数据结构基础

引言：

我们已经知道用计算机解决一个具体问题时,首先要从具体问题抽象出一个适当的数学模型，然后设计一个解此数学模型的算法，最后编出程序，进行测试、调整直至得到最终解答。如果一个问题可以用数学的方程来描述，如人口增长可以用微分方程来描述，只需要输入相应的数据，然后通过计算机求解该方程即可。这一般称为数值计算。然而，很多问题是无法用数学方程来描述的,如计算机和人的对弈问题。这类非数值计算问题需要用其他的方式来描述和求解，其核心是数据结构和算法。

本节对常用的数据结构作了简单讲述，实现了线性表、队列和栈等常用的数据结构。

教学目的：

- 掌握数据与数据结构的基本概念
- 掌握线性表的基本结构及其上的运算
- 掌握栈和队列的基本概念
- 了解树和图的基本概念

8.1 数据与数据结构

8.1.1 数据

什么叫**数据**？数据是描述客观事物的信息符号的集合,这些信息符号能被输入到计算机中存储起来，又能被程序处理、输出。事实上，数据这个概念本身是随着计算机的发展而不断扩展的概念。在计算机发展的初期由于计算机主要用于数值计算，数据指的就是整数、实数等数值；在计算机用于文字处理时，数据指的就是由英文字母和汉字组成的字符串；随着计算机硬件和软件技术的不断发展，扩大了计算机的应用领域，诸如表格、图形、图象、声音等也属于数据的范畴。目前非数值问题的处理占用 90%以上的计算机时间。

数据类型是程序设计中的概念，程序中的数据都属于某个特殊的数据类型，它是指具有相同特性的数据的集合。数据类型决定了数据的性质，如取值范围、操作运算等。常用的数据类型有整型、浮点型、字符型等。数据类型还决定了数据在内在中所占空间的大小，如字符型占一个字节，而长整型一般占 4 个字节等。

对于复杂一些的数据，仅用数据类型无法完整地描述，如表示教师得分要描述教师的姓名、各项得分，这时需要用到数据元素的概念。数据元素中可能用到多个数据类型（称为数据项），共同描述一个客体，如教师。数据元素有时也被称为记录或结点。在程序设计中，前面所说的数据类型又被称为基本数据类型,由基本数据类型组成的数据元素的定义被称为构造数据类型（结构和类都属此列）。

教师得分登记表

姓　名	教学得分	科研得分	其他得分	合计
张　力	35	34	11	80
王　五	36	35	12	83
…	…	…	…	。。。

教师得分登记表的数据元素是姓名、教学得分、科研得分、其他得分、合计，也就是说每个数据元素由姓名、教学得分、科研得分、其他得分、合计五个数据项组成。这五个数据项含义明确，若再细分就无明确独立的含义，属于基本数据类型（字符型和整型或浮点型）。

8.1.2 数据结构

我们说计算机的处理效率与数据的组织形式和存储结构密切相关。这类似于人们所用的“英语词典”、“科学技术辞海”和“新华字典”等工具书，它们都是按字母或拼音字母的顺序组织排列“词条”。这样人们查阅工具书的速度较快。假如“词条”不是按字母顺序组织排列，而是任意顺序组织排列，那么查词速度一定很低。因此，很有必要研究数据的组织形式和存储结构。另外在当今网络世界中传递数据更加依赖于数据的组织形式和存储结构。

什么是**数据结构**？数据结构在计算机科学界至今没有标准的定义。个人根据各自的理解的不同而有不同的表述方法。

Sartaj Sahni 在他的《数据结构、算法与应用》一书中称：“数据结构是数据对象，以及存在于该对象的实例和组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。”他将数据对象（data object）定义为“一个数据对象是实例或值的集合”。

Clifford A.Shaffer 在《数据结构与算法分析》一书中的定义是：“数据结构是 ADT（抽象数据类型 Abstract Data Type） 的物理实现。”

Lobert L.Kruse 在《数据结构与程序设计》一书中，将一个数据结构的设计过程分成抽象层、数据结构层和实现层。其中，抽象层是指抽象数据类型层，它讨论数据的逻辑结构及其运算，数据结构层和实现层讨论一个数据结构的表示和在计算机内的存储细节以及运算的实现。

由此可见，在任何问题中，构成数据的数据元素并不是孤立存在的，它们之间存在着一定的关系以表达不同的事物及事物之间的联系。所以，简单地说数据结构就是研究数据及数据元素之间关系的一门学科，它包括三个方面的内容：

- 数据的逻辑结构；
- 数据的存贮结构；
- 数据的运算（即数据的处理操作）。

一般认为，一个数据结构是由数据元素依据某种逻辑联系组织起来的。对数据元素间逻辑关系的描述称为数据的逻辑结构；数据必须在计算机内存储，数据的存储结构是数据结构的实现形式，是其在计算机内的表示；此外讨论一个数据结构必须同时讨论在该类数据上执

行的运算才有意义。

1．数据的逻辑结构

数据的逻辑结构就是数据元素之间的逻辑关系。这里，我们对数据所描述的客观事物本身的属性意义不感兴趣，只关心它们的结构及关系。将那些在结构形式上相同的数据抽象成某一数据结构。比如线性表、树和图等等。

根据数据元素之间关系的不同特性，数据结构又可分为以下四大类（图 8-1）：

1）集合 数据元素之间的关系只有“是否属于同一个集合”

2）线性结构 数据元素之间存在线性关系，即最多只有一个前导和后继元素；

3）树形结构 数据元素之间程层次关系，即最多有一个前导和多个后继元素；

4）图状结构 数据元素之间的关系为多对多的关系。

其中树和图又被统称为非线性数据结构。

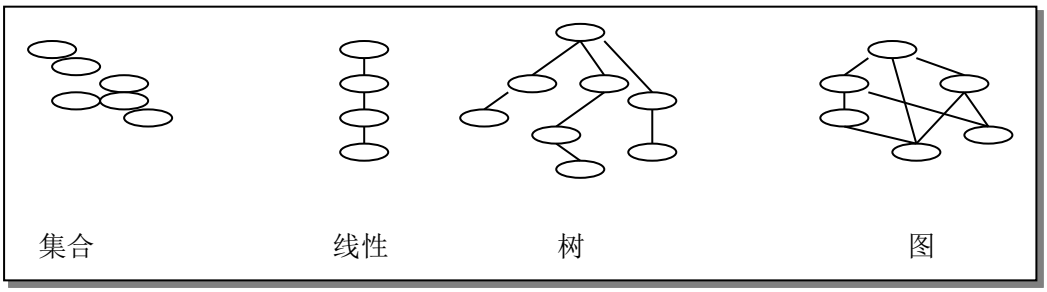


图 8-1 四种逻辑结构示意图

2．数据的存贮结构

数据的逻辑结构是从逻辑上来描述数据元素之间的关系的，是独立于计算机的。然而讨论数据结构的目的是为了在计算机中实现对它的处理。因此还需要研究数据元素和数据元素之间的关系如何在计算机中表示，这就是数据的存贮结构。又称数据的映象。

计算机的存贮器是由很多个存贮单位组成的，每个存贮单元有唯一的地址。数据的存贮结构要讨论的就是数据结构在计算机存贮器上的存贮映象方法。根据数据结构的形式定义，数据结构在存贮器上的映象，不仅包括数据元素集合如何存贮映象，而且还包括数据元素之间的关系如何存贮映象。

一般来说，数据在存贮器中的存贮有四种基本的映象方法。

1）顺序存贮结构

插入删除，开销大

顺序存贮结构是把数据元素按某种顺序放在一块连续的存贮单元中，其特点是借助数据元素在存贮器中的相对位置来表示数据元素之间的关系。顺序存贮的问题是，如果元素集合很大，则可能找不到一块很大的连续的空间来存放。

2）链式存贮结构 只保留相对位置

有时往往存在这样一些情况：存贮器中没有足够大的连续可用空间，只有不相邻的零碎小块存贮单元；另一种情况是在事前申请一段连续空间时，因无法预计所需存贮空间的大小，需要临时增加空间。所有这些情况，要得到一块合适的连续存贮单元并非易事，即这种情况

