1 有元素进栈 */
        if(stackNumber == 1) {
                S->data[++S->top1] = e; // 若是栈 1 则先 top1+1 后给数组元素赋值。
        }
        /* 栈 2 有元素进栈 */
        else if(stackNumber == 2) {
                S->data[--S->top2] = e; // 若是栈 2 则先 top2-1 后给数组元素赋值。
        }
        return OK;
}

Status Pop(SqDoubleStack *S, SElemType *e, int stackNumber) {
        if(stackNumber == 1) {
                /* 说明栈 1 已经是空栈，溢出 */
                if(S->top1 == -1) {
                        return ERROR;
                }
                *e = S->data[S->top1--]; // 将栈 1 的栈顶元素出栈
        }
```

```
        else if(stackNumber == 2) {
                /* 说明栈 2 已经是空栈，溢出 */
                if(S->top2 == MAXSIZE) {
                        return ERROR;
                }
                *e = S->data[S->top2++]; // 将栈 2 的栈顶元素出栈
        }
        return OK;
}

Status StackTraverse(SqDoubleStack S) {
        int i;
        i = 0;
        while(i <= S.top1) {
                printf("%d ", S.data[i++]);
        }
        i = S.top2;
        while(i < MAXSIZE) {
                printf("%d ", S.data[i++]);
        }
        printf("\n");
        return OK;
}
```

2. 2. 4. 测试

```
int main(int argc, char** argv) {
        SqDoubleStack s;
        SElemType e;
        int j;

        if(InitStack(&s) == OK) {
                for(j = 1; j <= 5; j++) {
                        Push(&s, j, 1);

                }
                for(j = MAXSIZE; j >= MAXSIZE - 2; j--) {
                        Push(&s, j, 2);
                }
        }

        printf("栈中元素依次为: ");
        StackTraverse(s);
        printf("栈长: %d\n", StackLength(s));

        Pop(&s, &e, 2);
        printf("弹出的栈顶元素 e=%d\n", e);
```

```
        printf("栈是否为：%d(1:空 0:否)\n", StackEmpty(s));

        for(j = 6; j <= MAXSIZE - 2; j++) {
                Push(&s, j, 1);
        }

        printf("栈中元素依次为：");
        StackTraverse(s);

        ClearStack(&s);
        printf("清空栈后，栈是否为空：%d(1:空 0:否)\n", StackEmpty(s));

        Push(&s, 1, 1);
        Push(&s, 3, 1);
        Push(&s, 5, 2);
        Push(&s, 7, 2);
        Pop(&s, &e, 1);
        printf("栈中元素依次为：");
        StackTraverse(s);
        printf("栈长：%d\n", StackLength(s));

        GetTop(s, &e, 1);
        printf("栈 1 的栈顶元素：%d\n", e);
        GetTop(s, &e, 2);
        printf("栈 2 的栈顶元素：%d\n", e);

        return 0;
}
```

2.3. 链栈

2.3.1. 定义

ADT Stack {

数据对象：$D = \{a_i \mid a_i \in ElemSet, i = 1, 2, \cdots, n; n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, 3, \cdots, n\}$

约定 a_n 端为栈顶，a_1 端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空链栈 S。

ClearStack(&S)

初始条件：链栈 S 已存在；

操作结果：将 S 清为空栈。

StackEmpty(S)

初始条件：链栈 S 已存在；
操作结果：若 S 为空栈，则返回 TRUE，否则返回 FALSE。
StackLength(S)
初始条件：链栈 S 已存在；
操作结果：返回 S 中数据元素个数。
GetTop(S, &e)
初始条件：链栈 S 已存在且非空；
操作结果：用 e 返回 S 的栈顶元素，不修改栈顶指针。
Push(&S, e)
初始条件：链栈 S 已存在；
操作结果：插入数据元素 e 到栈顶。
Pop(&S, e)
初始条件：链栈 S 已存在且非空；
操作结果：删除 S 的栈顶数据元素，并用 e 返回其值。
StackTraverse(S)
初始条件：链栈 S 已存在且非空；
操作结果：从栈顶到栈低依次对 S 中每个数据元素进行访问并输出。
}

2.3.2. 表示

```c
#include <stdio.h>
#include <stdlib.h> // malloc、free

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Statu 是函数返回的数据类型，其值为状态代码
typedef int SElemType; // SElemType 为数据类型，类型根据实际情况而定，这里假设为 int

/* 定义结点 */
typedef struct StackNode {
        SElemType data; // 储存数据元素
        struct StackNode *next; // 指针
}StackNode, *LinkStackPtr;

/* 定义链队列 */
typedef struct {
        LinkStackPtr top; // 栈顶
        int count; // 计数器
}LinkStack;
```

2.3.3. 实现

```
Status InitStack(LinkStack *S) {
	S->top = (LinkStackPtr)malloc(sizeof(StackNode));
	if(!S->top) {
		return ERROR;
	}
	S->top = NULL;
	S->count = 0;
	return OK;
}

Status ClearStack(LinkStack *S) {
	LinkStackPtr p, q;
	p = S->top;
	while(p) {
		q = p;
		p = p->next;
		free(q);
	}
	S->count = 0;
	return OK;
}

Status StackEmpty(LinkStack S) {
	if(S.count == 0) {
		return TRUE;
	}
	return FALSE;
}

int StackLength(LinkStack S) {
	return S.count;
}

Status GetTop(LinkStack S, SElemType *e) {
	if(S.top == NULL) {
		return ERROR;
	}
	else {
		*e = S.top->data;
	}
	return OK;
}

Status Push(LinkStack *S, SElemType e) {
	LinkStackPtr s = (LinkStackPtr)malloc(sizeof(StackNode));
	s->data = e;
```

```
        s->next = S->top; // 把当前的栈顶元素赋值给新结点的直接后继，见图中①
        S->top = s; // 将新的结点 s 赋值给栈顶指针，见图中②
        S->count++;
        return OK;
}

Status Pop(LinkStack *S, SElemType *e) {
        LinkStackPtr p;
        if(StackEmpty(*S)) {
                return ERROR;
        }
        *e = S->top->data;
        p = S->top;    // 将栈顶结点赋值给 p，见图中③
        S->top = S->top->next; // 使得栈顶指针下移一位，指向后一结点，见图中④
        free(p); // 释放结点 p
        S->count--;
        return OK;
}

Status StackTraverse(LinkStack S) {
        LinkStackPtr p;
        p = S.top;
        while(p) {
                printf("%d ", p->data);
                p = p->next;
        }
        printf("\n");
        return OK;
}
```

2.3.4. 测试

```
int main(int argc, char** argv) {
        LinkStack s; // 声明链栈 s
        SElemType e; // 声明数据元素 e
        int j; // 声明临时变量

        if(InitStack(&s) == OK) {
                for(j = 1; j <= 10; j++) {
                        Push(&s, j);
                }
        }

        printf("栈中元素依次为: ");
        StackTraverse(s);
        printf("栈长: %d\n", StackLength(s));
```

```
    Pop(&s, &e);
    printf("弹出的栈顶元素 e=%d\n", e);
    printf("栈是否为空：%d(1:空 0:否)\n", StackEmpty(s));
    GetTop(s, &e);
    printf("栈顶元素:%d\n 栈的长度为%d\n", e, StackLength(s));

    ClearStack(&s);
    printf("清空栈后，栈是否为空：%d(1:空 0:否)\n", StackEmpty(s));

    return 0;
}
```

3. 队列

3.1. 顺序循环队列

3.1.1. 定义

ADT Queue {

数据对象：$D = \{a_i \mid a_i \in ElemSet,\ i = 1, 2, \cdots, n;\ n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i = 2, 3, \cdots, n\}$

约定 a_n 端为队尾，a_1 端为队头。

基本操作：

`InitQueue(&Q)`

操作结果：构造一个空顺序循环队列 Q。

`ClearQueue(&Q)`

初始条件：顺序循环队列 Q 已存在；

操作结果：将 Q 清为空队。

`QueueEmpty(Q)`

初始条件：顺序循环队列 Q 已存在；

操作结果：若 Q 为空队，则返回 TRUE，否则返回 FALSE。

`QueueLength(Q)`

初始条件：顺序循环队列 Q 已存在；

操作结果：返回 Q 中数据元素个数。

`GetHead(Q, &e)`

初始条件：顺序循环队列 Q 已存在且非空；

操作结果：用 e 返回 Q 的队头元素，不修改队头指针。

`EnQueue (&Q, e)`

初始条件：顺序循环队列 Q 已存在；

操作结果：插入数据元素 e 到队尾。

`DeQueue (&Q, &e)`

初始条件：顺序循环队列 Q 已存在且非空；

操作结果：删除 Q 的队头数据元素，并用 e 返回其值。

`QueueTraverse(Q)`

初始条件：顺序循环队列 Q 已存在且非空；

操作结果：从队头到队尾依次对 Q 中每个数据元素进行访问并输出。

}

3.1.2. 表示

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

```c
/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef int QElemType; // QElemType 类型根据实际情况而定，这里假设为 int

/* 定义顺序循环队列 */
typedef struct {
        QElemType data[MAXSIZE]; // 储存数据元素
        int front; // 头指针
        int rear; // 尾指针，若队列不空，指向队列尾元素的下一个位置
}SqQueue;
```

3.1.3. 实现

```c
Status InitQueue(SqQueue *Q) {
        Q->front = 0;
        Q->rear = 0;
        return  OK;
}

Status ClearQueue(SqQueue *Q) {
        Q->front = 0;
        Q->rear = 0;
        return OK;
}

Status QueueEmpty(SqQueue Q) {
        /* 队列空的标志 */
        if(Q.front == Q.rear) {
                return TRUE;
        }
        return FALSE;
}

int QueueLength(SqQueue Q) {
        return (Q.rear - Q.front + MAXSIZE) % MAXSIZE;
}

Status GetHead(SqQueue Q, QElemType *e) {
        /* 队列空 */
        if(Q.front == Q.rear) {
```

```
                return ERROR;
        }
        *e = Q.data[Q.front];
        return OK;
}

Status EnQueue(SqQueue *Q, QElemType e) {
        /* 队列满标志 */
        if((Q->rear + 1) % MAXSIZE == Q->front) {
                return ERROR;
        }
        Q->data[Q->rear] = e; // 将元素 e 赋值给队尾
        Q->rear = (Q->rear + 1) % MAXSIZE; // rear 指针向后移一位置，若到最后则转到数组
头部
        return OK;
}

Status DeQueue(SqQueue *Q, QElemType *e) {
        /* 队列空标志 */
        if(Q->front == Q->rear) {
                return ERROR;
        }
        *e = Q->data[Q->front];       // 将队头元素赋值给 e
        Q->front = (Q->front + 1) % MAXSIZE; // front 指针向后移一位置，若到最后则转到
数组头部
        return OK;
}

Status QueueTraverse(SqQueue Q) {
        int i;
        i = Q.front;
        while(i != Q.rear) {
                printf("%d ", Q.data[i]);
                i = (i + 1) % MAXSIZE;
        }
        printf("\n");
        return OK;
}
```

3.1.4. 测试

```
int main(int argc, char** argv) {
        SqQueue Q; // 声明顺序循环队列 Q
        QElemType e; // 声明数据元素 e

        InitQueue(&Q);
        printf("初始化后队是否为空：%d（1:空，0:否）\n", QueueEmpty(Q));
```

```
    EnQueue(&Q, 1);
    EnQueue(&Q, 2);
    EnQueue(&Q, 3);
    printf("把 1 2 3 依次入队后数据元素: ");
    QueueTraverse(Q);
    printf("入队后队是否为空: %d（1:空，0:否）\n", QueueEmpty(Q));
    printf("入队后队长: %d\n", QueueLength(Q));

    GetHead(Q, &e);
    printf("队头数据元素: %d\n", e);

    DeQueue(&Q, &e);
    printf("出队后数据元素: ");
    QueueTraverse(Q);
    GetHead(Q, &e);
    printf("队头数据元素: %d\n", e);

    DeQueue(&Q, &e);
    printf("出队后数据元素: ");
    QueueTraverse(Q);
    printf("出队后队长: %d\n", QueueLength(Q));

    ClearQueue(&Q);
    printf("清空后队长: %d\n", QueueLength(Q));
    printf("清空后队是否为空: %d（1:空，0:否）\n", QueueEmpty(Q));

    return 0;
}
```

3.2. 链队列

3.2.1. 定义

ADT Queue {

数据对象: $D = \{a_i \mid a_i \in ElemSet,\ i = 1, 2, \cdots, n;\ n \geq 0\}$

数据关系: $R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i = 2, 3, \cdots, n\}$

约定 a_n 端为队尾，a_1 端为队头。

基本操作:

InitQueue(&Q)

操作结果：构造一个空链队列 Q。

DestroyQueue(&Q)

初始条件：链队列 Q 已存在；

操作结果：销毁队列 Q，释放空间。

ClearQueue(&Q)
初始条件：链队列 Q 已存在；
操作结果：将 Q 清为空队。
QueueEmpty(Q)
初始条件：链队列 Q 已存在；
操作结果：若 Q 为空队，则返回 TRUE，否则返回 FALSE。
QueueLength(Q)
初始条件：链队列 Q 已存在；
操作结果：返回 Q 中数据元素个数。
GetHead(Q, &e)
初始条件：链队列 Q 已存在且非空；
操作结果：用 e 返回 Q 的队头元素，不修改队头指针。
EnQueue (&Q, e)
初始条件：链队列 Q 已存在；
操作结果：插入数据元素 e 到队尾。
DeQueue (&Q, &e)
初始条件：链队列 Q 已存在且非空；
操作结果：删除 Q 的队头数据元素，并用 e 返回其值。
QueueTraverse(Q)
初始条件：链队列 Q 已存在且非空；
操作结果：从队头到队尾依次对 Q 中每个数据元素进行访问并输出。
}

3.2.2. 表示

```
#include <stdio.h>
#include <stdlib.h> // malloc、free

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef int QElemType; // QElemType 类型根据实际情况而定，这里假设为 int

/* 定义结点 */
typedef struct QNode {
        QElemType data;
        struct QNode *next;
}QNode, *QueuePtr;

/* 定义链栈 */
typedef struct {
```

```
        QueuePtr front, rear; // 队头、队尾指针
}LinkQueue;
```

3. 2. 3. 实现

```
Status InitQueue(LinkQueue *Q) {
        Q->front = (QueuePtr)malloc(sizeof(QNode));
        Q->rear = Q->front;
        if(!Q->front) {
                return ERROR;
        }
        Q->front->next = NULL;
        return OK;
}

Status DestroyQueue(LinkQueue *Q) {
        while(Q->front) {
                Q->rear = Q->front->next;
                free(Q->front);
                Q->front = Q->rear;
        }
        return OK;
}

Status ClearQueue(LinkQueue *Q) {
        QueuePtr p, q;
        Q->rear = Q->front;
        p = Q->front->next;
        Q->front->next = NULL;
        while(p) {
                q = p;
                p = p->next;
                free(q);
        }
        return OK;
}

Status QueueEmpty(LinkQueue Q) {
        if(Q.front == Q.rear) {
                return TRUE;
        }
        return FALSE;
}

int QueueLength(LinkQueue Q) {
        int i = 0;
        QueuePtr p;
```

```
        p = Q.front;
        while(Q.rear != p) {
                i++;
                p = p->next;
        }
        return i;
}

Status GetHead(LinkQueue Q, QElemType *e) {
        QueuePtr p;
        if(Q.front == Q.rear) {
                return ERROR;
        }
        p = Q.front->next;
        *e = p->data;
        return OK;
}

Status EnQueue(LinkQueue *Q, QElemType e) {
        QueuePtr s = (QueuePtr)malloc(sizeof(QNode));
        /* 存储分配失败 */
        if(!s) {
                return ERROR;
        }
        s->data = e;
        s->next = NULL;
        Q->rear->next = s; // 把拥有元素 e 的新结点 s 赋值给原队尾结点的后继
        Q->rear = s; // 把当前的 s 设置为队尾结点，rear 指向 s
        return OK;
}

Status DeQueue(LinkQueue *Q, QElemType *e) {
        QueuePtr p;
        if(Q->front == Q->rear) {
                return ERROR;
        }
        p = Q->front->next; // 将欲删除的队头结点暂存给 p
        *e = p->data; // 将欲删除的队头结点的值赋值给 e
        Q->front->next = p->next; // 将原队头结点的后继 p->next 赋值给头结点后继
        /* 若队头就是队尾，则删除后将 rear 指向头结点*/
        if(Q->rear == p) {
                Q->rear = Q->front;
        }
        free(p);
        return OK;
}

Status QueueTraverse(LinkQueue Q) {
```

```
        QueuePtr p;
        p = Q.front->next;
        while(p) {
                printf("%d ", p->data);
                p = p->next;
        }
        printf("\n");
        return OK;
}
```

3.2.4. 测试

```
int main(int argc, char** argv) {
        LinkQueue Q; // 声明链队 Q
        QElemType e; // 声明数据元素 e
        Status i; // 声明状态码 i

        InitQueue(&Q);
        printf("初始化后队是否为空：%d（1:空，0:否）\n", QueueEmpty(Q));

        EnQueue(&Q, 1);
        EnQueue(&Q, 2);
        EnQueue(&Q, 3);
        printf("把 1 2 3 依次入队后数据元素：");
        QueueTraverse(Q);
        printf("入队后队是否为空：%d（1:空，0:否）\n", QueueEmpty(Q));
        printf("入队后队长：%d\n", QueueLength(Q));

        GetHead(Q, &e);
        printf("队头数据元素：%d\n", e);

        DeQueue(&Q, &e);
        printf("出队后数据元素：");
        QueueTraverse(Q);
        GetHead(Q, &e);
        printf("队头数据元素：%d\n", e);