下顺序存贮结构无法实现。

链式存贮结构的特点就是将存放每个数据元素的结点分为两部分：一部分存放数据元素（称为数据域）；另一部分存放指示存贮地址的指针（称为指针域），借助指针表示数据元素之间的关系。结点的结构如下：

数据域	指针域

链式存贮结构可用一组任意的存贮单元来存贮数据元素，这组存贮单元可以是连续的，也可以是不连续的。链式存贮因为有指针域，增加了额外的存贮开销，并且实现上也较为麻烦，但大大增加了数据结构的灵活性。

3）索引存贮结构

在线性表中，数据元素可以排成一个序列：R_1，R_2，R_3，…，R_n，每个数据元素 R_i 在序列里都有对应的位置码 i，这就是元素的索引号。索引存贮结构就是通过数据元素的索引号 i 来确定数据元素 R_i 的存贮地址。一般索引存贮结构有两种实现方法：a. 建立附加的索引表，索引表里第 i 项的值就是第 i 个元素的存贮地址；b. 当每个元素所占单元数都相等时，可用位置码 i 的线性函数值来确定元素对应的存贮地址，即

$$Loc（R_i）=（i-1）*L+d_0。$$

4）散列存贮结构

这种存贮方法就是在数据元素与其在存贮器上的存贮位置之间建立一个映象关系 F。根据这个映象关系 F，已知某数据元素就可以得到它的存贮地址。即 D=F（E），这里 E 是要存放的数据元素，D 是该数据元素的存贮位置。可见，这种存贮结构的关键是设计这个函数 F，但函数 F 不可能解决数据存贮中所有问题，还应有一套意外事件的处理方法，它们共同实现数据的散列存贮结构。哈希表是一种常见的散列存贮结构。

3．数据的运算

数据的运算是定义在数据逻辑结构上的操作，如插入、删除、查找、排序、遍历等。每种数据结构都有一个运算的集合。

8.2 线性表

线性表是最基本、最简单、也是最常用的一种数据结构。线性表中数据元素之间的关系是一对一的关系，即除了第一个和最后一个数据元素之外，其它数据元素都是首尾相接的。线性表的逻辑结构简单，便于实现和操作，在实际应用中是广泛采用的一种数据结构。

8.2.1 线性表的逻辑结构及运算

线性表是一个线性结构，它是一个含有 n≥0 个结点的有限序列，对于其中的结点，有且仅有一个开始结点（第一个结点）没有前驱但有一个后继结点，有且仅有一个终端结点（最后一个结点）没有后继但有一个前驱结点，其它的结点都有且仅有一个前驱和一个后继结点。

一般地，一个线性表可以表示成一个线性序列：k1,k2,…,kn，其中 k1 是开始结点，kn 是终端结点。线性表具有以下一些基本性质：

- 数据元素的个数 n 定义为表的长度，当 n=0 时，称为空表。空表中无数据元素；
- 若表非空，则必存在唯一的一个开始结点；
- 必存在唯一的一个终端结点；
- 除最后一个元素之外，其余结点均有唯一的后继；
- 除第一个元素之外，其余结点均有唯一的前驱；
- 数据元素 ki（1≤i≤n）在不同情况下的具体含义不同，它可以是一个数，或者是一个符号，或者是更复杂的信息。虽然不同数据表的数据元素可以是各种各样的，但对于同一线性表的各数据元素必定具有相同的数据类型和长度。

【例 8-1】线性表的例子：

- 某班学生的数学成绩（78，92，66，84，45，72，92）是一个线性表，每个数据元素是一个正整数，表长为 7。
- 一星期的七天的英文缩写词（SUN，MON，TUE，WED，THU，FRI，SAT）是一线性表，表中数据元素是一字符串，表长为 7。
- 某企业职工基本工资情况（(张三，助工，3，543)，(李四，高工，21，986)，(王五，工程师，9，731)）亦是一线性表，表中数据元素是由姓名、职称、工龄，基本工资四个数据项组成的一个记录（对象），表长为 3。

线性表可以进行的常用基本操作有以下几种：

（1）置空表：将线性表 L 的表长置为 0。

（2）return：求出线性表 L 中数据元素的个数。

（3）取表中元素：仅当 1≤i≤Length（L）时，取得线性表 L 中的第 i 个元素 ki（或 ki 的存储位置），否则无意义。

（4）取元素 ki 的直接前趋：当 2≤i≤Length（L）时，返回 ki 的直接前趋 ki-1。

（5）取元素 ki 的直接后继：当 1≤i≤Length（L）-1 时，返回 ki 的直接后继 ki+1。

（6）定位：返回元素 x 在线性表 L 中的位置。若在 L 中有多个 x，则只返回第一个 x 的位置，若在 L 中不存在 x，则返回 0。

（7）插入：在线性表 L 的第 i 个位置上插入元素 x，运算结果使得线性表的长度增加 1。

（8）删除：删除线性表 L 的第 i 个位置上的元素 ki，此运算的前提应是 Length（L）≠0，运算结果使得线性表的长度减 1。

对线性表还有一些更为复杂的操作，如：将两个线性表合并成一个线性表；将一个线性表分解为 n 个线性表；对线性表中的元素按值的大小重新排列等。这些运算都可以通过上述八种基本运算的组合派生来实现。

存储(1)顺序 2.链表

8.2.2 顺序线性表（例数组）

要使线性表成为计算机可以处理的对象，就必须把线性表的数据元素及数据元素之间的逻辑关系都存储到计算机的存储器中。线性表常用顺序方式和链表方式来存储。

线性表的顺序存储结构就是将线性表的每个数据元素按其逻辑次序依次存放在一组地址连续的存储单元里。由于逻辑上相邻的元素存放在内存的相邻单元中，所以线性表的逻辑关系蕴含在存储单元的物理位置相邻的关系中。也就是说，在顺序存储结构中，线性表的逻辑关系的存储是隐含的。

设线性表中每个元素占用 C 个存储单元，用 Loc（ki）表示元素 ki 的存储位置，则顺序存储结构的存储示意图如图 8-2 所示。

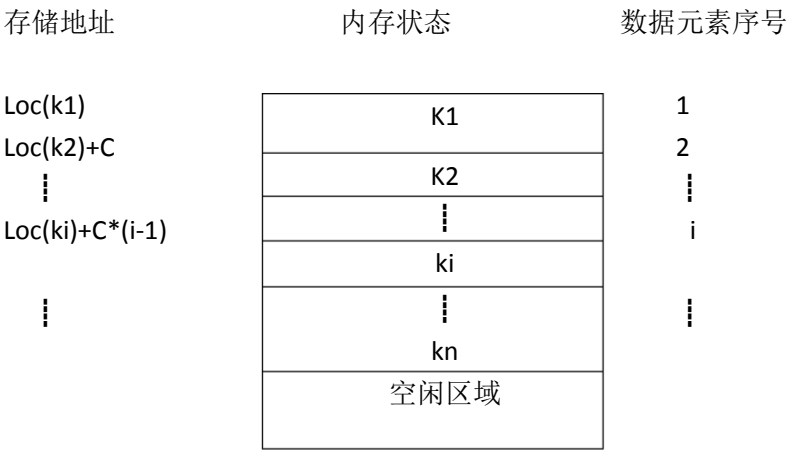


图 8-2 线性表的顺序存储结构示意图

从图中可以看出，若已知线性表的第一个元素的存储位置是 Loc（k1），则第 i 个元素的存储位置为：

$$Loc(ki) = Loc(k1) + C*(i-1) \qquad 1 \leqslant i \leqslant n$$

可见，线性中每个元素的存储地址是该元素在表中序号的线性函数。只要知道某元素在线性表中的序号就可以确定其在内存中的存储位置。所以说，线性表的顺序存储结构是一种随机存取结构。

在 C 语言中，数组是在内存中连续分配的。所以数组天生就是一种线性结构。用数组来实现线性表，可以预先定义一个较大的数组，用来存放线性表中的元素。元素从数组的 0 位置存起，数组最后的一些位置是空闲的。

【例 8-2】一个整数线性表的实现。用整型数组存储元素，实现线性表的基本操作。

分析：使用数组 list 来存储元素。list 的大小设为 MAX=1000。表长用 length 来表示，线性表中的每一个元素都是整数。这里，使用一个自定义的数据类型来描述线性表。该数据类型 ListType 包含一个数组用于存放数据。一个整数表示表长以及一个表示最大容量的整数。