        DeQueue(&Q, &e);
        printf("出队后数据元素：");
        QueueTraverse(Q);
        printf("出队后队长：%d\n", QueueLength(Q));

        ClearQueue(&Q);
        printf("清空后队长：%d\n", QueueLength(Q));
        printf("清空后队是否为空：%d（1:空，0:否）\n", QueueEmpty(Q));

        i = DestroyQueue(&Q);
```

```
        if(i == OK) {
                printf("链队销毁成功\n");
        }
        else {
                printf("链队销毁失败\n");
        }

        return 0;
}
```

4. 串

4.1. 串

4.1.1. 定义

ADT String {

数据对象：$D = \{a_i \mid a_i \in CharacterSet, i = 1, 2, \cdots, n; n \geq 0\}$

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i = 2, 3, \cdots, n\}$

基本操作：

StrAssign(&T, chars)

初始条件：chars 是字符型常量；

操作结果：生成一个其值等于 chars 的串 T。

StringCopy(&T, S)

初始条件：串 S 已存在；

操作结果：由串 S 复制得串 T。

StrEmpty(S)

初始条件：串 S 已存在；

操作结果：若 S 为空串，返回 TRUE，否则返回 FALSE。

StrCompare(S, T)

初始条件：串 S 和串 T 已存在；

操作结果：若 S > T，则返回值>0；若 S = T，则返回值=0；若 S < T，则返回值<0。

StrLength(S)

初始条件：串 S 已存在；

操作结果：返回 S 串的长度。

ClearString(&S)

初始条件：串 S 已存在；

操作结果：将 S 清为空串。

Concat(&T, S1, S2)

初始条件：串 S1、S2 已存在；

操作结果：用 T 返回由 S1 和 S2 联接而成的新串。

SubString(&Sub, S, pos, len)

初始条件：串 S 已存在，$1 \leq pos \leq StrLength(S)$且$0 \leq len \leq StrLength(S) - pos + 1$；

操作结果：用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。

Index(S, T, pos)

初始条件：串 S 和串 T 已存在，T 是非空串，$1 \leq pos \leq StrLength(S)$；

操作结果：若主串 S 中存在和串 T 值相同的子串，则返回它在主串 S 中第 pos 个字符之后第一次出现的位置；否则函数值为 0。(BF 算法)

Index_KMP(S, T, pos)

初始条件：串 S 和串 T 已存在，T 是非空串，$1 \leq pos \leq StrLength(S)$；

操作结果：若主串 S 中存在和串 T 值相同的子串，则返回它在主串 S 中第 pos 个字符之后第一次出

现的位置；否则函数值为 0。（KMP 算法）

Replace(&S, T, V)

初始条件：串 S、T、V 存在，T 为非空串；

操作结果：用 V 替换主串 S 中出现的所有与 T 相等的不重叠的子串。

StrInsert(&S, pos, T)

初始条件：串 S 和串 T 已存在，T 是非空串，1 ≤ pos ≤ StrLength(S) + 1；

操作结果：在串 S 的第 pos 个字符之前插入串 T。

StrDelete(&S, pos, len)

初始条件：串 S 和串 T 已存在，T 是非空串，1 ≤ pos ≤ StrLength(S) – len + 1；

操作结果：从串 S 中删除第 pos 个字符起长度为 len 的子串。

StrPrint(S)

初始条件：串 S 已存在；

操作结果：输出串 S。

}

4.1.2. 表示

```c
#include <stdio.h>
#include <string.h> // strlen

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 存储空间初始分配量

typedef int Status;   // Status 是函数的类型,其值是函数结果状态代码，如 OK 等

typedef struct {
        char ch[MAXSIZE]; // 储存字符串
        int Length; // 串长
}String;

Status StrInsert(String *S, int pos, String T); // 字符串插入，供其他函数使用
Status StrDelete(String *S, int pos, int len); // 字符串删除，供其他函数使用

/* KMP 算法 next 函数，结果存入数组 nextval */
void get_next(String T, int *next) {
        int i, j;
        i = 1;
        j = 0;
        next[1] = 0;
        while(i < T.Length) {
                /* T[i]表示后缀的单个字符，T[j]表示前缀的单个字符 */
                if(j == 0 || T.ch[i] == T.ch[j]) {
```

```
                ++i;
                ++j;
                next[i] = j;
            }
            else {
                j = next[j]; // 若字符不相同，则 j 值迭代
            }
        }
}

/* KMP 算法 next 函数修正值,结果存入数组 nextval */
void get_nextval(String T, int *nextval) {
        int i, j;
        i = 1;
        j = 0;
        nextval[1] = 0;
        while(i < T.ch[0]) {
            /* T[i]表示后缀的单个字符，T[j]表示前缀的单个字符 */
            if(j == 0 || T.ch[i] == T.ch[j]) {
                ++i;
                ++j;
                /* 若当前字符与前缀字符不同 */
                if(T.ch[i] != T.ch[j]) {
                    nextval[i] = j; // 则当前的 j 为 nextval 在 i 位置的值
                }
                else {
                    nextval[i] = nextval[j]; // 如果与前缀字符相同，则将前缀
字符的 nextval 值赋值给 nextval 在 i 位置的值
                }
            }
            else {
                j = nextval[j]; // 若字符不相同，则 j 值迭代
            }
        }
}
```

4.1.3. 实现

```
Status StrAssign(String *T, char *chars) {
        int i;
        if(strlen(chars) > MAXSIZE) {
            return ERROR;
        }
        else {
            T->Length = strlen(chars);
            for(i = 0; i < T->Length; i++) {
                T->ch[i] = *(chars + i);
```

```
                }
                return OK;
        }
}

Status StrCopy(String *T, String S) {
        int i;
        for(i = 0; i < S.Length; i++) {
                T->ch[i] = S.ch[i];
        }
        T->Length = S.Length;
        return OK;
}

Status StrEmpty(String S) {
        if(S.Length == 0) {
                return TRUE;
        }
        return FALSE;
}

int StrCompare(String S, String T) {
        int i;
        for(i = 0; i < S.Length && i < T.Length; ++i)
                if(S.ch[i] != T.ch[i]) {
                        return S.ch[i] - T.ch[i];
                }
        return S.Length - T.Length;
}

int StrLength(String S) {
        return S.Length;
}

Status ClearString(String *S) {
        S->Length = 0; // 令串长为零
        return OK;
}

Status Concat(String *T, String S1, String S2) {
        int i;
        /* 未截断 S2 */
        if(S1.Length + S2.Length < MAXSIZE) {
                for(i = 0; i < S1.Length; i++) {
                        T->ch[i] = S1.ch[i];
                }
                for(i = 0; i < S2.Length; i++) {
                        T->ch[S1.Length + i] = S2.ch[i];
```

```
                }
                T->Length = S1.Length + S2.Length;
                return TRUE;
        }
        /* 截断 S2 */
        else {
                for(i = 0; i < S1.Length; i++) {
                        T->ch[i] = S1.ch[i];
                }
                for(i = 0; i < MAXSIZE - S1.Length; i++) {
                        T->ch[S1.Length + i] = S2.ch[i];
                }
                T->Length = MAXSIZE;
                return FALSE;
        }
}

Status SubString(String *Sub, String S, int pos, int len) {
        int i;
        if(pos <= 0 || pos > S.Length || len < 0 || len > S.Length - pos + 1) {
                return ERROR;
        }
        for(i = 0; i < len; i++) {
                Sub->ch[i] = S.ch[pos + i - 1];
        }
        Sub->Length = len;
        return OK;
}

int Index(String S, String T, int pos) {
        int i = pos; // i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹配
        int j = 1; // j 用于子串 T 中当前位置下标值
        /* 若 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续 */
        while(i <= S.Length && j <= T.Length) {
                /* 两字母相等则继续 */
                if(S.ch[i - 1] == T.ch[j - 1]) {
                        ++i;
                        ++j;
                }
                /* 指针后退重新开始匹配 */
                else {
                        i = i - j + 2; // i 退回到上次匹配首位的下一位
                        j = 1; // j 退回到子串 T 的首位
                }
        }
        if(j > T.Length) {
                return i - T.Length;
        }
```

```
        else {
                return 0;
        }
}

int Index_KMP(String S, String T, int pos) {
        int i = pos; // i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹配
        int j = 1; // j 用于子串 T 中当前位置下标值
        int next[255]; // 定义 next 数组
        //get_next(T, next);
        get_nextval(T, next); // 对串 T 作分析，得到 next 数组
        /* 若 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续 */
        while(i <= S.Length && j <= T.Length) {
                /* 两字母相等则继续，与朴素算法增加了 j=0 判断 */
                if(j == 0 || S.ch[i - 1] == T.ch[j - 1]) {
                        ++i;
                        ++j;
                }
                /* 指针后退重新开始匹配 */
                else {
                        j = next[j]; // j 退回合适的位置，i 值不变
                }
        }
        if(j > T.Length) {
                return i - T.Length;
        }
        else {
                return 0;
        }
}

Status Replace(String *S, String T, String V) {
        int i = 0; // 从串 S 的第一个字符起查找串 T
        /* T 是空串 */
        if(StrEmpty(T)) {
                return ERROR;
        }
        do {
                /* 结果 i 为从上一个 i 之后找到的子串 T 的位置 */
                i = Index(*S, T, i);
                /* 串 S 中存在串 T */
                if(i) {
                        StrDelete(S, i, StrLength(T)); // 删除该串 T
                        StrInsert(S, i, V); // 在原串 T 的位置插入串 V
                        i += StrLength(V); // 在插入的串 V 后面继续查找串 T
                }
        } while (i);
        return OK;
```

```
}

Status StrInsert(String *S, int pos, String T) {
	int i;
	if(pos < 1 || pos > S->Length + 1) {
		return ERROR;
	}
	/* 完全插入 */
	if(S->Length + T.Length <= MAXSIZE) {
		for(i = S->Length; i >= pos; i--) {
			S->ch[i + T.Length - 1] = S->ch[i - 1];
		}
		for(i = pos; i < pos + T.Length; i++) {
			S->ch[i - 1] = T.ch[i - pos];
		}
		S->Length = S->Length + T.Length;
		return TRUE;
	}
	/* 部分插入 */
	else {
		for(i = MAXSIZE; i <= pos; i--) {
			S->ch[i - 1] = S->ch[i - T.Length - 1];
		}
		for(i = pos; i < pos + T.Length; i++) {
			S->ch[i - 1] = T.ch[i - pos];
		}
		S->Length = MAXSIZE;
		return FALSE;
	}
}

Status StrDelete(String *S, int pos, int len) {
	int i;
	if(pos < 1 || pos > S->Length - len + 1 || len < 0) {
		return ERROR;
	}
	for(i = pos + len; i <= S->Length; i++) {
		S->ch[i - len - 1] = S->ch[i - 1];
	}
	S->Length -= len;
	return OK;
}

void StrPrint(String T) {
	int i;
	for(i = 0; i < T.Length; i++) {
		printf("%c", T.ch[i]);
	}
```

```
        printf("\n");
}
```

4.1.4. 测试

```
int main(int argc, char** argv) {
        String t, s1, s2; // 声明串 t, s1, s2
        String t1, t2, t3; // 声明串 t1, t2, t3
        int i; // 声明临时变量
        char s; // 声明临时变量

        StrAssign(&s1, "abcde");
        printf("串 s1: ");
        StrPrint(s1);
        printf("s1 是否为空串: %d（0:否; 1:空）\n", StrEmpty(s1));
        printf("s1 的串长: %d\n", StrLength(s1));

        StrCopy(&t, s1);
        printf("串 s1 复制到 t 后，串 t: ");
        StrPrint(t);

        StrAssign(&s2, "fghij");
        printf("串 s2: ");
        StrPrint(s2);

        i = StrCompare(s1, s2);
        if(i < 0) {
                s = '<';
        }
        else if(i == 0) {
                s = '=';
        }
        else {
                s = '>';
        }
        printf("s1 %c s2\n", s);

        ClearString(&t);
        printf("清空后 t 是否为空串: %d（0:否; 1:空）\n", StrEmpty(t));
        printf("清空后 t 的串长: %d\n", StrLength(t));

        SubString(&t, s1, 2, 3);
        printf("s1 中第 2 个位置起长度为 3 的子串 t 为: ");
        StrPrint(t);
        printf("t 是否为空串: %d（0:否; 1:空）\n", StrEmpty(t));
        printf("t 的串长: %d\n", StrLength(t));
```

```
    ClearString(&t);
    Concat(&t, s1, s2);
    printf("由 s1 和 s2 联接而成的串 t: ");
    StrPrint(t);
    printf("联接后 t 是否为空串: %d（0:否; 1:空）\n", StrEmpty(t));
    printf("联接后 t 的串长: %d\n", StrLength(t));

    printf("串 t:");
    StrPrint(t);
    printf("串 s2:");
    StrPrint(s2);
    i = Index(t, s2, 0);
    printf("s2 在 t 的第%d 个位置出现(BF 算法)\n", i);
    i = Index_KMP(t, s2, 0);
    printf("s2 在 t 的第%d 个位置出现(KMP 算法)\n", i);

    StrInsert(&s1, 2, s2);
    printf("在 s1 的第 2 个位置插入 s2 后，串 s1: ");
    StrPrint(s1);
    printf("插入后 s1 的串长: %d\n", StrLength(s1));

    StrDelete(&s1, 2, 5);
    printf("从 s1 的第 2 个位置删除长度为 5 的子串后: ");
    StrPrint(s1);
    printf("删除后 s1 的串长: %d\n", StrLength(s1));

    StrAssign(&t1, "abcdefgh");
    StrAssign(&t2, "bcd");
    StrAssign(&t3, "wxyz");

    printf("串 t1:");
    StrPrint(t1);
    printf("串 t2:");
    StrPrint(t2);
    printf("串 t3:");
    StrPrint(t3);
    Replace(&t1, t2, t3);
    printf("用 t3 替换 t1 中与 t2 匹配的子串后，串 t1:");
    StrPrint(t1);

    return 0;
}
```

5. 树

5.1. 二叉树顺序结构

5.1.1. 定义

ADT BinaryTree {

数据对象：D 是具有相同特性的数据元素的集合。

数据关系：R

若 D = ∅，则 R = ∅，称 BinaryTree 为空二叉树；

若 D ≠ ∅，则 R = {H}，H 是如下二元关系：

(1)在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

(2)若 D-{root} ≠ ∅，则存在 D-{root} = {D_1, D_r}，且 D_1 ∩ D_r = ∅；

(3)若 D_1 ≠ ∅，则 D_1 中存在唯一的元素 x_1，<root, x_1> ∈ H，且存在 D_1 上的关系 H_1 ⊂ H；

若 D_r ≠ ∅，则 D_r 中存在唯一的元素 x_r，<root, x_r> ∈ H，且存在 D_r 上的关系 H_r ⊂ H；

H = {<root, x_1>, <root, x_r>, H_1, H_r};

(4)(D_1, {H_1})是一棵符合本定义的二叉树，称为根的左子树，(D_r, {H_r})是一棵符合本定义的二叉树，称为根的右子树。

基本操作：

InitBiTree(T)

操作结果：构造空二叉树 T。

CreateBiTree(T)

初始条件：二叉树 T 已存在；

操作结果：按层序次序为二叉树 T 的结点赋值。

ClearBiTree(T)

初始条件：二叉树 T 已存在；

操作结果：将二叉树 T 清为空二叉树。

BiTreeEmpty(T)

初始条件：二叉树 T 已存在；

操作结果：若二叉树 T 为空二叉树，则返回 TRUE；否则返回 FALSE。

BiTreeDepth(T)

初始条件：二叉树 T 已存在；

操作结果：返回二叉树 T 的高度。

Root(T, &e)

初始条件：二叉树 T 已存在；；

操作结果：用 e 返回二叉树 T 的根，函数返回 OK；若二叉树 T 为空二叉树，则函数返回 ERROR。

Value(T, e)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：返回二叉树 T 中结点 e 的值。

Assgin(T, e, value)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：将二叉树 T 中的结点 e 赋值为 value。

Parent(T, e)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：若结点 e 是二叉树 T 中的非根节点，则返回它的双亲，否则返回“空”。

LeftChild(T, e)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：返回结点 e 的左孩子；若 e 无左孩子，则返回“空”。

RightChild(T, e)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：返回结点 e 的右孩子；若 e 无右孩子，则返回“空”。

LeftSibling(T, e)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：返回结点 e 的左兄弟；若 e 是左结点或 e 无左兄弟，则返回“空”。

RightSibling(T, e)

初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；

操作结果：返回结点 e 的右兄弟；若 e 是右结点或 e 无右兄弟，则返回“空”。

PreOrderTraverse(T)

初始条件：二叉树 T 已存在；

操作结果：先序遍历二叉树 T 并输出结点值。

InOrderTraverse(T)

初始条件：二叉树 T 已存在；

操作结果：中序遍历二叉树 T 并输出结点值。

PostOrderTraverse(T)

初始条件：二叉树 T 已存在；

操作结果：后序遍历二叉树 T 并输出结点值。

LevelOrderTraverse(T)

初始条件：二叉树 T 已存在；

操作结果：层序遍历二叉树 T 并输出结点值。

PrintBiTree(T)

初始条件：二叉树 T 已存在；

操作结果：逐层并按每层序号输出二叉树 T 的结点值。

}

5.1.2. 表示

```c
#include <stdio.h>
#include <math.h> // powl

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 // 存储空间初始分配量
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数
```

```c
typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef int TElemType; // QElemType 类型根据实际情况而定，这里假设为 int

typedef TElemType SqBiTree[MAX_TREE_SIZE]; // 定义二叉树，0 号单元存储根结点

/* 定义二叉树属性 */
typedef struct {
        int level, order; // 结点的层,本层序号(按满二叉树计算)
}Position;

TElemType Nil = 0; // 定义整型以 0 为空

/* 先序遍历 PreOrderTraverse()调用 */
void PreTraverse(SqBiTree T, TElemType e) {
        printf("%d ", T[e]);
        /* 左子树不空 */
        if(T[2 * e + 1] != Nil) {
                PreTraverse(T, 2 * e + 1);
        }
        /* 右子树不空 */
        if(T[2 * e + 2] != Nil) {
                PreTraverse(T, 2 * e + 2);
        }
}

/* 中序遍历 InOrderTraverse()调用 */
void InTraverse(SqBiTree T, TElemType e) {
        /* 左子树不空 */
        if(T[2 * e + 1] != Nil) {
                InTraverse(T, 2 * e + 1);
        }
        printf("%d ", T[e]);
        /* 右子树不空 */
        if(T[2 * e + 2] != Nil) {
                InTraverse(T, 2 * e + 2);
        }
}

/* 后序遍历 PostOrderTraverse()调用 */
void PostTraverse(SqBiTree T, TElemType e) {
        /* 左子树不空 */
        if(T[2 * e + 1] != Nil) {
                PostTraverse(T, 2 * e + 1);
        }
        /* 右子树不空 */
        if(T[2 * e + 2] != Nil) {
                PostTraverse(T, 2 * e + 2);
        }
```

```
        printf("%d ", T[e]);
}
```

5.1.3. 实现

```
Status InitBiTree(SqBiTree T) {
        int i;
        for(i = 0; i < MAX_TREE_SIZE; i++) {
                T[i] = Nil; // 初值为空
        }
        return OK;
}

Status CreateBiTree(SqBiTree T) {
        int i = 0;
        printf("请按层序输入结点的值(int)，0 表示空结点，输入 999 结束，结点数
≤%d: ", MAX_TREE_SIZE);
        while(1) {
                scanf("%d", &T[i]);
                if(T[i] == 999) {
                        T[i] = Nil;
                        break;
                }
                i++;
        }
        for(i = 1; i < MAX_TREE_SIZE; i++) {
                /* 此非根结点(不空)无双亲 */
                if(T[(i + 1) / 2 - 1] == Nil && T[i] != Nil) {
                        printf("出现无双亲的非根结点%d\n", T[i]);
                        return ERROR;
                }
        }
        return OK;
}

Status ClearBiTree(SqBiTree T) {
        int i;
        for(i = 0; i < MAX_TREE_SIZE; i++) {
                T[i] = Nil; // 值置空
        }
        return OK;
}

Status BiTreeEmpty(SqBiTree T) {
        /* 根结点为空,则树空 */
        if(T[0] == Nil) {
                return TRUE;
```

```
        }
        return FALSE;
}

int BiTreeDepth(SqBiTree T) {
        int i, j = -1;
        /* 找到最后一个结点 */
        for(i = MAX_TREE_SIZE - 1; i >= 0; i--) {
                if(T[i] != Nil) {
                        break;
                }
        }
        i++;
        do {
                j++;
        }
        /* 计算 2 的 j 次幂 */
        while(i >= powl(2, j));
        return j;
}

Status Root(SqBiTree T, TElemType *e) {
        /* T 空 */
        if(BiTreeEmpty(T)) {
                return ERROR;
        }
        else {
                *e = T[0];
                return OK;
        }
}

TElemType Value(SqBiTree T, Position e) {
        return T[(int)powl(2, e.level - 1) + e.order - 2];
}

Status Assign(SqBiTree T, Position e, TElemType value) {
        int i = (int)powl(2, e.level - 1) + e.order - 2; // 将层、本层序号转为矩阵的
序号
        /* 给叶子赋非空值但双亲为空 */
        if(value != Nil && T[(i + 1) / 2 - 1] == Nil) {
                return ERROR;
        }
        /* 给双亲赋空值但有叶子(不空) */
        else if(value == Nil && (T[i * 2 + 1] != Nil || T[i * 2 + 2] != Nil)) {
                return ERROR;
        }
        T[i] = value;
```

```
        return OK;
}

TElemType Parent(SqBiTree T, TElemType e) {
        int i;
        /* 空树 */
        if(T[0] == Nil) {
                return Nil;
        }
        for(i = 1; i <= MAX_TREE_SIZE - 1; i++) {
                /* 找到 e */
                if(T[i] == e) {
                        return T[(i + 1) / 2 - 1];
                }
        }
        return Nil; // 没找到 e
}

TElemType LeftChild(SqBiTree T, TElemType e) {
        int i;
        /* 空树 */
        if(T[0] == Nil) {
                return Nil;
        }
        for(i = 0; i <= MAX_TREE_SIZE - 1; i++) {
                /* 找到 e */
                if(T[i] == e) {
                        return T[i * 2 + 1];
                }
        }
        return Nil; // 没找到 e
}

TElemType RightChild(SqBiTree T, TElemType e) {
        int i;
        /* 空树 */
        if(T[0] == Nil) {
                return Nil;
        }
        for(i = 0; i <= MAX_TREE_SIZE - 1; i++) {
                /* 找到 e */
                if(T[i] == e) {
                        return T[i * 2 + 2];
                }
        }
        return Nil; // 没找到 e
}
```

```c
TElemType LeftSibling(SqBiTree T, TElemType e) {
	int i;
	/* 空树 */
	if(T[0] == Nil) {
		return Nil;
	}
	for(i = 1; i <= MAX_TREE_SIZE - 1; i++) {
		/* 找到 e 且其序号为偶数(是右孩子) */
		if(T[i] == e && i % 2 == 0) {
			return T[i - 1];
		}
	}
	return Nil; // 没找到 e
}

TElemType RightSibling(SqBiTree T, TElemType e) {
	int i;
	/* 空树 */
	if(T[0] == Nil) {
		return Nil;
	}
	for(i = 1; i <= MAX_TREE_SIZE - 1; i++) {
		/* 找到 e 且其序号为奇数(是左孩子) */
		if(T[i] == e && i % 2) {
			return T[i + 1];
		}
	}
	return Nil; // 没找到 e
}

Status PreOrderTraverse(SqBiTree T) {
	/* 树不空 */
	if(!BiTreeEmpty(T)) {
		PreTraverse(T, 0);
	}
	printf("\n");
	return OK;
}

Status InOrderTraverse(SqBiTree T) {
	/* 树不空 */
	if(!BiTreeEmpty(T)) {
		InTraverse(T, 0);
	}
	printf("\n");
	return OK;
}
```

```
Status PostOrderTraverse(SqBiTree T) {
	/* 树不空 */
	if(!BiTreeEmpty(T)) {
		PostTraverse(T, 0);
	}
	printf("\n");
	return OK;
}

Status LevelOrderTraverse(SqBiTree T)  {
	int i = MAX_TREE_SIZE - 1, j;
	while(T[i] == Nil) {
		i--; // 找到最后一个非空结点的序号
	}
	/* 从根结点起,按层序遍历二叉树 */
	for(j = 0; j <= i; j++) {
		if (T[j] != Nil){
			printf("%d ", T[j]); // 只遍历非空的结点
		}
	}
	printf("\n");
	return OK;
}

void PrintBiTree(SqBiTree T) {
	int j, k;
	Position p;
	TElemType e;
	printf("\n");
	for(j = 1; j <= BiTreeDepth(T); j++) {
		printf("第%d 层:\t", j);
		for(k = 1; k <= powl(2, j - 1); k++) {
			p.level = j;
			p.order = k;
			e = Value(T, p);
			if(e != Nil) {
				printf("%d:%d ", k, e);
			}
		}
		printf("\n");
	}
}
```

5.1.4. 测试

```
int main(int argc, char** argv) {
	SqBiTree T; // 声明二叉树 T
```

```
TElemType e; // 声明数据元素 e
Position p; // 声明二叉树属性 p

InitBiTree(T);
CreateBiTree(T);
/* 测试样例数据:
       1 2 3 4 0 5 6 7 8 0 0 9 999
       1 2 3 4 5 6 7 8 9 999
*/

printf("二叉树 T 是否是空树: %d（0:否,1:空）\n", BiTreeEmpty(T));
printf("二叉树 T 的树高: %d\n", BiTreeDepth(T));
Root(T, &e);
printf("二叉树 T 的树根: %d\n", e);

printf("层序遍历二叉树 T: ");
LevelOrderTraverse(T);
printf("先序遍历二叉树 T: ");
PreOrderTraverse(T);
printf("中序遍历二叉树 T: ");
InOrderTraverse(T);
printf("后序遍历二叉树 T: ");
PostOrderTraverse(T);
printf("逐层逐结点画出二叉树 T: ");
PrintBiTree(T);

printf("请输入结点位置(层数 第几个): ");
scanf("%d %d", &p.level, &p.order);
/* 测试样例数据:
       1 2
       4 2
*/
printf("该结点值: %d\n", Value(T, p));

/* 为了测试方便，直接引用了上边的赋值 */
Assign(T, p, 50);
printf("重新赋值后层序遍历二叉树 T: ");
LevelOrderTraverse(T);
printf("重新赋值后先序遍历二叉树 T: ");
PreOrderTraverse(T);
printf("重新赋值后中序遍历二叉树 T: ");
InOrderTraverse(T);
printf("重新赋值后后序遍历二叉树 T: ");
PostOrderTraverse(T);
printf("重新赋值后逐层逐结点画出二叉树 T: ");
PrintBiTree(T);

e = 5;
```

```
    printf("值为%d 的结点的双亲是：%d（0 表示无）\n", e, Parent(T, e));
    printf("值为%d 的结点的左孩子是：%d（0 表示无）\n", e, LeftChild(T, e));
    printf("值为%d 的结点的右孩子是：%d（0 表示无）\n", e, RightChild(T, e));
    printf("值为%d 的结点的左兄弟是：%d（0 表示无）\n", e, LeftSibling(T, e));
    printf("值为%d 的结点的右兄弟是：%d（0 表示无）\n", e, RightSibling(T, e));

    ClearBiTree(T);
    printf("清空二叉树 T 后，T 是否为空：%d（0:否，1:空）\n", BiTreeEmpty(T));

    return 0;
}
```

5.2. 二叉树二叉链式结构

5.2.1. 定义

ADT BinaryTree {

数据对象：D 是具有相同特性的数据元素的集合。

数据关系：R

若 D = ∅，则 R = ∅，称 BinaryTree 为空二叉树；

若 D ≠ ∅，则 R = {H}，H 是如下二元关系：

(1)在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

(2)若 D-{root} ≠ ∅，则存在 D-{root} = {D_1, D_r}，且 $D_1 \cap D_r = \emptyset$；

(3)若 $D_1 \neq \emptyset$，则 D_1 中存在唯一的元素 x_1，<root, x_1> ∈ H，且存在 D_1 上的关系 $H_1 \subset H$；

若 $D_r \neq \emptyset$，则 D_r 中存在唯一的元素 x_r，<root, x_r> ∈ H，且存在 D_r 上的关系 $H_r \subset H$；

H = {<root, x_1>, <root, x_r>, H_1, H_r};

(4)(D_1, {H_1})是一棵符合本定义的二叉树，称为根的左子树，(D_r, {H_r})是一棵符合本定义的二叉树，称为根的右子树。

基本操作：

InitBiTree(&T)

操作结果：构造空二叉树 T。

DestroyBiTree(&T)

初始条件：二叉树 T 已存在；

操作结果：摧毁二叉树 T。

CreateBiTree(&T)

初始条件：二叉树 T 已存在；

操作结果：按先序次序为二叉树 T 的结点赋值。

ClearBiTree(&T)

初始条件：二叉树 T 已存在；

操作结果：将二叉树 T 清为空二叉树。

BiTreeEmpty(T)

初始条件：二叉树 T 已存在；

操作结果：若二叉树 T 为空二叉树，则返回 TRUE；否则返回 FALSE。

BiTreeDepth(T)

初始条件：二叉树 T 已存在；
操作结果：返回二叉树 T 的高度。
Root(T)
初始条件：二叉树 T 已存在；；
操作结果：返回二叉树 T 的根。
Assgin(T, e, value)
初始条件：二叉树 T 已存在，e 是二叉树中某结点的值；
操作结果：将二叉树 T 中值为 e 的结点重新赋值为 value。
Parent(T, e)
初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；
操作结果：若结点 e 是二叉树 T 中的非根节点，则返回它的双亲，否则返回“空”。
LeftChild(T, e)
初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；
操作结果：返回结点 e 的左孩子；若 e 无左孩子，则返回“空”。
RightChild(T, e)
初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；
操作结果：返回结点 e 的右孩子；若 e 无右孩子，则返回“空”。
LeftSibling(T, e)
初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；
操作结果：返回结点 e 的左兄弟；若 e 是左结点或 e 无左兄弟，则返回“空”。
RightSibling(T, e)
初始条件：二叉树 T 已存在，e 是二叉树中某层某号结点；
操作结果：返回结点 e 的右兄弟；若 e 是右结点或 e 无右兄弟，则返回“空”。
PreOrderTraverse(T)
初始条件：二叉树 T 已存在；
操作结果：先序遍历二叉树 T 并输出结点值。
InOrderTraverse(T)
初始条件：二叉树 T 已存在；
操作结果：中序遍历二叉树 T 并输出结点值。
PostOrderTraverse(T)
初始条件：二叉树 T 已存在；
操作结果：后序遍历二叉树 T 并输出结点值。
LevelOrderTraverse(T)
初始条件：二叉树 T 已存在；
操作结果：层序遍历二叉树 T 并输出结点值。
}

5.2.2. 表示

```c
#include <stdio.h>
#include <stdlib.h> // malloc、free

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
```

```c
#define MAXSIZE 100 // 存储空间初始分配量

typedef int Status; // Status 是函数的类型，其值是函数结果状态代码，如 OK 等
typedef int TElemType; // TlemType 为数据类型，类型根据实际情况而定，这里假设为 int

TElemType Nil = 0; // 定义整型以 0 为空

/* 定义二叉树结点结构 */
typedef struct BiTNode {
	TElemType data; // 结点数据
	struct BiTNode *lchild, *rchild; // 左右孩子指针
}BiTNode, *BiTree;

#if 1 // 以下是引入的链队列的基本操作

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef BiTree QElemType; // QElemType 类型定义为 BiTree

/* 定义结点 */
typedef struct QNode {
	QElemType data;
	struct QNode *next;
}QNode, *QueuePtr;

/* 定义链队列 */
typedef struct {
	QueuePtr front, rear; // 队头、队尾指针
}LinkQueue;

Status InitQueue(LinkQueue *Q) {
	Q->front = (QueuePtr)malloc(sizeof(QNode));
	Q->rear = Q->front;
	if(!Q->front) {
		return ERROR;
	}
	Q->front->next = NULL;
	return OK;
}

Status QueueEmpty(LinkQueue Q) {
	if(Q.front == Q.rear) {
		return TRUE;
	}
	return FALSE;
}

Status EnQueue(LinkQueue *Q, QElemType e) {
```

```
        QueuePtr s = (QueuePtr)malloc(sizeof(QNode));
        /* 存储分配失败 */
        if(!s) {
                return ERROR;
        }
        s->data = e;
        s->next = NULL;
        Q->rear->next = s; // 把拥有元素 e 的新结点 s 赋值给原队尾结点的后继
        Q->rear = s; // 把当前的 s 设置为队尾结点，rear 指向 s
        return OK;
}

Status DeQueue(LinkQueue *Q, QElemType *e) {
        QueuePtr p;
        if (Q->front == Q->rear) {
                return ERROR;
        }
        p = Q->front->next; // 将欲删除的队头结点暂存给 p
        *e = p->data; // 将欲删除的队头结点的值赋值给 e
        Q->front->next = p->next; // 将原队头结点的后继 p->next 赋值给头结点后继
        /* 若队头就是队尾，则删除后将 rear 指向头结点*/
        if(Q->rear == p) {
                Q->rear = Q->front;
        }
        free(p);
        return OK;
}

#endif // 1

/* 返回二叉树 T 中指向数据元素值为 s 的结点指针，供其他函数调用 */
BiTree Point(BiTree T, TElemType s) {
        LinkQueue q;
        QElemType a;
        /* 非空树 */
        if(T) {
                InitQueue(&q); // 初始化队列
                EnQueue(&q, T); // 根指针入队
                /* 队不空 */
                while(!QueueEmpty(q)) {
                        DeQueue(&q, &a); // 出队，数据元素赋给 a
                        if(a->data == s) {
                                return a;
                        }
                        /* 有左孩子 */
                        if(a->lchild) {
                                EnQueue(&q, a->lchild); // 入队左孩子
                        }
```

```
                /* 有右孩子 */
                if(a->rchild) {
                        EnQueue(&q, a->rchild); // 入队右孩子
                }
        }
    }
    return NULL;
}
```

5.2.3. 实现

```
Status InitBiTree(BiTree *T) {
    *T = NULL;
    return OK;
}

void DestroyBiTree(BiTree *T) {
    if(*T) {
        /* 有左孩子 */
        if((*T)->lchild) {
            DestroyBiTree(&(*T)->lchild); // 销毁左孩子子树
        }
        /* 有右孩子 */
        if((*T)->rchild) {
            DestroyBiTree(&(*T)->rchild); // 销毁右孩子子树
        }
        free(*T); // 释放根结点
        *T = NULL; // 空指针赋 0
    }
}

Status CreateBiTree(BiTree *T) {
    TElemType i;
    scanf("%d ", &i);
    if(i == 0 || i == 999) {
        *T = NULL;
        return OK;
    }
    else {
        *T = (BiTree)malloc(sizeof(BiTNode));
        if(!*T) {
            return ERROR;
        }
        (*T)->data = i; // 生成根结点
        CreateBiTree(&(*T)->lchild); // 构造左子树
        CreateBiTree(&(*T)->rchild); // 构造右子树
    }
```