程序：

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

/*本例设计一个元素类型为整数的线性表
 *通用起见，此处将整数重定义为 DataType
 */
```

```
typedef int DataType;

//定义线性表的结构
typedef struct List
{
    DataType* list;      //指向线性表的指针
    int length;     //表长
    int maxLength;       //表容量
}ListType;

//声明线性表具有的方法
ListType* CreateList(int length);                          //创建一个长度为 length 的线性表
void DestroyList(ListType* pList);                          //销毁线性表
void ClearList(ListType* pList);                            //置空线性表
int IsEmptyList(ListType* pList);                           //检测线性表是否为空
int GetListLength(ListType* pList);                         //获取线性表长度
int GetListElement(ListType* pList, int n, DataType* data);     //获取线性表中第 n 个元素
int FindElement(ListType* pList, int pos, DataType data); //从 pos 起查找 data 第 1 次出现的位置
int GetPriorElement(ListType* pList, int n, DataType* data);    //获取第 n 个元素的前驱
int GetNextElement(ListType* pList, int n, DataType* data);    //获取第 n 个元素的后继
int InsertToList(ListType* pList, int pos, DataType data);       //将 data 插入到 pos 处
int DeleteFromList(ListType* pList, int pos);                   //删除线性表上位置为 pos 的元素
void PrintList(ListType* pList);                               //输出线性表

/* 主函数，创建一个线性表，并测试*/
int main()
{
    const int MAXLENGTH = 1000;                 //假设最大容量为 1000

    //创建线性表
    ListType* sqList = CreateList(MAXLENGTH);

    //以下是对线形表的测试
    ClearList(sqList);                          //置表为空

    //插入 10 个元素并显示
    for (int i = 0; i < 10; ++i)
        InsertToList(sqList, i, i + 1);

    //输出线性表
    PrintList(sqList);

    //在位置 5 插入 99 并显示
```

```
    InsertToList(sqList, 5, 99);
    printf("插入 99 后的线性表\n");
    PrintList(sqList);

    //删除第 8 个元素
    DeleteFromList(sqList, 8);
    printf("删除第 8 个元素后的线性表\n");
    PrintList(sqList);

    //显示第 3 个元素的前驱
    DataType data;
    if (GetPriorElement(sqList, 3, &data) > -1);
        printf("第 3 个元素的前驱是%d\n", data);
    return 0;
}

//线性表方法实现
/**
 *@brief 创建一个新的线性表
 *@param length 线性表的最大容量
 *@return 成功返回直线该表的指针，否则返回 NULL
*/
ListType* CreateList(int length)
{
    ListType* sqList=(ListType*)malloc(sizeof(ListType));
    if (sqList != NULL)
    {
        //为线性表分配内存
        sqList->list = (DataType*)malloc(sizeof(DataType)*length);

        //如果分配失败，返回 NULL
        if (sqList->list == NULL)
            return NULL;

        //置为空表
        sqList->length = 0;

        //最大长度
        sqList->maxLength = length;
    }
    return sqList;
}
/**
```

```
 *@brief  销毁线性表
 *@param pList  指向需要销毁的线性表的指针
*/
void DestroyList(ListType* pList)
{
    free(pList->list);
}
/**
*@brief  置空线性表
*@param pList  指向需要置空线性表的指针
*/
void ClearList(ListType* pList)
{
    pList->length = 0;
}
/**
*@brief  检测线性表是否为空
*@param pList  指向线性表的指针
*@return  如果线性表为空，返回 1；否则返回 0
*/
int IsEmptyList(ListType* pList)
{
    return pList->length == 0 ? 1 : 0;
}
/**
*@brief  获取线性表长度
*@param pList  指向线性表的指针
*@return  线性表的长度
*/
int GetListLength(ListType* pList)
{
    return pList->length;
}
/**
 *@brief  获取线性表中第 n 个元素
 *@param pList  指向线性表的指针
 *@param n  要获取元素在线性表中的位置
 *@param data  获取成功，取得元素存放与 data 中
 *@return  获取成功返回 1，失败则返回 0
 */
int GetListElement(ListType* pList, int n, DataType* data)
```

```
{
    if (n<0 || n>pList->length - 1)
        return 0;
    *data = pList->list[n];
    return 1;
}
/**
 *@brief 从 pos 起查找 data 第一次出现的位置
 *@param pList 指向线性表的指针
 *@param pos 查找的起始位置
 *@param data 要查找的元素
 *@return 找到则返回该位置，未找到，返回-1
 */
int FindElement(ListType* pList, int pos, DataType data)
{
    for (int n = pos; n < pList->length; ++n)
    {
        if (data == pList->list[n])
            return n;
    }
    return -1;
}
/**
*@brief 获取第 n 个元素的前驱
*@param pList 指向线性表的指针
*@param n n 的前驱
*@param data 获取成功，取得元素存放与 data 中
*@return 找到则返回前驱的位置（n-1），未找到，返回-1
*/
int GetPriorElement(ListType* pList, int n , DataType* data)
{
    if (n < 1 || n>pList->length-1)
        return -1;
    *data = pList->list[n - 1];
    return n-1;
}
/**
*@brief 获取第 n 个元素的后继
*@param pList 指向线性表的指针
*@param n n 的后继
*@param data 获取成功，取得元素存放与 data 中
```

```
*@return 找到则返回后继的位置（n+1），未找到，返回-1
*/
int GetNextElement(ListType* pList, int n,DataType* data)
{
    if (n<0 || n>pList->length - 2)
        return -1;
    *data = pList->list[n + 1];
    return n+1;
}
/**
*@brief 将 data 插入到线性表的 pos 位置处
*@param pList 指向线性表的指针
*@param pos 插入的位置
*@param data 要插入的元素存放 data 中
*@return 成功，返回新的表长（原表长+1），失败，返回-1
*/
int InsertToList(ListType* pList, int pos, DataType data)
{

    //如果插入的位置不正确或者线性表已满，则插入失败
    if (pos<0 || pos>pList->length || pList->length == pList->maxLength)
        return -1;

    //从 pos 起，所有的元素向后移动 1 位
    for (int n = pList->length; n > pos; --n)
    {
        pList->list[n] = pList->list[n - 1];
    }

    //插入新的元素
    pList->list[pos] = data;

    //表长增加 1
    return ++pList->length;
}
/**
*@brief 将 pos 位置处的元素删除
*@param pList 指向线性表的指针
*@param pos 删除元素的位置
*@return 成功，返回新的表长（原表长-1），失败，返回-1
*/
int DeleteFromList(ListType* pList, int pos)
{
```

```
    if (pos<0 || pos>pList->length)
        return -1;
    //将 pos 后的元素向前移动一位
    for (int n = pos; n < pList->length - 1; ++n)
        pList->list[n] = pList->list[n + 1];
    return --pList->length;
}
/**
*@brief 输出线性表
*@param pList 指向线性表的指针
*/
void PrintList(ListType* pList)
{
    for (int n = 0; n < pList->length; ++n)
        printf("第%d 项: %d\n", n, pList->list[n]);
}
```

需要注意的是由于 C 中的数组下标是从 0 开始的。方便起见，本程序实现的线性表默认的第一个元素下标是 0。因此在第 5 个位置上插入，实际是在线性表的第 6 个位置上插入。你也可以修改程序，使其和前面描述的线性表一致。程序运行的结果：

```
    第 0 项: 1
    第 1 项: 2
    第 2 项: 3
    第 3 项: 4
    第 4 项: 5
    第 5 项: 6
    第 6 项: 7
    第 7 项: 8
    第 8 项: 9
    第 9 项: 10