```
	return OK;
}

void ClearBiTree(BiTree *T) {
	if(*T) {
		/* 有左孩子 */
		if((*T)->lchild) {
			ClearBiTree(&(*T)->lchild); // 销毁左孩子子树
		}
		/* 有右孩子 */
		if((*T)->rchild) {
			ClearBiTree(&(*T)->rchild); // 销毁右孩子子树
		}
		free(*T); // 释放根结点
		*T = NULL; // 空指针赋 0
	}
}

Status BiTreeEmpty(BiTree T) {
	if(T) {
		return FALSE;
	}
	return TRUE;
}

int BiTreeDepth(BiTree T) {
	int i, j;
	if(!T) {
		return 0;
	}
	if(T->lchild) {
		i = BiTreeDepth(T->lchild);
	}
	else {
		i = 0;
	}
	if(T->rchild) {
		j = BiTreeDepth(T->rchild);
	}
	else {
		j = 0;
	}
	return i > j ? i + 1 : j + 1;
}

TElemType Root(BiTree T) {
	if(BiTreeEmpty(T)) {
		return Nil;
```

```
        }
        return T->data;
}

Status Assign(BiTree T, TElemType s, TElemType value) {
        Point(T, s)->data = value;
        return OK;
}

TElemType Parent(BiTree T, TElemType e) {
        LinkQueue q;
        QElemType a;
        /* 非空树 */
        if(T) {
                InitQueue(&q); // 初始化队列
                EnQueue(&q, T); // 树根指针入队
                /* 队不空 */
                while(!QueueEmpty(q)) {
                        DeQueue(&q, &a); // 出队，队列元素赋给 a
                        /* 找到 e(是其左或右孩子) */
                        if(a->lchild && a->lchild->data == e || a->rchild && \
                                a->rchild->data == e) {
                                return a->data; // 返回 e 的双亲的值
                        }
                        /* 没找到 e，则入队其左右孩子指针(如果非空) */
                        else {
                                if(a->lchild) {
                                        EnQueue(&q, a->lchild);
                                }
                                if(a->rchild) {
                                        EnQueue(&q, a->rchild);
                                }
                        }
                }
        }
        return Nil; // 树空或没找到 e
}

TElemType LeftChild(BiTree T, TElemType e) {
        BiTree a;
        /* 非空树 */
        if(T) {
                a = Point(T, e); // a 是结点 e 的指针
                /* T 中存在结点 e 且 e 存在左孩子 */
                if(a && a->lchild) {
                        return a->lchild->data; // 返回 e 的左孩子的值
                }
        }
```

```
        return Nil; // 其他情况返回空
}

TElemType RightChild(BiTree T, TElemType e) {
        BiTree a;
        /* 非空树 */
        if(T) {
                a = Point(T, e); // a 是结点 e 的指针
                /* T 中存在结点 e 且 e 存在右孩子 */
                if(a && a->rchild) {
                        return a->rchild->data; // 返回 e 的右孩子的值
                }
        }
        return Nil; // 其他情况返回空
}

TElemType LeftSibling(BiTree T, TElemType e)  {
        TElemType a;
        BiTree p;
        /* 非空树 */
        if(T) {
                a = Parent(T, e); // a 为 e 的双亲
                /* 找到 e 的双亲 */
                if(a != Nil) {
                        p = Point(T, a); // p 为指向结点 a 的指针
                        /* p 存在左右孩子且右孩子是 e */
                        if(p->lchild && p->rchild && p->rchild->data == e) {
                                return p->lchild->data; // 返回 p 的左孩子(e 的左兄弟)
                        }
                }
        }
        return Nil; /* 其他情况返回空 */
}

TElemType RightSibling(BiTree T, TElemType e) {
        TElemType a;
        BiTree p;
        /* 非空树 */
        if(T) {
                a = Parent(T, e); // a 为 e 的双亲
                /* 找到 e 的双亲 */
                if(a != Nil) {
                        p = Point(T, a); // p 为指向结点 a 的指针
                        /* p 存在左右孩子且左孩子是 e */
                        if(p->lchild && p->rchild && p->lchild->data == e) {
                                return p->rchild->data; // 返回 p 的右孩子(e 的右兄弟)
                        }
                }
```

```
        }
        return Nil; // 其他情况返回空
}

void PreOrderTraverse(BiTree T) {
        if(T == NULL) {
                return;
        }
        printf("%d ", T->data); // 显示结点数据，可以更改为其它对结点操作
        PreOrderTraverse(T->lchild); // 再先序遍历左子树
        PreOrderTraverse(T->rchild); // 最后先序遍历右子树
}

void InOrderTraverse(BiTree T) {
        if(T == NULL) {
                return;
        }
        InOrderTraverse(T->lchild); // 中序遍历左子树
        printf("%d ", T->data); // 显示结点数据，可以更改为其它对结点操作
        InOrderTraverse(T->rchild); // 最后中序遍历右子树
}

void PostOrderTraverse(BiTree T) {
        if(T == NULL) {
                return;
        }
        PostOrderTraverse(T->lchild); // 先后序遍历左子树
        PostOrderTraverse(T->rchild); // 再后序遍历右子树
        printf("%d ", T->data); // 显示结点数据，可以更改为其它对结点操作
}

void LevelOrderTraverse(BiTree T) {
        LinkQueue q;
        QElemType a;
        if(T) {
                InitQueue(&q); // 初始化队列 q
                EnQueue(&q, T); // 根指针入队
                /* 队列不空 */
                while(!QueueEmpty(q)) {
                        DeQueue(&q, &a); // 出队元素(指针)，赋值给 a
                        printf("%d ", a->data); // 访问 a 所指结点
                        /* a 有左孩子 */
                        if(a->lchild != NULL) {
                                EnQueue(&q, a->lchild); // 入队 a 的左孩子
                        }
                        /* a 有右孩子 */
                        if (a->rchild != NULL) {
                                EnQueue(&q, a->rchild); // 入队 a 的右孩子
```

```
                }
            }
        }
}
```

5.2.4. 测试

```
int main(int argc, char** argv) {
        BiTree T; // 声明二叉树 T
        TElemType e; // 声明数据元素 e

        InitBiTree(&T);
        printf("请按先序输入结点的值(int)，0 表示空结点，输入 999 结束，结点数
≤%d: ", MAXSIZE);
        CreateBiTree(&T);
        /* 测试样例数据:
        1 2 4 7 0 0 8 0 0 0 3 5 9 0 0 0 6 0 0 999
        1 2 4 8 0 0 9 0 0 5 0 0 3 6 0 0 7 0 0 999
        */

        printf("二叉树 T 是否是空树: %d（0:否,1:空）\n", BiTreeEmpty(T));
        printf("二叉树 T 的树高: %d\n", BiTreeDepth(T));
        printf("二叉树 T 的树根: %d\n", Root(T));

        printf("\n 层序遍历二叉树 T: ");
        LevelOrderTraverse(T);
        printf("\n 先序遍历二叉树 T: ");
        PreOrderTraverse(T);
        printf("\n 中序遍历二叉树 T: ");
        InOrderTraverse(T);
        printf("\n 后序遍历二叉树 T: ");
        PostOrderTraverse(T);

        printf("\n\n 给值为 2 的结点重新赋值为 50:");
        Assign(T, 2, 50);
        printf("\n 重新赋值后层序遍历二叉树 T: ");
        LevelOrderTraverse(T);
        printf("\n 重新赋值后先序遍历二叉树 T: ");
        PreOrderTraverse(T);
        printf("\n 重新赋值后中序遍历二叉树 T: ");
        InOrderTraverse(T);
        printf("\n 重新赋值后后序遍历二叉树 T: ");
        PostOrderTraverse(T);

        e = 5;
        printf("\n 值为%d 的结点的双亲是: %d（0 表示无）", e, Parent(T, e));
        printf("\n 值为%d 的结点的左孩子是: %d（0 表示无）", e, LeftChild(T, e));
```

```
    printf("\n值为%d 的结点的右孩子是：%d（0 表示无）", e, RightChild(T, e));
    printf("\n值为%d 的结点的左兄弟是：%d（0 表示无）", e, LeftSibling(T, e));
    printf("\n值为%d 的结点的右兄弟是：%d（0 表示无）", e, RightSibling(T, e));

    ClearBiTree(&T);
    printf("\n清空二叉树 T 后，T 是否为空：%d（0:否，1:空）\n", BiTreeEmpty(T));
    DestroyBiTree(&T);

    return 0;
}
```

6. 图

6.1. 图

6.1.1. 定义

ADT Graph {

数据对象：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系：VR = {<v, w> | v, w ∈ V 且 P(v, w), <v, w>表示从 v 到 w 的弧，

谓词 P(v, w)定义了弧<v, w>的意义或信息}。

基本操作：

CreateMGraph(&G)

操作结果：以邻接矩阵的方式构造图 G。

ChangeALGraph(G, &GL)

初始条件：图的邻接矩阵 G 已存在；

操作结果：由 G 生成图的邻接表 GL。

M_LocateVex(G, u)

初始条件：图的邻接矩阵 G 已存在；

操作结果：若 G 中存在顶点 u，则返回该顶点在图中位置；否则返回-1。

A_LocateVex(GL, u)

初始条件：图的邻接表 GL 已存在；

操作结果：若 GL 中存在顶点 u，则返回该顶点在图中位置；否则返回-1。

M_GetVex(G, v)

初始条件：图的邻接矩阵 G 已存在；

操作结果：返回 G 中 v 的值，若 v 不存在，则返回其他值。

A_GetVex(GL, v)

初始条件：图的邻接表 GL 已存在；

操作结果：返回 GL 中 v 的值，若 v 不存在，则返回其他值。

M_DFSTraverse(G)

初始条件：图的邻接矩阵 G 已存在；

操作结果：深度优先遍历 G。

A_DFSTraverse(GL)

初始条件：图的邻接表 GL 已存在；

操作结果：深度优先遍历 GL。

M_BFSTraverse(G)

初始条件：图的邻接矩阵 G 已存在；

操作结果：广度优先遍历 G。

A_BFSTraverse(GL)

初始条件：图的邻接表 GL 已存在；

操作结果：广度优先遍历 GL。

MiniSpanTree_Prim(G)

初始条件：图的邻接矩阵 G 已存在；
操作结果：使用 Prim 算法求 G 的最小生成树。
MiniSpanTree_Kruskal(G)
初始条件：图的邻接矩阵 G 已存在；
操作结果：使用 Kruskal 算法求 G 的最小生成树。
ShortesPath_Dijkstra(G，vstart，vend)
初始条件：图的邻接矩阵 G 已存在；
操作结果：使用 Dijkstra 算法求 G 从 vstart 顶点到 vend 顶点的最短路径。
ShortesPath_Floyd(G，vstart，vend)
初始条件：图的邻接矩阵 G 已存在；
操作结果：使用 Floyd 算法求 G 从 vstart 顶点到 vend 顶点的最短路径。
TopologicalSort(GL)
初始条件：图的邻接表 GL 已存在；
操作结果：求 GL 的拓扑排序。
}

6.1.2. 表示

```c
#include <stdio.h>
#include <stdlib.h> // malloc

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE       50 // 最大储存长度
#define MAXVEX 50 // 最大顶点数，应由用户定义
#define MAXEDGE MAXVEX * MAXVEX - 1 // 最大边数
#define INF 0x3f3f3f3f // 定义无穷大

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef char VertexType; // 顶点 VertexType 类型根据实际情况而定，这里假设为 char
typedef int EdgeType; // 边上的权值 EdgeType 类型根据实际情况而定，这里假设为 int

/* 邻接矩阵********************** */
/* 定义图的邻接矩阵结构 */
typedef struct {
        VertexType vexs[MAXVEX]; // 顶点表
        EdgeType arc[MAXVEX][MAXVEX]; // 邻接矩阵，可看作边表
        int numVertexes, numEdges; // 图中当前的顶点数和边数
        Status undirected; // 是否是无向图，值为 TRUE 是为无向图，否则为有向图
}MGraph;
/* ***************************** */

/* 邻接表************************ */
```

```c
/* 定义图的边表 */
typedef struct EdgeNode {
        int adjvex; // 邻接点域，存储该顶点对应的下标
        int weight; // 用于存储权值，对于非网图可以不需要
        struct EdgeNode *next; // 链域，指向下一个邻接点
}EdgeNode;

/* 定义图的顶点表结点 */
typedef struct VertexNode {
        int in; // 顶点入度
        VertexType data; // 顶点域，存储顶点信息
        EdgeNode *firstedge; // 边表头指针
}VertexNode, AdjList[MAXVEX];

/* 定义图的邻接表结构 */
typedef struct{
        AdjList adjList;
        int numVertexes, numEdges; // 图中当前顶点数和边数
}adjListGraph, *AdjListGraph;
/* ***************************** */

/* 对边集数组 Edge 结构的定义，最小生成树 Kruskal 算法使用 */
typedef struct {
        int begin;
        int end;
        int weight;
}Edge;

Status visited[MAXVEX]; // 访问标志的数组

typedef int PatharcDijkstra[MAXVEX]; // 用于存储 Dijkstra 算法最短路径下标的数组
typedef int ShortPathTableDijkstra[MAXVEX]; // 用于存储 Dijkstra 算法到各点最短路径的权值和

typedef int PatharcFloyd[MAXVEX][MAXVEX]; // 用于存储 Floyd 算法最短路径下标的数组
typedef int ShortPathTableFloyd[MAXVEX][MAXVEX]; // 用于存储 Floyd 算法到各点最短路径的权值和

#if 1 // 以下是引入的链队列的基本操作

/* 定义顺序循环队列 */
typedef EdgeType QElemType;
typedef struct {
        QElemType data[MAXSIZE]; // 储存数据元素
        int front; // 头指针
        int rear; // 尾指针，若队列不空，指向队列尾元素的下一个位置
}SqQueue;
```

```
Status InitQueue(SqQueue *Q) {
	Q->front = 0;
	Q->rear = 0;
	return  OK;
}

Status QueueEmpty(SqQueue Q) {
	/* 队列空的标志 */
	if(Q.front == Q.rear) {
		return TRUE;
	}
	return FALSE;
}

Status EnQueue(SqQueue *Q, QElemType e) {
	/* 队列满标志 */
	if((Q->rear + 1) % MAXSIZE == Q->front) {
		return ERROR;
	}
	Q->data[Q->rear] = e; // 将元素 e 赋值给队尾
	Q->rear = (Q->rear + 1) % MAXSIZE; // rear 指针向后移一位置，若到最后则转到数组
头部
	return OK;
}

Status DeQueue(SqQueue *Q, QElemType *e) {
	/* 队列空标志 */
	if(Q->front == Q->rear) {
		return ERROR;
	}
	*e = Q->data[Q->front];		// 将队头元素赋值给 e
	Q->front = (Q->front + 1) % MAXSIZE; // front 指针向后移一位置，若到最后则转到
数组头部
	return OK;
}

#endif // 1

/* 邻接矩阵的深度优先递归算法，内部接口，M_DFSTaverse()调用 */
void M_DFS(MGraph G, int i) {
	int j;
	visited[i] = TRUE;
	printf("%c ", G.vexs[i]); // 打印顶点，也可以其它操作
	for(j = 0; j < G.numVertexes; j++) {
		if(G.arc[i][j] == 1 && !visited[j]) {
			M_DFS(G, j); // 对为访问的邻接顶点递归调用
		}
	}
```

```
}

/* 邻接表的深度优先递归算法，A_DFSTraverse()调用 */
void A_DFS(AdjListGraph GL, int i) {
	EdgeNode *p;
	visited[i] = TRUE;
	printf("%c ", GL->adjList[i].data); // 打印顶点,也可以其它操作
	p = GL->adjList[i].firstedge;
	while(p) {
		if(!visited[p->adjvex]) {
			A_DFS(GL, p->adjvex); // 对为访问的邻接顶点递归调用
		}
		p = p->next;
	}
}

/* 交换权值以及头和尾，kruskal 算法内部接口 */
void Swapn(Edge *edges, int i, int j) {
	int temp;
	temp = edges[i].begin;
	edges[i].begin = edges[j].begin;
	edges[j].begin = temp;
	temp = edges[i].end;
	edges[i].end = edges[j].end;
	edges[j].end = temp;
	temp = edges[i].weight;
	edges[i].weight = edges[j].weight;
	edges[j].weight = temp;
}

/* 对权值进行排序，kruskal 算法内部接口 */
void sort(Edge edges[], MGraph *G) {
	int i, j;
	for(i = 0; i < G->numEdges; i++) {
		for(j = i + 1; j < G->numEdges; j++) {
			if(edges[i].weight > edges[j].weight) {
				Swapn(edges, i, j);
			}
		}
	}
	printf("\n 权排序之后的为:\n");
	for(i = 0; i < G->numEdges; i++) {

	printf("(%c, %c) %d\n", G->vexs[edges[i].begin], G->vexs[edges[i].end], edg
es[i].weight);
	}

}
```

```c
/* 查找连线顶点的尾部下标，kruskal 算法内部接口 */
int Find(int *parent, int f) {
	while(parent[f] > 0) {
		f = parent[f];
	}
	return f;
}
```

6.1.3. 实现

```c
void CreateMGraph(MGraph *G) {
	int i, j, k, w;
	printf("是否是无向图(1:是；0:否)：");
	scanf("%d", &G->undirected);
	printf("输入顶点数和边数:\n");
	scanf("%d %d", &G->numVertexes, &G->numEdges); // 输入顶点数和边数
	/* 读入顶点信息,建立顶点表 */
	for(i = 0; i < G->numVertexes; i++) {
		printf("%d 号顶点：  ", i);
		rewind(stdin); // 清空缓冲区
		scanf("%c", &G->vexs[i]);
	}
	for(i = 0; i <G->numVertexes; i++) {
		for(j = 0; j < G->numVertexes; j++) {
			if(i == j) {
				G->arc[i][j] = 0; // 自己到自己的距离是 0
			}
			else {
				G->arc[i][j] = (int)INF; // 邻接矩阵初始化
			}
		}
	}
	/* 读入 numEdges 条边，建立邻接矩阵 */
	for(k = 0; k <G->numEdges; k++) {
		printf("输入边(vi,vj)上的下标 i，下标 j 和权 w:\n");
		scanf("%d %d %d", &i, &j, &w); // 输入边(vi,vj)上的权 w
		G->arc[i][j] = w;
		/* 判断是否是无向图 */
		if(G->undirected) {
			G->arc[j][i] = G->arc[i][j]; // 因为是无向图，矩阵对称
		}
	}
}

Status ChangeALGraph(MGraph G, AdjListGraph *GL) {
	int i, j;
```

```
        EdgeNode *e;
        *GL = (AdjListGraph)malloc(sizeof(adjListGraph));
        (*GL)->numVertexes = G.numVertexes;
        (*GL)->numEdges = G.numEdges;
        /* 读入顶点信息，建立顶点表 */
        for(i = 0; i < G.numVertexes; i++) {
                (*GL)->adjList[i].in = 0;
                (*GL)->adjList[i].data = G.vexs[i];
                (*GL)->adjList[i].firstedge = NULL; // 将边表置为空表
        }
        /* 建立边表 */
        for(i = 0; i < G.numVertexes; i++) {
                for(j = 0; j < G.numVertexes; j++) {
                        if(G.arc[i][j] != 0 && G.arc[i][j] < INF) {
                                e = (EdgeNode *)malloc(sizeof(EdgeNode));
                                e->adjvex = j; // 邻接序号为 j
                                e->weight = G.arc[i][j];
                                e->next = (*GL)->adjList[i].firstedge; // 将当前顶点上
的指向的结点指针赋值给 e
                                (*GL)->adjList[i].firstedge = e; // 将当前顶点的指针指
向 e
                                (*GL)->adjList[j].in++;
                        }
                }
        }
        return TRUE;
}

int M_LocateVex(MGraph G, VertexType u) {
        int i;
        for(i = 0; i < G.numVertexes; i++) {
                if(u == G.vexs[i]) {
                        return i;
                }
        }
        return -1;
}

int A_LocateVex(AdjListGraph G, VertexType u) {
        int i;
        for(i = 0; i < G->numVertexes; i++) {
                if(u == G->adjList[i].data) {
                        return i;
                }
        }
        return -1;
}
```

```
VertexType M_GetVex(MGraph G, int v) {
	if(v < 0 || v > G.numVertexes) {
		return '?';
	}
	return G.vexs[v];
}

VertexType A_GetVex(AdjListGraph G, int v) {
	if(v < 0 || v > G->numVertexes) {
		return '?';
	}
	return G->adjList[v].data;
}

void M_DFSTraverse(MGraph G) {
	int i;
	for(i = 0; i < G.numVertexes; i++) {
		visited[i] = FALSE; // 初始所有顶点状态都是未访问过状态
	}
	for(i = 0; i < G.numVertexes; i++) {
		/* 对未访问过的顶点调用 DFS，若是连通图，只会执行一次 */
		if(!visited[i]) {
			M_DFS(G, i);
		}
	}
}

void A_DFSTraverse(AdjListGraph GL) {
	int i;
	for(i = 0; i < GL->numVertexes; i++) {
		visited[i] = FALSE; // 初始所有顶点状态都是未访问过状态
	}
	for(i = 0; i < GL->numVertexes; i++) {
		/* 对未访问过的顶点调用 DFS,若是连通图,只会执行一次 */
		if(!visited[i]) {
			A_DFS(GL, i);
		}
	}
}

void M_BFSTraverse(MGraph G) {
	int i, j;
	SqQueue Q;
	for(i = 0; i < G.numVertexes; i++) {
		visited[i] = FALSE;
	}
	InitQueue(&Q); // 初始化一辅助用的队列
	/* 对每一个顶点做循环 */
```

```
	for(i = 0; i < G.numVertexes; i++) {
		/* 若是未访问过就处理 */
		if(!visited[i]) {
			visited[i] = TRUE; // 设置当前顶点访问过
			printf("%c ", G.vexs[i]); // 打印顶点，也可以其它操作
			EnQueue(&Q, i);		// 将此顶点入队列
			/* 若当前队列不为空 */
			while(!QueueEmpty(Q)) {
				DeQueue(&Q, &i); // 将队对元素出队列，赋值给 i
				for(j = 0; j < G.numVertexes; j++) {
					/* 判断其它顶点若与当前顶点存在边且未访问过 */
					if(G.arc[i][j] == 1 && !visited[j]) {
						visited[j] = TRUE; // 将找到的此顶点标记为已访问

						printf("%c ", G.vexs[j]); // 打印顶点
						EnQueue(&Q, j); // 将找到的此顶点入队列
					}
				}
			}
		}
	}
}

void A_BFSTraverse(AdjListGraph GL) {
	int i;
	EdgeNode *p;
	SqQueue Q;
	for(i = 0; i < GL->numVertexes; i++) {
		visited[i] = FALSE;
	}
	InitQueue(&Q);
	for(i = 0; i < GL->numVertexes; i++) {
		if(!visited[i]) {
			visited[i] = TRUE;
			printf("%c ", GL->adjList[i].data);/* 打印顶点,也可以其它操作 */
			EnQueue(&Q, i);
			while(!QueueEmpty(Q)) {
				DeQueue(&Q, &i);
				p = GL->adjList[i].firstedge;		/* 找到当前顶点的边表链表头指针 */
				while(p) {
					/* 若此顶点未被访问 */
					if(!visited[p->adjvex]) {
						visited[p->adjvex] = TRUE;

	printf("%c ", GL->adjList[p->adjvex].data);
```

```
                                        EnQueue(&Q, p->adjvex);        /* 将此顶点
入队列 */
                                }
                                p = p->next; // 指针指向下一个邻接点
                        }
                }
        }
}

void MiniSpanTree_Prim(MGraph G) {
        int min, i, j, k;
        int adjvex[MAXVEX]; // 保存相关顶点下标
        int lowcost[MAXVEX]; // 保存相关顶点间边的权值
        lowcost[0] = 0; // 初始化第一个权值为 0，即 v0 加入生成树，lowcost 的值为 0，在这
里就是此下标的顶点已经加入生成树
        adjvex[0] = 0; // 初始化第一个顶点下标为 0
        /* 循环除下标为 0 外的全部顶点 */
        for(i = 1; i < G.numVertexes; i++) {
                lowcost[i] = G.arc[0][i]; // 将 v0 顶点与之有边的权值存入数组
                adjvex[i] = 0; // 初始化都为 v0 的下标
        }
        printf("\n");
        for(i = 1; i < G.numVertexes; i++) {
                min = INF; // 初始化最小权值为∞
                j = 1; k = 0;
                /* 循环全部顶点 */
                while(j < G.numVertexes) {
                        /* 如果权值不为 0 且权值小于 min */
                        if(lowcost[j] != 0 && lowcost[j] < min) {
                                min = lowcost[j]; // 则让当前权值成为最小值
                                k = j; // 将当前最小值的下标存入 k
                        }
                        j++;
                }
                printf("边:(%c, %c)\t 权值: %d\n", G.vexs[adjvex[k]], G.vexs[k]\
                        , G.arc[adjvex[k]][k]); // 打印当前顶点边中权值最小的边
                lowcost[k] = 0; // 将当前顶点的权值设置为 0,表示此顶点已经完成任务
                /* 循环所有顶点 */
                for(j = 1; j < G.numVertexes; j++) {
                        /* 如果下标为 k 顶点各边权值小于此前这些顶点未被加入生成树权值 */
                        if(lowcost[j] != 0 && G.arc[k][j] < lowcost[j]) {
                                lowcost[j] = G.arc[k][j]; // 将较小的权值存入 lowcost 相
应位置
                                adjvex[j] = k; // 将下标为 k 的顶点存入 adjvex
                        }
                }
        }
```

```
}

void MiniSpanTree_Kruskal(MGraph G) {
	int i, j, n, m;
	int k = 0;
	int parent[MAXVEX]; // 定义一数组用来判断边与边是否形成环路
	Edge edges[MAXEDGE]; // 定义边集数组,edge 的结构为 begin,end,weight,均为整型
	/* 构建边集数组 */
	for(i = 0; i < G.numVertexes - 1; i++) {
		for(j = i + 1; j < G.numVertexes; j++) {
			if(G.arc[i][j] < INF) {
				edges[k].begin = i;
				edges[k].end = j;
				edges[k].weight = G.arc[i][j];
				k++;
			}
		}
	}
	sort(edges, &G); // 排序边集数组
	for(i = 0; i < G.numVertexes; i++) {
		parent[i] = 0; // 初始化数组值为 0
	}
	printf("打印最小生成树: \n");
	/* 循环每一条边 */
	for(i = 0; i < G.numEdges; i++) {
		n = Find(parent, edges[i].begin);
		m = Find(parent, edges[i].end);
		/* 假如 n 与 m 不等，说明此边没有与现有的生成树形成环路 */
		if(n != m) {
			parent[n] = m; // 将此边的结尾顶点放入下标为起点的 parent 中，表
示此顶点已经在生成树集合中
			printf("边:(%c, %c)\t 权值: %d\n", G.vexs[edges[i].begin]\
				, G.vexs[edges[i].end], edges[i].weight);
		}
	}
}

Status ShortestPath_Dijkstra(MGraph G, int vstart, int vend) {
	if(vend < 0 || vstart < 0 || vstart > G.numVertexes || vend > G.numVertexes
) {
		printf("输入有误\n");
		return FALSE;
	}
	PatharcDijkstra P; // 存储最短路径下标的数组
	ShortPathTableDijkstra D; // 某点到其余各点的最短路径
	int v, w, k, min;
	int final[MAXVEX]; // final[w]=1 表示求得顶点 vstart 至 vw 的最短路径
	int i, j; // 临时变量
```

```
    /* 若起点位置比终点位置大，则调换 */
    if(vstart > vend) {
        int tmp = vstart;
        vstart = vend;
        vend = tmp;
    }
    /* 初始化数据 */
    for(v = 0; v < G.numVertexes; v++) {
        final[v] = 0; // 全部顶点初始化为未知最短路径状态
        D[v] = G.arc[vstart][v]; // 将与 vstart 点有连线的顶点加上权值
        P[v] = 0; // 初始化路径数组 P 为 0
    }
    D[vstart] = 0; // vstart 至 vstart 路径为 0
    final[vstart] = 1; // vstart 至 vstart 不需要求路径
    /* 开始主循环，每次求得 vstart 到某个 v 顶点的最短路径 */
    for(v = 0; v < G.numVertexes; v++) {
        min = INF; // 当前所知离 v0 顶点的最近距离
        /* 寻找离 vstart 最近的顶点 */
        for(w = 0; w < G.numVertexes; w++) {
            if(!final[w] && D[w] < min) {
                k = w;
                min = D[w]; // w 顶点离 vstart 顶点更近
            }
        }
        final[k] = 1; // 将目前找到的最近的顶点置为 1
        /* 修正当前最短路径及距离 */
        for(w = 0; w < G.numVertexes; w++) {
            /* 若经过 v 顶点的路径比现在这条路径的长度短，则说明找到了更短的路径，修改 D[w]和 P[w] */
            if(!final[w] && (min + G.arc[k][w] < D[w])) {
                D[w] = min + G.arc[k][w]; // 修改当前路径长度
                P[w] = k;
            }
        }
    }

    /* 以下为输出 */
    printf("最短路径倒序如下:\n");
    for(i = 0; i <= vend; ++i) {
        printf("%c - %c : ", G.vexs[vstart], G.vexs[i]);
        j = i;
        while(P[j] != 0) {
            printf("%c ", G.vexs[P[j]]);
            j = P[j];
        }
        printf("\n");
    }
    printf("顶点 %c 到各顶点的最短路径长度为:\n", G.vexs[vstart]);
```

```
        for(i = 0; i < G.numVertexes; ++i) {
                if(D[i] == INF) {
                        printf("%c --> %c :∞\n", G.vexs[vstart], G.vexs[i]);
                }
                else {
                        printf("%c --> %c : %d \n", G.vexs[vstart], G.vexs[i], D[i]);
                }
        }
        printf("%c 到 %c 的最短路径长度: %d\n"\
                , G.vexs[vstart], G.vexs[vend], D[vend]);

        return TRUE;
}

Status ShortestPath_Floyd(MGraph G, int vstart, int vend) {
        if(vend < 0 || vstart < 0 || vstart > G.numVertexes || vend > G.numVertexes
) {
                printf("输入有误\n");
                return FALSE;
        }
        PatharcFloyd P;
        ShortPathTableFloyd D;
        int v, w; // v 为各起点，w 为各终点
        int k; // k 为临时变量
        /* 初始化 D 与 P */
        for(v = 0; v < G.numVertexes; ++v) {
                for(w = 0; w < G.numVertexes; ++w) {
                        D[v][w] = G.arc[v][w]; // D[v][w]值即为对应点间的权值
                        P[v][w] = w; // 初始化 P
                }
        }
        for(k = 0; k < G.numVertexes; ++k) {
                for(v = 0; v < G.numVertexes; ++v) {
                        for(w = 0; w < G.numVertexes; ++w) {
                                /* 如果经过下标为 k 顶点路径比原两点间路径更短 */
                                if(D[v][w] > D[v][k] + D[k][w]) {
                                        D[v][w] = D[v][k] + D[k][w]; // 将当前两点间权
值设为更小的一个

                                        P[v][w] = P[v][k]; // 路径设置为经过下标为 k 的顶
点
                                }
                        }
                }
        }

        /* 以下为输出 */
        printf("起点到终点最短下一个路径 P(纵坐标：起点；横坐标：终点)：\n");
        printf("\t");
```

```
	for(w = 0; w < G.numVertexes; ++w) {
		printf("%c:\t", G.vexs[w]);
	}
	printf("\n\n");
	for(v = 0; v < G.numVertexes; ++v) {
		printf("%c:\t", G.vexs[v]);
		for(w = 0; w < G.numVertexes; ++w) {
			printf("%c\t", G.vexs[P[v][w]]);
		}
		printf("\n");
	}
	printf("各顶点间最短路径如下:\n");
	for(v = 0; v < G.numVertexes; ++v) {
		for(w = v + 1; w < G.numVertexes; w++) {
			if(D[v][w] == INF) {
				printf("%c - %c 路径长度: ∞ ", G.vexs[v], G.vexs[w]);
			}
			else {
				printf("%c - %c 路径长
度: %d ", G.vexs[v], G.vexs[w], D[v][w]);
			}
			k = P[v][w];                          /* 获得第一个路径顶点下
标 */
			printf("\t路径: %c", G.vexs[v]);      /* 打印源点 */
			/* 如果路径顶点下标不是终点 */
			while(k != w) {
				printf(" -> %c", G.vexs[k]); /* 打印路径顶点 */
				k = P[k][w];                      /* 获得下一个路径顶点下
标 */
			}
			printf(" -> %c\n", G.vexs[w]);       /* 打印终点 */
		}
		printf("\n");
	}
	printf("起点到终点最短路径长度 D(纵坐标：起点；横坐标：终点)：\n");
	printf("\t");
	for(w = 0; w < G.numVertexes; ++w) {
		printf("%c:\t", G.vexs[w]);
	}
	printf("\n\n");
	for(v = 0; v < G.numVertexes; ++v) {
		printf("%c:\t", G.vexs[v]);
		for(w = 0; w < G.numVertexes; ++w) {
			if(D[v][w] == INF) {
				printf("∞\t");
			}
			else{
				printf("%d\t", D[v][w]);
```

```
                }
            }
            printf("\n");
        }
        printf("%c 到 %c 的最短路径长度: %d\n"\
            , G.vexs[vstart], G.vexs[vend], D[vstart][vend]);
        return TRUE;
}

Status TopologicalSort(AdjListGraph GL) {
        EdgeNode *e;
        int i, k;
        int m = 0;
        EdgeType gettop;
        EdgeType top = 0;  /* 用于栈指针下标  */
        int count = 0;/* 用于统计输出顶点的个数  */
        EdgeType *stack;      /* 建栈将入度为 0 的顶点入栈  */
        stack = (EdgeType *)malloc(GL->numVertexes * sizeof(EdgeType));
        for(i = 0; i < GL->numVertexes; i++) {
            /* 将入度为 0 的顶点入栈 */
            if(0 == GL->adjList[i].in) {
                stack[++top] = i;
            }
        }
        while(top != 0) {
            gettop = stack[top--];
            printf("%c -> ", GL->adjList[gettop].data);
            count++; // 输出 i 号顶点，并计数
            for(e = GL->adjList[gettop].firstedge; e; e = e->next) {
                k = e->adjvex;
                /* 将 i 号顶点的邻接点的入度减 1，如果减 1 后为 0，则入栈 */
                if(!(--GL->adjList[k].in)) {
                    stack[++top] = k;
                }
            }
        }
        printf("\n");
        if(count < GL->numVertexes) {
            return ERROR;
        }
        else {
            return OK;
        }
}
```

6.1.4. 测试

```
int main(int argc, char** argv) {
        MGraph G; // 声明图邻接矩阵表示 G
        AdjListGraph GL; // 声明图邻接表表示 GL
        VertexType ch; // 声明顶点元素 ch
        int i; // 声明临时变量
        int start, end; // 声明临时变量