插入 99 后的线性表
    第 0 项: 1
    第 1 项: 2
    第 2 项: 3
    第 3 项: 4
    第 4 项: 5
    第 5 项: 99
    第 6 项: 6
    第 7 项: 7
    第 8 项: 8
    第 9 项: 9
```

第 10 项: 10

删除第 8 个元素后的线性表

第 0 项: 1

第 1 项: 2

第 2 项: 3

第 3 项: 4

第 4 项: 5

第 5 项: 99

第 6 项: 6

第 7 项: 7

第 8 项: 9

第 9 项: 10

第 3 个元素的前驱是 3

8.2.3 链表

线性表的顺序存储结构是把整个线性表存放在一片连续的存储区域，其逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，每个元素的存储位置可用一个简单、直观的公式来表示。然而，如果需要对某一线性表中的元素频繁进行插入和删除操作时，为了保持元素在存储区域的连续性，在插入元素时必须移动大量元素给新插入的元素“腾位置”，而在删除时，又必须移动大量后继元素“补缺”，因而在操作执行时要花大量时间去移动数据元素。此外顺序表类在创建时需要开辟较大的连续空间，而表中元素进进出出不可能一下子占满这块连续空间，所以存储空间的使用效率不高。

能否设计一种新的存储结构来弥补顺序表类的不足，尤其在元素插入、删除时无须改变已存储元素的位置？这就是我们将讨论的另一种存储结构——非顺序存储结构，又称链式存储结构。

链式结构用一组任意的存储区域存储线性表，此存储区域可以是连续的，也可以是分散的。这样，逻辑上相邻的元素在物理位置上就不一定是相邻的，为了能正确反映元素的逻辑次序，就必须在存储每个元素 a_i 的同时，存储其直接后继（或直接前驱）的存储位置。因此在链式结构中每个元素都由两部分组成：存储数据元素的数据域（data）；存储直接后继元素存储位置的指针域（next）。其存储结构示意如下：

data	next

在下面讨论中将这样存储的每个数据元素称为结点。每个数据元素存储结构的定义：

```
typedef struct NODE
{
    datatype    data;       //数据域
    Node* next;             //指针域
}Node;
```

由于线性表每个元素都有唯一的后继（除了最尾元素），所以开辟指针域记录后继元素的地址。最尾元素的指针域为空，即为 NULL。数据元素本身存储类型同顺序表类一样定义成结构类型。线性表的非顺序存储结构如图 8-3 所示：

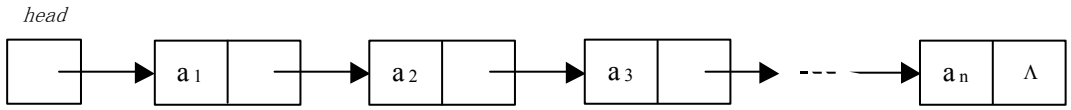


图 8-1 线性表的非顺序存储结构示意图

非顺序存储结构中每个结点的存储既不是连续的，也不是顺序的，而是散落在存储器的各个区域。通过指针将线性表中每个结点有机地连接起来，指针就好像自行车链条一样。因此图 8-3 所示的线性表存储结构被称为单向链表，简称单链表。

单链表类的特点是：（1）线性表中实际有多少元素就存储多少个结点；（2）元素存放可以不连续，其物理存放次序与逻辑次序不一定一致，换句话说，a_{i-1} 可能存放在存储器的下半区，而 a_i 可能存放在存储器的上半区；（3）线性表中元素的逻辑次序通过每个结点指针有机地连接来体现；（4）插入和删除不需要大量移动表中元素。

要在链表中插入一个新结点怎样实现呢？设有线性表（a_1，a_2，…，a_{i-1}，a_i，…，a_n），采用单链表存储结构，头指针为 head，要求在数据元素 a_i 的结点之前插入一个数据元素为 data 的新结点。插入前单链表的逻辑状态如图 8-4 所示。

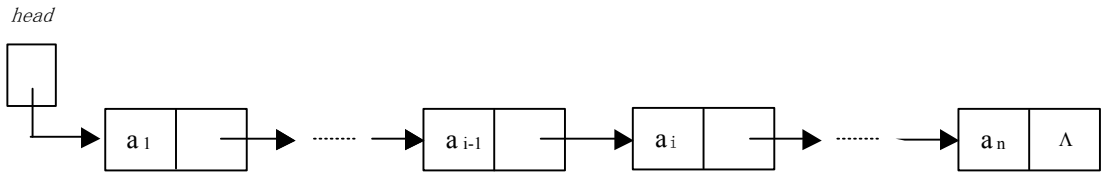


图 8-2 插入前单链表的存储结构

设新插入的结点指针是 s，插入后单链表的逻辑状态如图 8-5 所示。

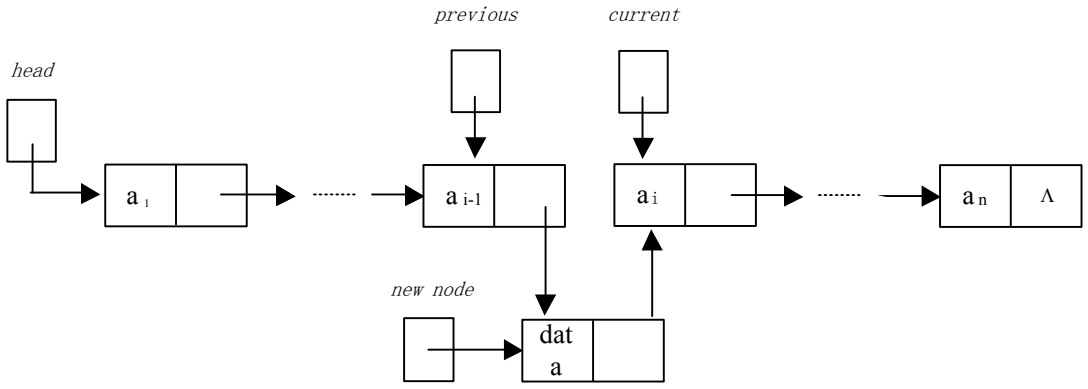


图 8-3 插入后单链表的存储结构

若已知 previous 指向为 a_i 的前趋 a_{i-1} 结点，newnode 指向新结点。只要执行以下两步操作即可完成插入新结点：

① 令新结点指针域指向 a_i 结点（newnode->next=previous->next）；

② 令 a i-1 结点的指针域指向新结点（previous->next=newnode）。

这就使得单链表成为如图 8-5 上所示的插入后的逻辑状态。

由此可见，插入操作执行之前，首先就是要得到单链表中插入位置的前一个结点的指针（存储位置）。插入算法的主要步骤和程序描述如下：

第 1 步： 判定插入位置是否正确，若正确继续下一步，否则结束算法；

第 2 步： 申请一个新结点，判定内存有无空间（即表满否）；

第 3 步： 元素放入数据域，NULL 放入指针域；

第 4 步： 判定是否为插入在表头，若是表头，则修改 head 和新结点指针；

第 5 步： 寻找插入位置，指向 a_{i-1} 结点；

第 6 步： 修改 a_{i-1} 结点的指针域和新结点的指针域。

删除操作和插入操作一样，首先要搜索单链表以找到指定删除结点的前趋结点(假设为 previous)，然后只要将待删除结点的指针域内容赋予 previous 所指向的结点的指针域就可以了。

已知单链表的头指针为 head，删除前单链表的逻辑状态如图 8-6 所示。

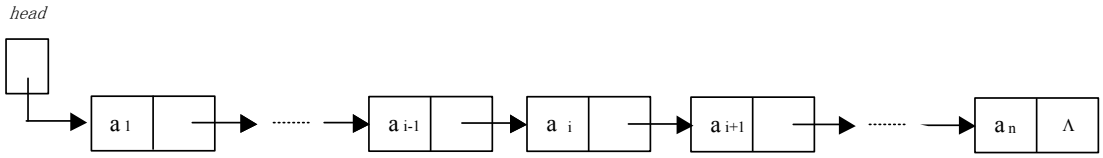


图 8-4 删除前单链表的存储结构

删除结点之后，单链表的逻辑状态如图 8-7 所示。

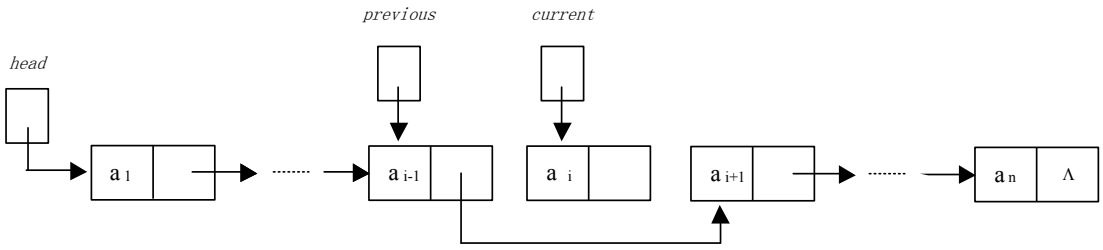


图 8-5 删除后单链表的存储结构

若已知 previous 指向为 a_i 的前趋 a_{i-1} 结点，只要执行以下两步操作即可完成删除结点：

① 令 a_{i-1} 结点指针域指向 a_{i+1} 结点（previous->next=current->next）；

② 释放 a i 结点所占存储空间（delete current）。

这就使得单链表成为如图 8-7 所示的删除后的逻辑状态。

删除算法的主要步骤为：

第 1 步：　判定是否为空表，若为空表，删除错误，结束算法，否则继续下一步；

第 2 步：　判定删除位置是否正确，若正确继续下一步，否则结束算法；

第 3 步：　寻找删除位置，指向 a_{i-1} 结点；

第 4 步：　若删除表头元素，修改 head 指针；

第 5 步：　若删除非表头元素，修改 a_{i-1} 结点指针域；

第 6 步：　释放被删结点的存储空间。

现在可以编写测试程序，测试插入函数和删除函数的正确性。参照前面测试顺序表类的思路，分别对链表头插入和删除、链表尾插入和删除、链表的中间位置插入和删除、错误位置插入和删除进行验证，对不同类型的数据元素（如浮点型、字符串类）的链表也进行验证。

【例 8-3】一个整数单链表的实现。

程序：

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

//假设使用的是整数链表
typedef int DataType;
//单链表的节点定义
typedef struct NODE Node;
typedef struct NODE
{
    DataType    data;      //数据域
    Node* next;           //指针域
}Node;
//头指针
typedef Node* Head;
//链表的方法声明
int GetLinkListLength(Head head);             //求表长
void DestroyLinkList(Head head);        //销毁链表
int GetElement(Head head, int n, DataType* data);     //获取线性表中第 n 个元素
int FindElement(Head head, DataType data);          //查找 data  第一次出现的位置
int GetPriorElement(Head head, int n, DataType* data);     //获取第 i 个元素的前驱
int GetNextElement(Head head, int n, DataType* data);        //获取第 i 个元素的后继
int InsertToList(Head* head, int pos, DataType data);   //将  data 插入到  pos  处
int DeleteFromList(Head head, int pos);                    //删除线性表上位置为 pos  的元素
void PrintList(Head head);                 //输出线性表
int InsertRear(Head* head, DataType data);                 //从表尾插入元素
int InsertHead(Head* head, DataType data);                  //从表头插入元素

//主函数
```

```
int main()
{
	//创建一个空的线性表
	Head head = NULL;
	//以下是对线形表的测试
	//插入 5 个元素并显示
	for (int i = 0; i < 5; ++i)
		InsertToList(&head, i, i + 1);
	//输出线性表
	PrintList(head);
	//在位置 2 插入 99 并显示
	printf("插入 99 后的表长：%d\n",InsertToList(&head, 2, 99));
	printf("插入 99 后的线性表\n");
	PrintList(head);
	//删除第 4 个元素
	printf("删除第 4 个元素后的表长：%d\n",DeleteFromList(head, 4));
	printf("删除第 4 个元素后的线性表\n");
	PrintList(head);
	//显示第 3 个元素的前驱
	DataType data;
	if (GetPriorElement(head, 3, &data) > -1);
	printf("第 3 个元素的前驱是%d\n", data);
	DestroyLinkList(head);
	return 0;
}

//链表的方法实现
/**
 *@brief     求表长
 *@param head  链表的头指针
 *@return  链表的长度
*/
int GetLinkListLength(Head head)
{
	if (head == NULL)
		return 0;
	int i = 1;
	Node* pNode = head;
	while (pNode->next)        //!=NULL
	{
		i++;
```

```
            pNode = pNode->next;     //下一节点
        }
        return i;
}
/**
*@brief     销毁链表
*@param head  链表的头指针
*/
void DestroyLinkList(Head head)
{
        //从头开始，依次释放每一个节点
        Node* pNode;
        while (head)
        {
            pNode = head;
            head = head->next;     //指向下一个节点
            free(pNode);   //释放当前节点
        }
}
/**
*@brief     获取线性表中第 n 个元素，第一个元素的位置为 0
*@param head  链表的头指针
*@param n  要获取的元素位置
*@param data  获取的数据存放与此
*@return    获取成功返回 1，失败则返回 0
*/
int GetElement(Head head, int n, DataType* data)
{
        if (n<0 || n>GetLinkListLength(head) - 1)
            return 0;
        for (int i = 0; i < n; ++i)
            head = head->next;            //移动到位置 n
        *data = head->data;
        return 1;
}
/**
*@brie  查找 data 第一次出现的位置
*@param head  指向线性表的头指针
*@param data  要查找的元素
*@return  找到则返回该位置，未找到，返回-1
*/
```

```
int FindElement(Head head, DataType data)
{
	int i = 0;
	while (head)
	{
		if (head->data == data)	//找到
			return i;
		head = head->next;	//下一个节点
		i++;
	}
	return -1;
}
/**
*@brief 获取第 n 个元素的前驱
*@param head 指向线性表的头指针
*@param n n 的前驱
*@param data 获取成功，取得元素存放与 data 中
*@return 找到则返回前驱的位置（n-1），未找到，返回-1
*/
int GetPriorElement(Head head, int n, DataType* data)
{
	if (n<1 || n>GetLinkListLength(head) - 1)
		return -1;
	for (int i = 0; i < n - 1; ++i)
		head = head->next;		//移动到 n-1
	*data = head->data;
	return n - 1;
}
/**
*@brief 获取第 n 个元素的后继
*@param head 指向线性表的头指针
*@param n n 的后继
*@param data 获取成功，取得元素存放与 data 中
*@return 找到则返回前驱的位置（n+1），未找到，返回-1
*/
int GetNextElement(Head head, int n, DataType* data)
{
	if (n<0 || n>GetLinkListLength(head) - 2)
		return -1;
	for (int i = 0; i < n+1 ; ++i)
		head = head->next;		//移动到 n
```

```
    *data = head->data;
    return n + 1;
}
/**
*@brief   将 data 插入到 pos 处
*@param head 指向线性表的头指针
*@param pos 插入的位置
*@param data 要插入的数据
*@return 插入成功返回新的表长，否则返回-1
*/
int InsertToList(Head* head, int pos, DataType data)
{
    Node* pNode = *head;
    int length = GetLinkListLength(pNode);   //得到表长
    if (pos<0 || pos> length)
        return -1;          //插入的位置不对
    if (pos == 0)       //在表头插入
        return InsertHead(head, data);
    if (pos == length - 1)     //在表尾插入
        return InsertRear(head, data);
    //定位到 pos 前 1 位置
    for (int i = 0; i < pos-1; ++i)
        pNode = pNode->next;
    //生成一个新的节点
    Node* pNewNode = (Node*)malloc(sizeof(Node));
    if (pNewNode==NULL)
        return -1;       //分配内存失败
    pNewNode->data = data;        //存入要插入的数据
    //插入链表
    pNewNode->next = pNode->next;
    pNode->next = pNewNode;
    //返回新的链表长度
    return ++length;
}
/**
*@brief   删除线性表上位置为 pos 的元素
*@param head 指向线性表的头指针
*@param pos 删除的位置
*@return 插入成功返回新的表长，否则返回-1
*/
int DeleteFromList(Head head, int pos)
```

```
{
    Node* pNode = head;
    int length = GetLinkListLength(head) ;   //得到表长
    if (pos<0 || pos> length - 1)
        return -1;          //删除的位置不对
    //定位到 pos 位置
    for (int i = 0; i < pos-1; ++i)
        pNode = pNode->next;
    Node* pDeleteNode = pNode->next;
    pNode->next = pNode->next->next;
    free(pDeleteNode);
    return --length;
}
/**
*@brief   输出线性表
*@param head  指向线性表的头指针
*/
void PrintList(Head head)
{
    int n = 0;
    while (head)
    {
        printf("第%d 项元素为%d\n", n, head->data);
        head = head->next;
        ++n;
    }
}
/**
*@brief   从表尾插入元素
*@param head  指向线性表的头指针
*@param data  插入的数据
*@return  插入成功返回新的表长，否则返回-1
*/
int InsertRear(Head* head, DataType data)
{
    //准备新数据
    Node* pNewNode = (Node*)malloc(sizeof(Node));
    if (pNewNode == NULL)   //内存分配失败
        return -1;
    pNewNode->data = data;
    if (*head == NULL)     //如果是空表
```

```
    {
        *head = pNewNode;
        pNewNode->next = NULL;
        return 1;   //表长为 1
    }
    //找到表尾
    Node* pNode = *head;
    while (pNode->next)
        pNode = pNode->next;
    //插入到表尾
    pNode->next = pNewNode;
    pNewNode->next = NULL;
    //返回新的表长
    return GetLinkListLength(*head);
}
/**
*@brief   从表头插入元素
*@param head  指向线性表的头指针
*@param data  插入的数据
*@return  插入成功返回新的表长，否则返回-1
*/
int InsertHead(Head* head, DataType data)
{
    //准备新数据
    Node* pNewNode = (Node*)malloc(sizeof(Node));
    if (pNewNode == NULL)   //内存分配失败
        return -1;
    pNewNode->data = data;
    //插入
    if (*head == NULL)      //如果是空表
        pNewNode->next = NULL;
    else
        pNewNode->next = (*head)->next;
    *head = pNewNode;
    //返回新的表长
    return GetLinkListLength(*head);
}
```