        CreateMGraph(&G);
        ChangeALGraph(G, &GL);

        printf("请输入要查找的顶点：");
        rewind(stdin); // 刷新缓冲区
        scanf("%c", &ch);
        printf("邻接矩阵查找 %c 的位置是：%d\n", ch, M_LocateVex(G, ch));
        printf("邻接表查找 %c 的位置是：%d\n", ch, A_LocateVex(GL, ch));

        printf("请输入要查找的顶点的位置：");
        rewind(stdin); // 刷新缓冲区
        scanf("%d", &i);
        printf("邻接矩阵 %d 的位置是顶点：%c\n", i, M_GetVex(G, i));
        printf("邻接表 %d 的位置是顶点：%c\n", i, A_GetVex(GL, i));

        printf("邻接矩阵深度优先遍历：");
        M_DFSTraverse(G);
        printf("\n");

        printf("邻接矩阵广度优先遍历：");
        M_BFSTraverse(G);
        printf("\n");

        printf("邻接表深度优先遍历：");
        A_DFSTraverse(GL);
        printf("\n");

        printf("邻接表广度优先遍历：");
        A_BFSTraverse(GL);
        printf("\n");

        printf("最小生成树 Prim 算法：");
        MiniSpanTree_Prim(G);
        printf("\n");

        printf("最小生成树 Kruskal 算法：");
        MiniSpanTree_Kruskal(G);
        printf("\n");
```

```
    printf("请输入待求路径的起点号和终点号：");
    rewind(stdin); // 刷新缓冲区
    scanf("%d %d", &start, &end);

    printf("最短路径 Dijkstra 算法：\n\n");
    ShortestPath_Dijkstra(G, start, end);
    printf("\n");

    printf("最短路径 Floyd 算法：\n\n");
    ShortestPath_Floyd(G, start, end);
    printf("\n");

    printf("拓扑排序：\n");
    TopologicalSort(GL);
    printf("\n");

    return 0;
}
```

7. 查找

7.1. 线性表静态查找

7.1.1. 表示

```c
#include <stdio.h>
#include <stdlib.h> // malloc、srand、rand
#include <string.h> // memcpy
#include <time.h> // time

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 50 // 存储空间初始分配量
#define INTERVAL 2 // 随机数波动区间
#define LIMIT MAXSIZE * INTERVAL - 1 // 随机数范围

typedef int Status; // Status 是函数的类型，其值是函数结果状态代码，如 OK 等
typedef int KeyType; // 关键字的数据类型，类型根据实际情况而定，这里假设为 int

/* 斐波那契函数，将斐波那契数列赋值到 a 数组，供 Interpolation_Search()函数使用 */
Status Fibonacci(int *a) {
	int i;
	a[0] = 0;
	a[1] = 1;
	for(i = 2; i < MAXSIZE; ++i) {
		a[i] = a[i - 1] + a[i - 2];
	}
	return OK;
}

/* 生成顺序随机数组，测试用 */
int CreatSeqArray(int *a) {
	/*随机产生长度为 LENGTH，数据范围在 0~LIMIT 的整形数组*/
	int i;
	for(i = 0; i < MAXSIZE; i++) {
		a[i] = rand() % INTERVAL + (i * INTERVAL); // 产生由小到大的随机数赋值给 b
	}
```

```
        return *a;
}

/* 生成顺随机不重复数组，测试用 */
int CreatArray(int *a) {
        int i;
        int tmp;
        int flag = 0;
        int t = 0;
        /* 随机数的个数不能超出 MAXSIZE 和 LIMIT 之间的范围，不然会导致死循 */
        while(1) {
                flag = 0;
                if(t == MAXSIZE + 1) {
                        break;
                }
                tmp = (rand() % (LIMIT - 0)) + 0;
                /* 0 号元素不赋值 */
                for(i = 1; i < t; i++) {
                        if(a[i] == tmp) {
                                flag = 1;
                        }
                }
                if(flag != 1) {
                        a[t] = tmp;
                        t++;
                }
        }
        return *a;
}
```

7.1.2. 实现

```
/* 顺序查找，a 为数组，n 为要查找的数组长度，key 为要查找的关键字 */
/* 平均时间复杂度:O(n)，当 n 很大时，查找效率低 */
int SequentialSearch(int *a, int n, KeyType key) {
        int i;
        for(i = 1; i <= n; i++) {
                if(a[i] == key) {
                        return i;
                }
        }
        return -1; // 没找到
}

/* 哨兵顺序查找，a 为数组，n 为要查找的数组长度，key 为要查找的关键字 */
/* 平均时间复杂度:O(n)；当 n 很大时，查找效率低 */
int GuardSequentialSearch(int *a, int n, KeyType key) {
```

```
        int i;
        a[0] = key;
        i = n;
        while(a[i] != key) {
                i--;
        }
        if(i == 0) {
                return -1; // 没找到
        }
        return i;
}

/* 折半查找，a 为数组，n 为要查找的数组长度，key 为要查找的关键字 */
/* 平均时间复杂度:O(logn)，前提条件:有序数列 */
int BinarySearch(int *a, int n, KeyType key) {
        int low, high, mid;
        low = 0; // 定义最低下标为记录首位
        high = n; // 定义最高下标为记录末位
        while(low <= high) {
                mid = (low + high) / 2; // 折半
                /* 若查找值比中值小 */
                if(key < a[mid]) {
                        high = mid - 1; // 最高下标调整到中位下标小一位
                }
                /* 若查找值比中值大 */
                else if(key > a[mid]) {
                        low = mid + 1; // 最低下标调整到中位下标大一位
                }
                else {
                        return mid + 1; // 若相等则说明 mid 即为查找到的位置
                }
        }
        return -1; // 没找到
}

/* 插值查找，a 为数组，n 为要查找的数组长度，key 为要查找的关键字 */
/* 平均时间复杂度:O(log(logn))，前提条件:已排序数;
        当数据分布比较均匀（成线性）时平均性能比折半查找要好得多，
        但当数据数据分布不均匀时，计算插值的代价比较大，性能不如二分查找 */
int InterpolationSearch(int *a, int n, KeyType key) {
        int low, high, mid;
        low = 0; // 定义最低下标为记录首位
        high = n; // 定义最高下标为记录末位
        while(low < high) {

        mid = low + (high - low) * (key - a[low]) / (a[high] - a[low]); // 插值
                /* 排除越界情况 */
                if(mid >= MAXSIZE) {
```

```
                return -1;
        }
        /* 若查找值比插值小 */
        if(key < a[mid]) {
                high = mid; // 最高下标调整到插值下标小一位
        }
        /* 若查找值比插值大 */
        else if(key > a[mid]) {
                low = mid + 1; // 最低下标调整到插值下标大一位
        }
        else {
                return mid + 1; // 若相等则说明 mid 即为查找到的位置
        }
    }
    return -1; // 没找到
}

/* 斐波那契查找，a 为数组，n 为要查找的数组长度，key 为要查找的关键字 */
/* 平均时间复杂度:O(logn)，前提条件:有序数列;
    只用到加减运算，查找海量数据时平均性能优于折半查找，
    但目标元素若都处于左侧长半区，则查找效率要低于折半查找 */
int FibonacciSearch(int *a, int n, KeyType key) {
    int F[MAXSIZE]; // 定义斐波那契数列
    int low, high, mid, i, k = 0;
    low = 0; // 定义最低下标为记录首位
    high = n - 1; // 定义最高下标为记录末位
    Fibonacci(F);
    while(n > F[k] - 1) {
        ++k;
    }
    int *tmp = (int*)malloc((F[k] - 1) * sizeof(int));
    memcpy(tmp, a, (n * sizeof(int))); // 将数组 a 扩展到 F[k]-1 的长度
    for(i = n; i < F[k] - 1; ++i) {
        tmp[i] = tmp[high];
    }
    while(low <= high) {
        mid = low + F[k - 1] - 1; // 黄金分割
        if(key < tmp[mid]) {
            high = mid - 1;
            k = k - 1;
        }
        else if(key > tmp[mid]) {
            low = mid + 1;
            k = k - 2;
        }
        else {
            if(mid <= high) {
                return mid + 1; // 若相等则说明 mid 即为查找到的位置
```

```
                }
                else {
                        return high + 1;
                }
            }
        }
        return -1; // 没找到
}
```

7.1.3. 测试

```
int main(int argc, char** argv) {
        int a[MAXSIZE + 1]; // 声明数组，0 号位置存放哨兵
        int b[MAXSIZE]; // 声明数组
        KeyType k; // 声明关键字 k
        int i, result; // 声明临时变量

        srand((unsigned int)time(NULL)); //设置随机数种子

        *a = CreatArray(a); // 随机不重复数组生成器
        printf("产生的范围在 0~%d 的随机数:\n", LIMIT);
        /* 打印随机不重复数组 */
        for(i = 1; i <= MAXSIZE; i++) {
                printf("%d: %d\t", i, a[i]);
        }
        printf("\n");

        printf("请输入要查找的关键字: ");
        scanf("%d", &k);

        result = SequentialSearch(a, MAXSIZE, k);
        printf("顺序查找 %d 的位置: %d \n", k, result);
        result = GuardSequentialSearch(a, MAXSIZE, k);
        printf("带哨兵顺序查找 %d 的位置: %d \n", k, result);
        printf("注: -1 表示没找到\n");

        printf("\n");
        *b = CreatSeqArray(b); // 随机顺序数组生成器
        printf("产生的范围在 0~%d，区间在%d 由小到大的随机数: \n", LIMIT, INTERVAL);
        /* 打印随机顺序数组 */
        for(i = 0; i < MAXSIZE; i++) {
                printf("%d: %d\t", i + 1, b[i]);
        }
        printf("\n");

        printf("请输入要查找的关键字: ");
        scanf("%d", &k);
```

```
        result = BinarySearch(b, MAXSIZE, k);
        printf("二分查找 %d 的位置：%d \n", k, result);
        result = InterpolationSearch(b, MAXSIZE, k);
        printf("插值查找 %d 的位置：%d \n", k, result);
        result = FibonacciSearch(b, MAXSIZE, k);
        printf("斐波那契查找 %d 的位置：%d \n", k, result);
        printf("注：-1 表示没找到\n");

        return 0;
}
```

7.2. 二叉排序树（BST）动态查找

7.2.1. 表示

```
#include <stdio.h>
#include <stdlib.h> // malloc、free、srand、rand
#include <time.h> // time

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 10 // 存储空间初始分配量
#define INTERVAL 3 // 随机数波动区间
#define LIMIT MAXSIZE * INTERVAL - 1 // 随机数范围

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef int KeyType; // 关键字的数据类型，类型根据实际情况而定，这里假设为 int

/* 定义二叉树二叉链表结点结构 */
typedef struct BiTNode {
        int data; // 结点数据
        struct BiTNode *lchild, *rchild; // 左右孩子指针
} BiTNode, *BiTree;

/* 从二叉排序树中删除结点 p，并重接它的左或右子树，供 DeleteBST()调用 */
Status Delete(BiTree *p) {
        BiTree q, s;
        /* 右子树空则只需重接它的左子树（待删结点是叶子也走此分支） */
        if((*p)->rchild == NULL) {
                q = *p;
                *p = (*p)->lchild;
```

```
                free(q);
        }
        /* 只需重接它的右子树 */
        else if((*p)->lchild == NULL) {
                q = *p;
                *p = (*p)->rchild;
                free(q);
        }
        /* 左右子树均不空 */
        else {
                q = *p;
                s = (*p)->lchild;
                /* 转左，然后向右到尽头（找待删结点的前驱） */
                while(s->rchild) {
                        q = s;
                        s = s->rchild;
                }
                (*p)->data = s->data; // s 指向被删结点的直接前驱（将被删结点前驱的值取代被删结点的值）
                if(q != *p) {
                        q->rchild = s->lchild; // 重接 q 的右子树
                }
                else {
                        q->lchild = s->lchild; // 重接 q 的左子树
                }
                free(s);
        }
        return TRUE;
}

/* 初始条件：二叉树 T 存在 */
/* 操作结果：前序递归遍历 T */
Status PreOrderTraverse(BiTree T) {
        if(T == NULL) {
                return ERROR;
        }
        printf("%d ", T->data); // 显示结点数据
        PreOrderTraverse(T->lchild); // 再先序遍历左子树
        PreOrderTraverse(T->rchild); // 最后先序遍历右子树
        return OK;
}

/* 初始条件：二叉树 T 存在 */
/* 操作结果：中序递归遍历 T */
Status InOrderTraverse(BiTree T) {
        if(T == NULL) {
                return ERROR;
        }
```

```
        InOrderTraverse(T->lchild); // 中序遍历左子树
        printf("%d ", T->data); // 显示结点数据
        InOrderTraverse(T->rchild); // 最后中序遍历右子树
        return OK;
}

/* 初始条件：二叉树 T 存在 */
/* 操作结果：后序递归遍历 T */
Status PostOrderTraverse(BiTree T) {
        if(T == NULL) {
                return ERROR;
        }
        PostOrderTraverse(T->lchild); // 先后序遍历左子树
        PostOrderTraverse(T->rchild); // 再后序遍历右子树
        printf("%d ", T->data); // 显示结点数据，可以更改为其它对结点操作
        return OK;
}

/* 生成顺随机不重复数组，测试用 */
KeyType CreatArray(KeyType *a) {
        int i;
        KeyType tmp;
        int flag = 0;
        int t = 0;
        /* 随机数的个数不能超出 MAXSIZE 和 LIMIT 之间的范围，不然会导致死循 */
        while(1) {
                flag = 0;
                if (t == MAXSIZE) {
                        break;
                }
                tmp = (rand() % (LIMIT - 0)) + 0;
                for(i = 0; i < t; i++) {
                        if(a[i] == tmp) {
                                flag = 1;
                        }
                }
                if (flag != 1) {
                        a[t] = tmp;
                        t++;
                }
        }
        return *a;
}
```

7.2.2. 实现

```
/* 递归查找二叉排序树 T 中是否存在 key */
```

```c
/* 指针 f 指向 T 的双亲，其初始调用值为 NULL
若查找成功，则指针 p 指向该数据元素结点，并返回 TRUE;
否则指针 p 指向查找路径上访问的最后一个结点并返回 FALSE */
Status SearchBST(BiTree T, int key, BiTree f, BiTree *p) {
	/* 查找不成功 */
	if(!T) {
		*p = f;
		return FALSE;
	}
	/* 查找成功 */
	else if(key == T->data) {
		*p = T;
		return TRUE; // 递归出口
	}
	else if(key < T->data) {
		return SearchBST(T->lchild, key, T, p); // 在左子树中继续查找
	}
	else {
		return SearchBST(T->rchild, key, T, p); // 在右子树中继续查找
	}
}

/* 当二叉排序树 T 中不存在关键字等于 key 的数据元素时 */
/* 插入 key 并返回 TRUE，否则返回 FALSE */
Status InsertBST(BiTree *T, KeyType key) {
	BiTree p, s;
	/* 查找不成功 */
	if(!SearchBST(*T, key, NULL, &p)) {
		s = (BiTree)malloc(sizeof(BiTNode));
		s->data = key;
		s->lchild = s->rchild = NULL;
		if(!p) {
			*T = s; // 插入 s 为新的根结点
		}
		else if(key < p->data) {
			p->lchild = s; // 插入 s 为左孩子
		}
		else {
			p->rchild = s; // 插入 s 为右孩子
		}
		return TRUE;
	}
	else {
		return FALSE; // 树中已有关键字相同的结点，不再插入
	}
}

/* 若二叉排序树 T 中存在关键字等于 key 的数据元素时，则删除该数据元素结点，*/
```

```
/* 并返回 TRUE；否则返回 FALSE。 */
Status DeleteBST(BiTree *T, KeyType key) {
	/* 不存在关键字等于 key 的数据元素 */
	if(!*T) {
		return FALSE;
	}
	else {
		/* 找到关键字等于 key 的数据元素 */
		if(key == (*T)->data) {
			return Delete(T);
		}
		else if(key < (*T)->data) {
			return DeleteBST(&(*T)->lchild, key);
		}
		else {
			return DeleteBST(&(*T)->rchild, key);
		}
	}
}
```

7.2.3. 测试

```
int main(int argc, char** argv) {
	BiTree T = NULL, f = NULL, p; // 声明 BST 结点
	KeyType a[MAXSIZE]; // 声明数组
	KeyType k; // 声明关键字 k
	int i; // 声明临时变量

	srand((unsigned int)time(NULL)); //设置随机数种子

	*a = CreatArray(a);
	printf("随机生成的不重复数组：");
	for (i = 0; i < MAXSIZE; i++) {
		printf("%d ", a[i]);
	}

	for(i = 0; i<MAXSIZE; i++) {
		InsertBST(&T, a[i]); // 将随机生成的数组插入到 BST 树里
	}
	printf("\n\n 插入后的 BST 前序遍历：");
	PreOrderTraverse(T);
	printf("\n 插入后的 BST 中序遍历：");
	InOrderTraverse(T);
	printf("\n 插入后的 BST 后序遍历：");
	PostOrderTraverse(T);
	printf("\n");
```