程序的运行结果：

```
    第 0 项元素为 1
    第 1 项元素为 2
    第 2 项元素为 3
```

第 3 项元素为 4
第 4 项元素为 5
插入 99 后的表长：6
插入 99 后的线性表
第 0 项元素为 1
第 1 项元素为 2
第 2 项元素为 99
第 3 项元素为 3
第 4 项元素为 4
第 5 项元素为 5
删除第 4 个元素后的表长：5
删除第 4 个元素后的线性表
第 0 项元素为 1
第 1 项元素为 2
第 2 项元素为 99
第 3 项元素为 3
第 4 项元素为 5
第 3 个元素的前驱是 99

8.3 栈和队列

栈和队列也是线性结构，线性表、栈和队列这三种数据结构的数据元素以及数据元素间的逻辑关系完全相同，差别是线性表的操作不受限制，而栈和队列的操作受到限制。栈的操作只能在表的一端进行，队列的插入操作在表的一端进行而其它操作在表的另一端进行，所以，把栈和队列称为操作受限的线性表。

8.3.1 栈

栈是只能在某一端插入和删除的特殊线性表。它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶(top)，另一端为栈底(bottom)；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（PUSH），删除则称为出栈（POP）。栈也称为先进后出表。

图 8-8 是一个栈的示意图：

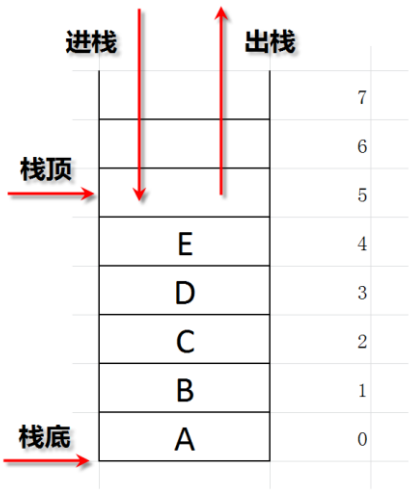


图 8-8 栈的示意图

栈顶始终指向栈顶最后一个元素之后的空位置。在图 8-8 中，栈里面共有 5 个元素，入栈的次序依次是 A B C D E。栈底始终等于 0，而栈顶等于 5。图 8-9 则描述了最后 2 个元素出栈，F 进栈的情形。

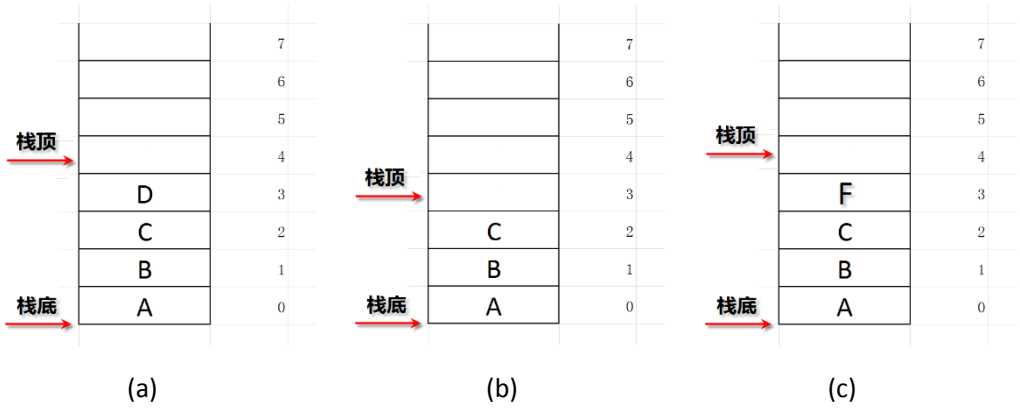


图 8-9 (a)最后一个元素 E 出栈，栈顶=4；(b)D 出栈，栈顶=3；(c)元素 F 进栈，栈顶=4

当栈中没有元素的时候，称为空栈，空栈的条件是栈顶=栈底。栈的大小一般是预先定义好的，当栈顶=栈的大小时，称为栈满。很显然的，当栈为空的时候，不能进行出栈操作，而当栈满的时候不能进行入栈操作。一般的，对栈有如下几个操作：

- 求栈的长度：GetLength，返回栈中数据元素的个数。
- 判断栈是否为空：IsEmpty，如果栈为空返回 true，否则返回 false。
- 清空栈：Clear，使栈为空。
- 入栈操作：Push 将新的数据元素添加到栈顶，栈发生变化。
- 出栈操作：Pop 将栈顶元素从栈中取出，栈发生变化。
- 取栈顶元素：GetTop 返回栈顶元素的值，栈不发生变化。

同样的，栈在计算机中的存储结构，也有顺序存储和链式存储结构。显然顺序结构自然可以用一个数组来实现。

【例 8-4】用数组实现栈。

分析：用一个指定大小的数组来存储栈的内容。在此，假设栈里存储的是字符串。变量 top 保存栈顶的数组下标，变量 bottom 为栈底，始终为 0。

程序：

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

//定义一个字符栈
typedef char DataType;
//定义栈
typedef struct STACK
{
    DataType* stackArray;
    int top;
    int bottom;
}Stack;

//申明栈的函数
Stack* CreateStack(int length);        //创建一个新的栈
void ClearStack(Stack* stack);         //清空栈
void DestroyStack(Stack* stack);     //销毁栈
DataType Pop(Stack* stack);       //弹栈
void Push(Stack* stack, DataType data);     //压栈
int GetLength(Stack* stack);    //得到栈的大小
DataType GetSatckPeek(Stack* stack);     //取得栈顶元素

//测试主函数
int main()
{
    //定义栈的大小
    const int MAXLENGTH = 100;
    //创建一个栈
    Stack* stack = CreateStack(MAXLENGTH);
    //输入 10 个字符并入栈
    for (int i = 0; i <10; i++)
    {
        char ch;
        scanf("%c", &ch);
        //入栈
        Push(stack, ch);
    }
```

```
    //弹栈并输出直到栈空
    while (GetLength(stack)>0)
        printf("%c", Pop(stack));
DestroyStack(stack);
    return 0;
}

//栈函数的实现
/**
  *@brief 创建一个新的栈
  *@param length 栈的大小
  *@return 指向栈的指针，如创建失败，返回 NULL
  */
Stack* CreateStack(int length)
{
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    if (stack)
    {
        //分配栈空间
        stack->stackArray = (DataType*)malloc(length*sizeof(DataType));
        if (stack->stackArray == NULL)
            return NULL;
        //置为空栈
        stack->bottom = 0;
        stack->top = 0;
    }
    return stack;
}
/**
*@brief 清空栈
*@param stack 指向栈的指针
*/
void ClearStack(Stack* stack)
{
    stack->bottom = 0;
    stack->top = 0;
}
/**
*@brief 销毁栈
*@param stack 指向栈的指针
*/
```