```
        printf("请输入要删除的数据:");
        scanf("%d", &k);
        DeleteBST(&T, k);
        printf("删除 %d 后的 BST 前序遍历: ", k);
        PreOrderTraverse(T);
        printf("\n 删除 %d 后的 BST 中序遍历: ", k);
        InOrderTraverse(T);
        printf("\n 删除 %d 后的 BST 后序遍历: ", k);
        PostOrderTraverse(T);
        printf("\n");

        printf("请输入要插入的数据:");
        scanf("%d", &k);
        InsertBST(&T, k);
        printf("插入 %d 后的 BST 前序遍历: ", k);
        PreOrderTraverse(T);
        printf("\n 插入 %d 后的 BST 中序遍历: ", k);
        InOrderTraverse(T);
        printf("\n 插入 %d 后的 BST 后序遍历: ", k);
        PostOrderTraverse(T);
        printf("\n");

        printf("\n 请输入要查找的数据元素: ");
        scanf("%d", &k);
        i = SearchBST(T, k, f, &p);
        if(i == TRUE) {
                printf(" %d 在树里\n", p->data);
        }
        else {
                printf("不在树里\n");
        }

        return 0;
}
```

7.3. 平衡二叉树（AVL 树）动态查找

7.3.1. 表示

```
#include <stdio.h>
#include <stdlib.h> // malloc、free、srand、rand
#include <time.h> // time

/* 状态码 */
#define OK 1
#define ERROR 0
```

```c
#define TRUE 1
#define FALSE 0

#define MAXSIZE 10 // 存储空间初始分配量
#define INTERVAL 3 // 随机数波动区间
#define LIMIT MAXSIZE * INTERVAL - 1 // 随机数范围

/* AVL 树的平衡因子 */
#define LH +1 // 左高
#define EH 0 // 等高
#define RH -1 // 右高

typedef int Status; // Status 是函数的类型,其值是函数结果状态代码，如 OK 等
typedef int KeyType; // 关键字的数据类型，类型根据实际情况而定，这里假设为 int

/* 二叉树的二叉链表结点结构定义 */
typedef  struct BiTNode {
        KeyType data; // 结点数据
        int bf; // 结点的平衡因子
        struct BiTNode *lchild, *rchild; // 左右孩子指针
} BiTNode, *BiTree;

/* 对以 P 为根的二叉排序树作左旋处理，处理之后 P 指向新的树根结点，
        即旋转处理之前的右子树的根结点 0  */
void L_Rotate(BiTree *P) {
        BiTree R;
        R = (*P)->rchild; // R 指向 P 的右子树根结点
        (*P)->rchild = R->lchild; // R 的左子树挂接为 P 的右子树
        R->lchild = (*P);
        *P = R; // P 指向新的根结点
}

/* 对以 p 为根的二叉排序树作右旋处理，处理之后 p 指向新的树根结点，
        即旋转处理之前的左子树的根结点 */
void R_Rotate(BiTree *P) {
        BiTree L;
        L = (*P)->lchild; // L 指向 P 的左子树根结点
        (*P)->lchild = L->rchild; // L 的右子树挂接为 P 的左子树
        L->rchild = (*P);
        *P = L; // P 指向新的根结点
}

/* 对以指针 T 所指结点为根的二叉树作左平衡旋转处理，
        本算法结束时，指针 T 指向新的根结点 */
void LeftBalance(BiTree *T) {
        BiTree L, Lr;
        L = (*T)->lchild; // L 指向 T 的左子树根结点
        /*  检查 T 的左子树的平衡度，并作相应平衡处理 */
```

```
        switch(L->bf) {
        case LH: // 新结点插入在 T 的左孩子的左子树上，要作单右旋处理
                (*T)->bf = EH;
                L->bf = EH;
                R_Rotate(T);
                break;
        case EH:
                (*T)->bf = LH;
                L->bf = RH;
                R_Rotate(T);
                break;
        case RH: // 新结点插入在 T 的左孩子的右子树上，要作双旋处理
                Lr = L->rchild; // Lr 指向 T 的左孩子的右子树根
                /* 修改 T 及其左孩子的平衡因子 */
                switch(Lr->bf) {
                case LH:
                        (*T)->bf = RH;
                        L->bf = EH;
                        break;
                case EH:
                        (*T)->bf = EH;
                        L->bf = EH;
                        break;
                case RH:
                        (*T)->bf = EH;
                        L->bf = LH;
                        break;
                default:
                        break;
                }
                Lr->bf = EH;
                L_Rotate(&(*T)->lchild); // 对 T 的左子树作左旋平衡处理
                R_Rotate(T); // 对 T 作右旋平衡处理
                break;
        default:
                break;
        }
}

/* 对以指针 T 所指结点为根的二叉树作右平衡旋转处理，
        本算法结束时，指针 T 指向新的根结点 */
void RightBalance(BiTree *T) {
        BiTree R, Rl;
        R = (*T)->rchild; // R 指向 T 的右子树根结点
        /* 检查 T 的右子树的平衡度，并作相应平衡处理 */
        switch(R->bf) {
        case RH: // 新结点插入在 T 的右孩子的右子树上，要作单左旋处理
                (*T)->bf = EH;
```

```
                R->bf = EH;
                L_Rotate(T);
                break;
        case EH:
                (*T)->bf = RH;
                R->bf = LH;
                L_Rotate(T);
                break;
        case LH: // 新结点插入在 T 的右孩子的左子树上，要作双旋处理
                Rl = R->lchild; // Rl 指向 T 的右孩子的左子树根
                /* 修改 T 及其右孩子的平衡因子 */
                switch(Rl->bf) {
                case RH:
                        (*T)->bf = LH;
                        R->bf = EH;
                        break;
                case EH:
                        (*T)->bf = EH;
                        R->bf = EH;
                        break;
                case LH:
                        (*T)->bf = EH;
                        R->bf = RH;
                        break;
                default:
                        break;
                }
                Rl->bf = EH;
                R_Rotate(&(*T)->rchild); // 对 T 的右子树作右旋平衡处理
                L_Rotate(T); // 对 T 作左旋平衡处理
                break;
        default:
                break;
        }
}

/* 寻找最小元素，供 DeleteAVL()调用 */
KeyType MinElement(BiTNode *p) {
        if(p->lchild) {
                return MinElement(p->lchild);
        }
        else {
                return p->data;
        }
}

/* 生成顺随机不重复数组，测试用 */
KeyType CreatArray(KeyType *a) {
```

```
        int i;
        KeyType tmp;
        int flag = 0;
        int t = 0;
        /* 随机数的个数不能超出 MAXSIZE 和 LIMIT 之间的范围，不然会导致死循 */
        while (1) {
                flag = 0;
                if (t == MAXSIZE) {
                        break;
                }
                tmp = (rand() % (LIMIT - 0)) + 0;
                for (i = 0; i < t; i++) {
                        if (a[i] == tmp) {
                                flag = 1;
                        }
                }
                if (flag != 1) {
                        a[t] = tmp;
                        t++;
                }
        }
        return *a;
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 前序递归遍历 T */
Status PreOrderTraverse(BiTree T) {
        if (T == NULL) {
                return ERROR;
        }
        printf("%d ", T->data); // 显示结点数据
        PreOrderTraverse(T->lchild); // 再先序遍历左子树
        PreOrderTraverse(T->rchild); // 最后先序遍历右子树
        return OK;
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 中序递归遍历 T */
Status InOrderTraverse(BiTree T) {
        if (T == NULL) {
                return ERROR;
        }
        InOrderTraverse(T->lchild); // 中序遍历左子树
        printf("%d ", T->data); // 显示结点数据
        InOrderTraverse(T->rchild); // 最后中序遍历右子树
        return OK;
}
```

```c
/* 初始条件：二叉树 T 存在 */
/* 操作结果：后序递归遍历 T */
Status PostOrderTraverse(BiTree T) {
	if (T == NULL) {
		return ERROR;
	}
	PostOrderTraverse(T->lchild); // 先后序遍历左子树
	PostOrderTraverse(T->rchild); // 再后序遍历右子树
	printf("%d ", T->data); // 显示结点数据，可以更改为其它对结点操作
	return OK;
}
```

7.3.2. 实现

```c
/* 查找 AVL 树 T 中是否存在 k */
Status SearchAVL(BiTree T, KeyType k) {
	BiTree p = T;
	while (p) {
		if (p->data == k) {
			return TRUE;
		}
		else if (p->data < k) {
			p = p->rchild;
		}
		else {
			p = p->lchild;
		}
	}
	return FALSE;
}

/* 若在平衡的二叉排序树 T 中不存在和 key 有相同关键字的结点，
	则插入一个数据元素值为 key 的新结点，并返回 TRUE，否则返回 FALSE。
	若因插入而使二叉排序树失去平衡，则作平衡旋转处理，变量 taller 反映 T 长高与否 */
Status InsertAVL(BiTree *T, KeyType key, Status *taller) {
	/* 插入新结点，树“长高”，置 taller 为 TRUE */
	if(!*T) {
		*T = (BiTree)malloc(sizeof(BiTNode));
		(*T)->data = key;
		(*T)->lchild = NULL;
		(*T)->rchild = NULL;
		(*T)->bf = EH;
		*taller = TRUE;
		return TRUE;
	}
	else {
		/* 树中已存在和 e 有相同关键字的结点则不再插入 */
```

```
if(key == (*T)->data) {
        *taller = FALSE;
        return FALSE;
}
/* 应继续在 T 的左子树中进行搜索 */
if(key < (*T)->data) {
        /* 未插入 */
        if(!InsertAVL(&(*T)->lchild, key, taller)) {
                return FALSE;
        }
        /* 已插入到 T 的左子树中且左子树“长高” */
        if(*taller) {
                /* 检查 T 的平衡度 */
                switch((*T)->bf) {
                case LH: // 原本左子树比右子树高，需要作左平衡处理
                        LeftBalance(T);
                        *taller = FALSE;
                        break;
                case EH: // 原本左、右子树等高，现因左子树增高而使树增高
                        (*T)->bf = LH;
                        *taller = TRUE;
                        break;
                case RH: // 原本右子树比左子树高，现左、右子树等高
                        (*T)->bf = EH;
                        *taller = FALSE;
                        break;
                default:
                        break;
                }
        }
}
/* 应继续在 T 的右子树中进行搜索 */
else {
        /* 未插入 */
        if(!InsertAVL(&(*T)->rchild, key, taller)) {
                return FALSE;
        }
        /* 已插入到 T 的右子树且右子树“长高” */
        if(*taller) {
                /* 检查 T 的平衡度 */
                switch((*T)->bf) {
                case LH: // 原本左子树比右子树高，现左、右子树等高
                        (*T)->bf = EH;
                        *taller = FALSE;
                        break;
                case EH: // 原本左、右子树等高，现因右子树增高而使树增高
                        (*T)->bf = RH;
                        *taller = TRUE;
```

```
                        break;
                    case RH: // 原本右子树比左子树高，需要作右平衡处理
                        RightBalance(T);
                        *taller = FALSE;
                        break;
                    default:
                        break;
                    }
                }
            }
        }
        return TRUE;
}

/* 若在平衡的二叉排序树 T 中存在和 key 有相同关键字的结点，
        则删除数据元素值为 key 的结点，并返回 TRUE，否则返回 FALSE。
        若因删除而使二叉排序树失去平衡，则作平衡旋转处理，变量 shorter 反映 T 变矮与否 */
Status DeleteAVL(BiTree *T, KeyType key, Status *shorter) {
        /* 不存在该元素 */
        if(*T == NULL) {
            return FALSE; // 删除失败
        }
        /* 待删结点小于当前结点 */
        if(key < (*T)->data) {
            DeleteAVL(&((*T)->lchild), key, shorter);
            /* 已删除结点，当左子树变短时，进行平衡判断 */
            if(*shorter) {
                switch ((*T)->bf) {
                case LH: // 原来左高，现在左子树-1，所以等高
                    (*T)->bf = EH;
                    *shorter = TRUE;
                    break;
                case EH: // 原来等高，现在左子树-1，所以右高
                    (*T)->bf = RH;
                    *shorter = FALSE;
                    break;
                case RH: // 原来右高，现在左子树-1，所以需要左旋，调整平衡
                    LeftBalance(T);
                    *shorter = TRUE;
                    break;
                default:
                    break;
                }
            }
        }
        /*待删结点大于当前结点*/
        else if(key > (*T)->data) {
            DeleteAVL(&((*T)->rchild), key, shorter);
```

```
        /* 已删除结点，当左子树变短时，进行平衡判断 */
        if(*shorter) {
                switch((*T)->bf) {
                case LH: // 原来左高，现在右子树-1，所以需要右旋，调整平衡
                        LeftBalance(T);
                        *shorter = TRUE;
                        break;
                case EH: // 原来等高，现在右子树-1，所以左高
                        (*T)->bf = LH;
                        *shorter = FALSE;
                        break;
                case RH: // 原来右高，现在右子树-1，所以等高
                        (*T)->bf = EH;
                        *shorter = TRUE;
                        break;
                default:
                        break;
                }
        }
    }
    /* 待删结点等于当前结点 */
    else {
        /* 如果待删结点有右子树 */
        if((*T)->rchild) {
                (*T)->data = MinElement((*T)->rchild);
                DeleteAVL(&((*T)->rchild), (*T)->data, shorter);
                /* 已删除结点，当右子树变短时，进行平衡判断 */
                if(shorter) {
                        switch((*T)->bf) {
                        case LH: // 原来左高，现在右子树-1，所以需要右旋，调整平
衡
                                LeftBalance(T);
                                *shorter = TRUE;
                                break;
                        case EH: // 原来等高，现在右子树-1，所以左高
                                (*T)->bf = LH;
                                *shorter = FALSE;
                                break;
                        case RH: // 原来右高，现在右子树-1，所以等高
                                (*T)->bf = EH;
                                *shorter = TRUE;
                                break;
                        default:
                                break;
                        }
                }
        }
        /* 若当前结点仅有左子树，则该结点只可能在倒数第二层，无需判断是否失衡 */
```

```
            else if((*T)->lchild) {
                    (*T)->data = (*T)->lchild->data;
                    (*T)->lchild = NULL;
                    free((*T)->lchild);
            }
            /* 该结点为叶子结点 */
            else {
                    free(*T);
                    *T = NULL;
                    *shorter = TRUE;
            }
    }
    return TRUE;
}
```

7.3.3. 测试

```
int main(int argc, char** argv) {
    BiTree T = NULL; // 声明二叉树 T
    KeyType a[MAXSIZE]; // 声明数组
    KeyType k; // 声明关键字 k
    Status taller, shorter; // 声明树高变化变量
    int i; // 声明临时变量

    srand((unsigned int)time(NULL)); //设置随机数种子

    *a = CreatArray(a);
    printf("随机生成的不重复数组：");
    for (i = 0; i < MAXSIZE; i++) {
            printf("%d ", a[i]);
    }

    for (i = 0; i < MAXSIZE; i++) {
            InsertAVL(&T, a[i], &taller);
    }
    printf("\n\n 插入后的 AVL 树 前序遍历：");
    PreOrderTraverse(T);
    printf("\n 插入后的 AVL 树 中序遍历：");
    InOrderTraverse(T);
    printf("\n 插入后的 AVL 树 后序遍历：");
    PostOrderTraverse(T);
    printf("\n");

    printf("\n 请输入要删除的结点：");
    scanf("%d", &k);
    DeleteAVL(&T, k, &shorter);
    printf("删除 %d 后的 AVL 树 前序遍历：", k);
```

```
        PreOrderTraverse(T);
        printf("\n 删除 %d 后的 AVL 树 中序遍历: ", k);
        InOrderTraverse(T);
        printf("\n 删除 %d 后的 AVL 树 后序遍历: ", k);
        PostOrderTraverse(T);
        printf("\n");

        printf("\n 请输入要插入的数据元素: ");
        scanf("%d", &k);
        InsertAVL(&T, k, &taller);
        printf("插入 %d 后的 AVL 树 前序遍历: ", k);
        PreOrderTraverse(T);
        printf("\n 插入 %d 后的 AVL 树 中序遍历: ", k);
        InOrderTraverse(T);
        printf("\n 插入 %d 后的 AVL 树 后序遍历: ", k);
        PostOrderTraverse(T);
        printf("\n");

        printf("\n 请输入要查找的数据元素: ");
        scanf("%d", &k);
        i = SearchAVL(T, k);
        if(i == TRUE) {
                printf(" %d 在树里\n", k);
        }
        else {
                printf("不在树里\n");
        }

        return 0;
}
```

7.4. 哈希表（散列表）动态查找

7.4.1. 表示

```
#include <stdio.h>
#include <stdlib.h> // malloc、srand、rand
#include <time.h> // time

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 // 数据元素存储空间初始分配量
```

```c
#define HASHSIZE 15 // 哈希表长度
#define NULLKEY -0x3f3f3f3f // 哈希表单元为空的标记

#define INTERVAL 3 // 随机数波动区间
#define LIMIT HASHSIZE * INTERVAL - 1 // 随机数范围

typedef int Status; // Status 是函数的类型，其值是函数结果状态代码，如 OK 等
typedef int KeyType; // 关键字的数据类型，类型根据实际情况而定，这里假设为 int

typedef struct {
	KeyType *elem; // 储存数据元素
	int count; // 当前已存进哈希表的数据元素个数
}HashTable;

/* 哈希函数，返回关键字 key 的地址，InsertHash()调用 */
int Hash(KeyType key) {
	return key % HASHSIZE; // 除留余数法
}

/* 生成顺随机不重复数组，测试用 */
KeyType CreatArray(KeyType *a) {
	int i;
	KeyType tmp;
	int flag = 0;
	int t = 0;
	/* 随机数的个数不能超出 MAXSIZE 和 LIMIT 之间的范围，不然会导致死循 */
	while(1) {
		flag = 0;
		if(t == HASHSIZE) {
			break;
		}
		tmp = (rand() % (LIMIT - 0)) + 0;
		for(i = 0; i < t; i++) {
			if(a[i] == tmp) {
				flag = 1;
			}
		}
		if(flag != 1) {
			a[t] = tmp;
			t++;
		}
	}
	return *a;
}
```

7.4.2. 实现

```
/* 初始化哈希表 H */
Status InitHashTable(HashTable *H) {
	int i;
	H->count = 0;
	H->elem = (KeyType *)malloc(HASHSIZE * sizeof(KeyType));
	for(i = 0; i < HASHSIZE; i++) {
		H->elem[i] = NULLKEY;
	}
	return OK;
}

/* 将关键字 key 插入哈希表 H 中 */
Status InsertHash(HashTable *H, KeyType key) {
	if(H->count >= HASHSIZE) {
		return ERROR;
	}
	int addr = Hash(key); // 求哈希地址
	/* 如果不为空，则冲突 */
	while(H->elem[addr] != NULLKEY) {
		addr = (addr + 1) % HASHSIZE; // 开放定址法的线性探测
	}
	H->elem[addr] = key; // 直到有空位后插入关键字
	H->count += 1;
	return OK;
}

/* 在哈希表 H 中查找关键字 key，若成功，用*addr 返回位置，函数返回 OK */
Status SearchHash(HashTable H, KeyType key, int *addr) {
	*addr = Hash(key); // 求哈希地址
	/* 如果不为空，则冲突 */
	while(H.elem[*addr] != key) {
		*addr = (*addr + 1) % HASHSIZE; // 开放定址法的线性探测
		/* 如果循环回到原点 */
		if(H.elem[*addr] == NULLKEY || *addr == Hash(key)) {
			return FALSE; // 则说明关键字不存在
		}
	}
	*addr += 1;
	return OK;
}

/* 打印哈希表 H，若某位置中没有数据元素，则在该位置打印 NaN */
Status PrintHash(HashTable H) {
	int i;
	printf("哈希表:  ");
	for(i = 0; i < HASHSIZE; i++) {
```

```
            if(H.elem[i] == NULLKEY) {
                    printf("NaN ");
            }
            else {
                    printf("%d ", H.elem[i]);
            }
        }
        printf("\n");
        return OK;
}
```

7.4.3. 测试

```
int main(int argc, char** argv) {
        HashTable H; // 声明哈希表 H
        KeyType a[MAXSIZE]; // 声明数组
        KeyType k; // 声明关键字 key
        Status result; // 声明状态码
        int i, p; // 声明临时变量

        srand((unsigned int)time(NULL)); //设置随机数种子

        *a = CreatArray(a);
        printf("随机生成的不重复数组: ");
        for(i = 0; i < 10; i++) {
                printf("%d ", a[i]);
        }
        printf("\n");

        InitHashTable(&H);
        for(i = 0; i < 10; i++) {
                InsertHash(&H, a[i]); // 把随机生成的 10 个数存进去
        }

        PrintHash(H);

        printf("请输入要插入的数据元素: ");
        scanf("%d", &k);
        InsertHash(&H, k);
        PrintHash(H);

        printf("请输入要查找的数据元素: ");
        scanf("%d", &k);
        result = SearchHash(H, k, &p);
        if(result) {
                printf("查找 %d 的位置:  %d \n", k, p);
        }
```

```
        else {
                printf("查找 %d 失败 \n", k);
        }

        return 0;
}
```

8. 排序

8.1. 排序

8.1.1. 表示

```c
#include <stdio.h>
#include <stdlib.h> // malloc、srand、rand
#include <time.h> // time

/* 状态码 */
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 // 用于要排序数组个数最大值，可根据需要修改
#define INTERVAL 4 // 随机数波动区间
#define LIMIT MAXSIZE * INTERVAL - 1 // 随机数范围

#define MAX_LENGTH_INSERT_SORT 7 // 用于快速排序时判断是否选用插入排序阈值

typedef int Status; // Status 是函数的类型，其值是函数结果状态代码，如 OK 等
typedef int KeyType; // 关键字的数据类型，类型根据实际情况而定，这里假设为 int

typedef struct {
	KeyType r[MAXSIZE + 1]; // 用于存储要排序数组，r[0]用作哨兵或暂存单元
	int length; // 用于记录顺序表的长度
}SqList;

void InsertSort(SqList *L); // 插入排序，OpQuickSort()调用

/* 交换 L 中数组 r 的下标为 i 和 j 的值，供其他函数调用 */
void swap(SqList *L, KeyType i, KeyType j) {
	KeyType temp = L->r[i];
	L->r[i] = L->r[j];
	L->r[j] = temp;
}

/* 堆排序 HeapSort()调用 */
/* 已知 L->r[s..m]中记录的关键字除 L->r[s]之外均满足堆的定义， */
```

```
/* 本函数调整 L->r[s]的关键字,使 L->r[s..m]成为一个大顶堆 */
void HeapAdjust(SqList *L, KeyType s, KeyType m) {
	int temp, j;
	temp = L->r[s];
	/* 沿关键字较大的孩子结点向下筛选 */
	for(j = 2 * s; j <= m; j *= 2) {
		if(j < m && L->r[j] < L->r[j + 1]) {
			++j; // j 为关键字中较大的记录的下标
		}
		if(temp >= L->r[j]) {
			break; // rc 应插入在位置 s 上
		}
		L->r[s] = L->r[j];
		s = j;
	}
	L->r[s] = temp; // 插入
}

/* 递归归并排序内部接口，MSort()调用 */
/* 将有序的 SR[i..m]和 SR[m+1..n]归并为有序的 TR[i..n] */
void Merge(KeyType SR[], KeyType TR[], KeyType i, KeyType m, KeyType n) {
	int j, k, l;
	/* 将 SR 中记录由小到大地并入 TR */
	for(j = m + 1, k = i; i <= m && j <= n; k++) {
		if(SR[i] < SR[j]) {
			TR[k] = SR[i++];
		}
		else {
			TR[k] = SR[j++];
		}
	}
	if(i <= m) {
		for(l = 0; l <= m - i; l++) {
			TR[k + l] = SR[i + l]; // 将剩余的 SR[i..m]复制到 TR
		}
	}
	if(j <= n) {
		for(l = 0; l <= n - j; l++) {
			TR[k + l] = SR[j + l]; // 将剩余的 SR[j..n]复制到 TR
		}
	}
}

/* 递归归并排序 ReMergeSort()调用 */
/* 将 SR[s..t]归并排序为 TR1[s..t] */
void MSort(KeyType SR[], KeyType TR1[], KeyType s, KeyType t) {
	KeyType m;
	KeyType TR2[MAXSIZE + 1];
```

```
        if(s == t) {
                TR1[s] = SR[s];
        }
        else {
                m = (s + t) / 2; // 将 SR[s..t]平分为 SR[s..m]和 SR[m+1..t]
                MSort(SR, TR2, s, m); // 递归地将 SR[s..m]归并为有序的 TR2[s..m]
                MSort(SR, TR2, m + 1, t); // 递归地将 SR[m+1..t]归并为有序的
TR2[m+1..t]
                Merge(TR2, TR1, s, m, t); // 将 TR2[s..m]和 TR2[m+1..t]归并到
TR1[s..t]
        }
}

/* 非递归归并排序 MergeSort()调用 */
/* 将 SR[]中相邻长度为 s 的子序列两两归并到 TR[] */
void MergePass(KeyType SR[], KeyType TR[], KeyType s, KeyType n) {
        int i = 1;
        int j;
        /* 两两归并 */
        while(i <= n - 2 * s + 1) {
                Merge(SR, TR, i, i + s - 1, i + 2 * s - 1);
                i = i + 2 * s;
        }
        /* 归并最后两个序列 */
        if(i < n - s + 1) {
                Merge(SR, TR, i, i + s - 1, n);
        }
        /* 若最后只剩下单个子序列 */
        else{
                for(j = i; j <= n; j++) {
                        TR[j] = SR[j];
                }
        }
}

/* 经典快速排序内部接口，QSort()调用 */
/* 交换顺序表 L 中子表的记录，使枢轴记录到位，并返回其所在位置 */
/* 此时在它之前(后)的记录均不大(小)于它。 */
int Partition(SqList *L, int low, int high) {
        int pivotkey;
        pivotkey = L->r[low]; // 用子表的第一个记录作枢轴记录
        /* 从表的两端交替地向中间扫描 */
        while(low < high) {
                while(low < high && L->r[high] >= pivotkey) {
                        high--;
                }
                swap(L, low, high); // 将比枢轴记录小的记录交换到低端
                while(low < high&&L->r[low] <= pivotkey) {
```

```
                low++;
            }
            swap(L, low, high); // 将比枢轴记录大的记录交换到高端
        }
        return low; // 返回枢轴所在位置
}

/* 经典快速排序 QuickSort()调用 */
/* 对顺序表 L 中的子序列 L->r[low..high]作快速排序 */
void QSort(SqList *L, int low, int high) {
        int pivot;
        if(low < high) {
            pivot = Partition(L, low, high); // 将 L->r[low..high]一分为二，算出枢轴值 pivot
            QSort(L, low, pivot - 1); // 对低子表递归排序
            QSort(L, pivot + 1, high); // 对高子表递归排序
        }
}

/* 优化快速排序优化内部接口，OpQSort()调用 */
int OpPartition(SqList *L, int low, int high) {
        int pivotkey;
        int m = low + (high - low) / 2; // 计算数组中间的元素的下标
        if(L->r[low] > L->r[high]) {
            swap(L, low, high); // 交换左端与右端数据，保证左端较小
        }
        if(L->r[m] > L->r[high]) {
            swap(L, high, m); // 交换中间与右端数据，保证中间较小
        }
        if(L->r[m] > L->r[low]) {
            swap(L, m, low); // 交换中间与左端数据，保证左端较小
        }
        pivotkey = L->r[low]; // 用子表的第一个记录作枢轴记录
        L->r[0] = pivotkey; // 将枢轴关键字备份到 L->r[0]
        /* 从表的两端交替地向中间扫描 */
        while (low < high) {
            while(low < high && L->r[high] >= pivotkey) {
                high--;
            }
            L->r[low] = L->r[high];
            while(low < high && L->r[low] <= pivotkey) {
                low++;
            }
            L->r[high] = L->r[low];
        }
        L->r[low] = L->r[0];
        return low; // 返回枢轴所在位置
}
```

```
/* 优化快速排序 OpQuickSort()调用 */
void OpQSort(SqList *L, int low, int high) {
	int pivot;
	if((high - low) > MAX_LENGTH_INSERT_SORT) {
		while(low < high) {
			pivot = OpPartition(L, low, high); // 将 L->r[low..high]一分为二，算出枢轴值 pivot
			OpQSort(L, low, pivot - 1); // 对低子表递归排序

	// QSort(L,pivot+1,high); // 对高子表递归排序
			low = pivot + 1; // 尾递归
		}
	}
	else {
		InsertSort(L);
	}
}

/* 打印函数，测试用 */
void Print(SqList L) {
	int i;
	for(i = 1; i <= L.length; i++) {
		printf("%d ", L.r[i]);
	}
	printf("\n");
}

/* 生成顺随机不重复数组，测试用 */
KeyType CreatArray(KeyType *a) {
	int i;
	/* 0 号元素不赋值 */
	for(i = 1; i <= MAXSIZE; i++) {
		a[i] = (rand() % (LIMIT - 0)) + 0;
	}
	return *a;
}
```

8.1.2. 实现

```
/* 对顺序表 L 作简单交换排序（简单冒泡排序）*/
void SwapSort(SqList *L) {
	int i, j;
	for(i = 1; i < L->length; i++) {
		for(j = i + 1; j <= L->length; j++) {
			if(L->r[i] > L->r[j]) {
				swap(L, i, j); // 交换 L->r[i]与 L->r[j]的值
```

```
                }
            }
        }
}

/* 对顺序表 L 作普通冒泡排序 */
void BubbleSort(SqList *L) {
        int i, j;
        for(i = 1; i < L->length; i++) {
                /* j 是从后往前循环 */
                for(j = L->length - 1; j >= i; j--) {
                        /* 若前者大于后者（注意这里与上一算法的差异）*/
                        if(L->r[j] > L->r[j + 1]) {
                                swap(L, j, j + 1); // 交换 L->r[j]与 L->r[j+1]的值
                        }
                }
        }
}

/* 对顺序表 L 作标志位冒泡排序 */
void FlagBubbleSort(SqList *L) {
        int i, j;
        Status flag = TRUE; // flag 用来作为标记
        /* 若 flag 为 true 说明有过数据交换，否则停止循环 */
        for(i = 1; i < L->length && flag; i++) {
                flag = FALSE; // 初始为 False
                for(j = L->length - 1; j >= i; j--) {
                        if(L->r[j] > L->r[j + 1]) {
                                swap(L, j, j + 1); // 交换 L->r[j]与 L->r[j+1]的值
                                flag = TRUE; // 如果有数据交换，则 flag 为 true
                        }
                }
        }
}

/* 对顺序表 L 作简单选择排序 */
void SelectSort(SqList *L) {
        int i, j, min;
        for(i = 1; i < L->length; i++) {
                min = i; // 将当前下标定义为最小值下标
                /* 循环之后的数据 */
                for(j = i + 1; j <= L->length; j++) {
                        /* 如果有小于当前最小值的关键字 */
                        if(L->r[min] > L->r[j]) {
                                min = j; // 将此关键字的下标赋值给 min
                        }
                }
                /* 若 min 不等于 i，说明找到最小值，交换 */
```

```
                if (i != min) {
                        swap(L, i, min); // 交换 L->r[i]与 L->r[min]的值
                }
        }
}

/* 对顺序表 L 作直接插入排序 */
void InsertSort(SqList *L) {
        int i, j;
        for(i = 2; i <= L->length; i++) {
                /* 需将 L->r[i]插入有序子表 */
                if(L->r[i] < L->r[i - 1]) {
                        L->r[0] = L->r[i]; // 设置哨兵
                        for(j = i - 1; L->r[j] > L->r[0]; j--) {
                                L->r[j + 1] = L->r[j]; // 记录后移
                        }
                        L->r[j + 1] = L->r[0]; // 插入到正确位置
                }
        }
}

/* 对顺序表 L 作希尔排序 */
void ShellSort(SqList *L) {
        int i, j, k = 0;
        int increment = L->length;
        do {
                increment = increment / 3 + 1; // 增量序列
                for(i = increment + 1; i <= L->length; i++) {
                        /* 需将 L->r[i]插入有序增量子表 */
                        if(L->r[i] < L->r[i - increment]) {
                                L->r[0] = L->r[i]; // 暂存在 L->r[0]

        for(j = i - increment; j > 0 && L->r[0] < L->r[j]; j -= increment) {
                                        L->r[j + increment] = L->r[j]; // 记录后移，查
找插入位置
                                }
                                L->r[j + increment] = L->r[0]; // 插入
                        }
                }
                //printf("     第%d 趟排序结果:  ", ++k);
                //Print(*L);
        } while(increment > 1);
}

/* 对顺序表 L 进行堆排序 */
void HeapSort(SqList *L) {
        int i;
        /*  把 L 中的 r 构建成一个大根堆 */
```

```
        for(i = L->length / 2; i > 0; i--) {
                HeapAdjust(L, i, L->length);
        }
        for(i = L->length; i > 1; i--) {
                swap(L, 1, i); // 将堆顶记录和当前未经排序子序列的最后一个记录交换
                HeapAdjust(L, 1, i - 1); // 将 L->r[1..i-1]重新调整为大根堆
        }
}

/* 对顺序表 L 作递归的归并排序 */
void ReMergeSort(SqList *L) {
        MSort(L->r, L->r, 1, L->length);
}

/* 对顺序表 L 作非递归的归并排序 */
void MergeSort(SqList *L) {
        KeyType* TR = (KeyType*)malloc(L->length * sizeof(KeyType)); // 申请额外空间
        int k = 1;
        while(k < L->length) {
                MergePass(L->r, TR, k, L->length);
                k = 2 * k; // 子序列长度加倍
                MergePass(TR, L->r, k, L->length);
                k = 2 * k; // 子序列长度加倍
        }
}

/* 对顺序表 L 作经典快速排序 */
void QuickSort(SqList *L) {
        QSort(L, 1, L->length);
}

/* 对顺序表 L 作优化快速排序 */
void OpQuickSort(SqList *L) {
        OpQSort(L, 1, L->length);
}
```

8.1.3. 测试

```
int main(int argc, char** argv) {
        int d[MAXSIZE + 1]; // 声明数组
        SqList l0, l1, l2, l3, l4, l5, l6, l7, l8, l9, l10; // 声明顺序表
        int i; // 声明临时变量

        srand((unsigned int)time(NULL)); //设置随机数种子

        *d = CreatArray(d); // 随机数组生成器
        for (i = 1; i <= MAXSIZE; i++) {
```

```
        l0.r[i] = d[i];
}
l0.length = MAXSIZE;
l1 = l2 = l3 = l4 = l5 = l6 = l7 = l8 = l9 = l10 = l0;
printf("排序前: ");
Print(l0);
printf("\n");

printf("交换排序:\n");
SwapSort(&l0);
Print(l0);

printf("冒泡排序:\n");
BubbleSort(&l1);
Print(l1);

printf("带标志冒泡排序:\n");
FlagBubbleSort(&l2);
Print(l2);

printf("选择排序:\n");
SelectSort(&l3);
Print(l3);

printf("直接插入排序:\n");
InsertSort(&l4);
Print(l4);

printf("希尔排序:\n");
ShellSort(&l5);
Print(l5);

printf("堆排序:\n");
HeapSort(&l6);
Print(l6);

printf("归并排序（递归）:\n");
ReMergeSort(&l7);
Print(l7);

printf("归并排序（非递归）:\n");
MergeSort(&l8);
Print(l8);

printf("快速排序:\n");
QuickSort(&l9);
Print(l9);
```

```
    printf("优化快速排序:\n");
    OpQuickSort(&l10);
    Print(l10);

    return 0;
}
```