```
void DestroyStack(Stack* stack)
{
	free(stack->stackArray);
	free(stack);
}
/**
*@brief 弹栈
*@param stack 指向栈的指针
*@return 弹出的栈顶元素，如果弹栈失败，返回 0
*/
DataType Pop(Stack* stack)
{
	if (stack->top > stack->bottom)
	{
		//如果栈不空，则出栈
		stack->top -= 1;
		return stack->stackArray[stack->top];
	}
	else
	{
		//栈已经空了，出栈失败
		return 0;
	}
}
/**
*@brief 压栈
*@param stack 指向栈的指针
*@param data 要入栈的元素
*/
void Push(Stack* stack, DataType data)
{
	//此处没有处理栈满的情况，因为无法取得栈的最大容量！
	stack->stackArray[stack->top] = data;
	stack->top++;
}
/**
*@brief 得到栈的大小
*@param stack 指向栈的指针
*@return 栈大小
*/
int GetLength(Stack* stack)
```

```
{
    return stack->top - stack->bottom;
}
/**
*@brief 取得栈顶元素，但是不出栈
*@param stack 指向栈的指针
*@return 栈顶元素，失败返回 0
*/
DataType GetSatckPeek(Stack* stack)
{
    return stack->top > stack->bottom ?
        stack->stackArray[stack->top - 1] : 0;
}
```

程序的运行结果如下：

```
0123456789
9876543210
```

8.3.2 队列

和栈相类似，队列是一种特殊的线性表，它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。在队列这种数据结构中，最先插入的元素将是最先被删除的元素；反之最后插入的元素将最后被删除的元素，因此队列又称为“先进先出”（FIFO—first in first out）的线性表。

队列可以用数组来存储，数组的上界即是队列所容许的最大容量。在队列的运算中需设两个索引下标：front，队头存放实际队头元素的前一个位置；rear，队尾存放实际队尾元素所在的位置。一般情况下，两个索引的初值设为 0 ，这时队列为空，没有元素。图 8-10 是一个队列的示意图：

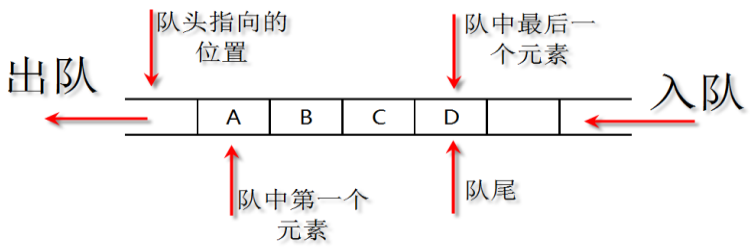


图 8-10 队列示意图

图 8-10 中，元素只能从队尾进入队列，只能从队头出队。也就是说要得到第 3 个元素 C，必须 A 和 B 先出队才可以。

当队头和队尾相等时，表示队列是空的，队尾到达了数组的上界，则队是满的。图 8-11 是一个队列变化的示意图。

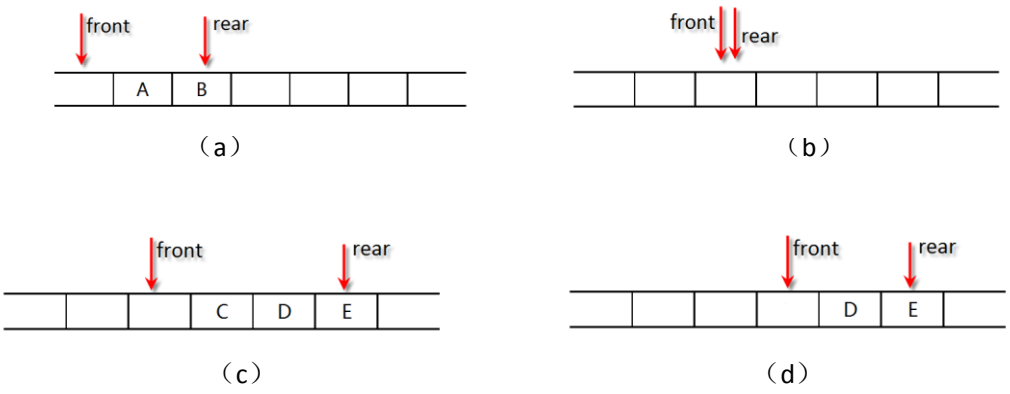


图 8-11 队列变化过程示意图

图 8-11 (a)中是这个队列中有 2 个元素的情形；(b)当 A 和 B 出队后，队列为空，此时队头等于队尾；(c)队中依次进入了 3 个元素 C、D 和 E；(d)C 出队后队列的情形

一个队列，常用的操作有：

- 求队列的长度 GetLength，得到队列中数据元素的个数
- 判断队列是否为空 IsEmpty，如果队列为空返回 true，否则返回 false
- 清空队列 Clear，使队列为空
- 入队 EnQueue，将新数据元素添加到队尾，队列发生变化
- 出队 DlQueue，将队头元素从队列中取出，队列发生变化
- 取队头元素 GetFront，返回队头元素的值，队列不发生变化

仔细观察图 8-11 的过程，会发现随着元素的出队和入队，队头和队尾均会不断地向后移动。当队尾移动到真个队列存储空间的最后一个位置时，如果还有元素要入队，则会发生溢出。因为队尾已经移到最后，没法再向后移动了。但实际上，队列中还是有空间的，因为有元素出队，也就是说，队头之前的空间是可以再用来存储数据的。如何利用空间呢？最直观的方法是将队列整个向前移动，但这样做效率并不高。一个较好的办法是将队列的头尾相连形成一个圆圈，这就是所谓的循环队列。循环队列的示意图如图 8-12 所示。当队尾和队头重叠时，队列为空还是满呢？我们约定，当队头和队尾相等时，队空。当队尾加 1 后等于队头时，队满。这样虽然浪费了一个存储空间（为什么？）但可以较为容易的区别队空和队满的情形。

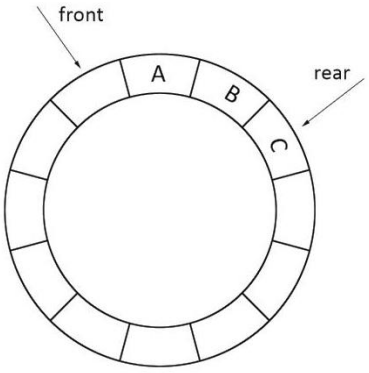


图 8-12 循环队列示意图

图 8-12 为一个循环队列的示意图，在循环队列中，空间可以反复利用

【例 8-5】一个队列的实现。

分析：使用固定大小的数组，实现队列的简单操作。

程序：

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

```
#include<stdlib.h>

typedef char DataType; //假设是字符队列

//定义队列结构
typedef struct QUEUE
{
	DataType* queArray;
	int front;		//队头
	int rear;		//队尾
}Queue;


//声明队列的方法
Queue* CreateQueue(int length);	//创建一个队列
void DestroyQueue(Queue*	queue);	//销毁队列
void ClearQueue(Queue* queue);	//清空队列
int GetQueueLength(Queue* queue);	//得到队列的长度
void EnQueue(Queue* queue, DataType data);	//入队
DataType DlQueue(Queue* queue);	//出队

// 主函数
int main()
{
	const int QueueMax = 100;	//队列最大容量

	//创建队列
	Queue* queue = CreateQueue(QueueMax);
	if (queue == NULL)
		return 1;		//创建失败，程序退出

	//入队操作，3 个元素进入队列
	EnQueue(queue, 'A');
	EnQueue(queue, 'B');
	EnQueue(queue, 'C');

	//一个元素出队,并显示
	printf("出队：%c\n", DlQueue(queue));

	//再入队 3 个后打印队列
	EnQueue(queue, 'D');
	EnQueue(queue, 'E');
	EnQueue(queue, 'F');
```

```
    //所有元素依次出队直到队空
    while (GetQueueLength(queue)>0)
        printf("出队：%c\n", DlQueue(queue));
    DestroyQueue(queue);
    return 0;
}

/**
 *@brief 创建一个队列
 *@param length 对列的容量
 *@return 指向队列的指针，失败返回 NULL
*/
Queue* CreateQueue(int length)
{
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    if (queue)    //不为空
    {
        //申请内存
        queue->queArray = (DataType*)malloc(length*sizeof(DataType));
        //失败则返回 NULL
        if (queue->queArray == NULL)
            return NULL;
        //清空队列
        queue->front = 0;
        queue->rear = 0;
    }
    return queue;
}
/**
*@brief   销毁队列
*@param queue 指向队列的指针
*/
void DestroyQueue(Queue*   queue)
{
    free(queue->queArray);
    free(queue);
}
/**
*@brief   清空队列
```

```
*@param queue  指向队列的指针
*/
void ClearQueue(Queue* queue)
{
	queue->front = 0;
	queue->rear = 0;
}
/**
*@brief    得到队列的长度
*@param queue  指向队列的指针
*@return  队列中的元素个数
*/
int GetQueueLength(Queue* queue)
{
	return queue->rear - queue->front;
}
/**
*@brief    入队
*@param queue  指向队列的指针
*@param data  要入队的元素
*/
void EnQueue(Queue* queue, DataType data)
{

	//没有考虑的队满的情况，没有记录对列的最大容量!
	queue->queArray[++queue->rear] = data;
}
/**
*@brief    出队
*@param queue  指向队列的指针
*@return  出队的元素值，如队空，返回 0
*/
DataType DlQueue(Queue* queue)
{
	return queue->rear - queue ->front > 0 ?
		queue->queArray[++queue->front] : 0;
}
```

程序的运行结果：

出队：A

出队：B

出队：C
出队：D
出队：E
出队：F

【例 8-6】循环队列

分析：使用固定大小的数组，实现一个循环队列。这里的代码和例 8-5 基本是相同的，只是在入队、出队和打印等操作上注意对对头和队尾就队列的总长度取余。此外注意到本例在队列的定义中增加了一项来记录队列的最大容量。

代码：

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

typedef char DataType;       //假设是字符队列

//定义队列结构
typedef struct QUEUE
{
    DataType* queArray;
    int front;        //队头
    int rear;         //队尾
    int maxLength;   //队的最大容量
}Queue;

//声明队列的方法
Queue* CreateQueue(int length);            //创建一个队列
void DestroyQueue(Queue*    queue);     //销毁队列
void ClearQueue(Queue* queue);          //清空队列
int GetQueueLength(Queue* queue);     //得到队列的长度
void EnQueue(Queue* queue, DataType data);     //入队
DataType DlQueue(Queue* queue);       //出队

// 主函数
int main()
{
    const int QueueMax = 100;     //队列最大容量

    //创建队列
    Queue* queue = CreateQueue(QueueMax);
    if (queue == NULL)
        return 1;        //创建失败，程序退出
```

```
    //入队操作，3 个元素进入队列
    EnQueue(queue, 'A');
    EnQueue(queue, 'B');
    EnQueue(queue, 'C');

    //一个元素出队,并显示
    printf("出队：%c\n", DlQueue(queue));

    //再入队 3 个后打印队列
    EnQueue(queue, 'D');
    EnQueue(queue, 'E');
    EnQueue(queue, 'F');

    //所有元素依次出队直到队空
    while (GetQueueLength(queue)>0)
        printf("出队：%c\n", DlQueue(queue));
    DestroyQueue(queue);
    return 0;
}

/**
*@brief 创建一个队列
*@param length 对列的容量
*@return 指向队列的指针，失败返回 NULL
*/
Queue* CreateQueue(int length)
{
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    if (queue)    //不为空
    {

        //申请内存
        queue->queArray = (DataType*)malloc(length*sizeof(DataType));

        //失败则返回 NULL
        if (queue->queArray == NULL)
            return NULL;

        //清空队列
        queue->front = 0;
        queue->rear = 0;
        queue->maxLength = length;
    }
    return queue;
```

```
}
/**
*@brief    销毁队列
*@param queue  指向队列的指针
*/
void DestroyQueue(Queue*    queue)
{
      free(queue->queArray);
      free(queue);
}
/**
*@brief    清空队列
*@param queue  指向队列的指针
*/
void ClearQueue(Queue* queue)
{
      queue->front = 0;
      queue->rear = 0;
}
/**
*@brief    得到队列的长度
*@param queue  指向队列的指针
*@return  队列中的元素个数
*/
int GetQueueLength(Queue* queue)
{
      return queue->rear >= queue->front ?
            queue->rear - queue->front :
            queue->rear - queue->front + queue->maxLength;
}
/**
*@brief    入队
*@param queue  指向队列的指针
*@param data  要入队的元素
*/
void EnQueue(Queue* queue, DataType data)
{
      if ((queue->rear + 1) % queue->maxLength == queue->front)
      {
            printf("队列已满，无法完成入队操作");
      }
```

```
    else
    {
        //队尾向后移动 1 位
        queue->rear = (queue->rear + 1) % queue->maxLength;
        //入队
        queue->queArray[queue->rear] = data;
    }
}
/**
*@brief   出队
*@param queue  指向队列的指针
*@return  出队的元素值，如队空，返回 0
*/
DataType DlQueue(Queue* queue)
{
    if (GetQueueLength(queue) > 0)          //队不空
    {
        queue->front = (queue->front + 1) % queue->maxLength;
        return queue->queArray[queue->front];
    }
    else
    {
        return 0;                           //队是空的，返回 0
    }
}
```

8.4 图和树

前几节讲述了一些线性结构的数据，而图和数则是非线性的数据结构。同时，现实中的很多问题也是用线性数据结构而无法描述的，需要借助非线性的数据结构来描述。

8.4.1 图的基本概念

1736 年，著名数学家欧拉(Euler)发表了著名论文“柯尼斯堡七座桥”的论文中，首先使用图的方法解决了柯尼斯堡七桥问题，从而欧拉也被誉为图论之父。这个问题是基于一个现实生活中的事例：当时东普鲁士柯尼斯堡（Königsberg，今日俄罗斯加里宁格勒）市区跨普列戈利亚河（Pregel）两岸，河中心有两个小岛。小岛与河的两岸有 7 座桥连接。于是，7 座桥将 4 块陆地连接了起来，如图 8-14 所示。而城里的居民想在散步的时候从任何一块陆地出发，经过每座桥 1 次且仅经过 1 次最后返回原来的出发点。当地的居民和游客做了不少尝试，却都没有成功，而欧拉最终解决了这个问题并断言这样的回路是不存在的。

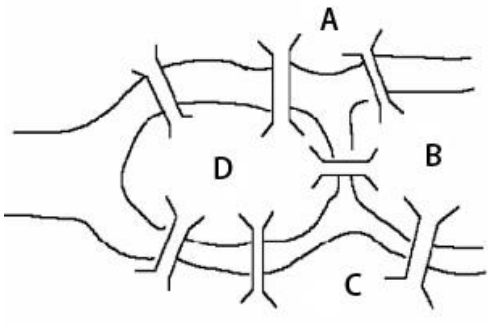


图 8-13 柯尼斯堡七桥问题示意图

欧拉在解决问题时，用 4 个结点来表示陆地 A、B、C 和 D，凡是陆地间有桥连接的，便在 2 点间连一条线，于是图 8-13 转换为图 8-14。

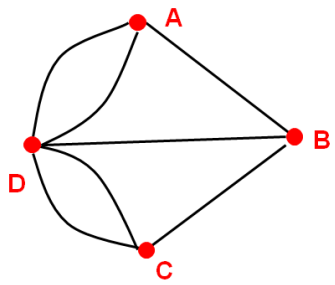


图 8-14 柯尼斯堡七桥问题抽象为图后的表示

此时，问题则转化为从图 8-14 中的 A、B、C、D 任一点出发，通过每条边一次且仅一次后回到原出发点的回路是否存在？欧拉断言了这个回路是不存在的，理由是从图 8-14 中的任一点出发，为了能够回到原出发点，则要求与每个点关联的边数均为偶数。这样才能保证从一条边进入某点后可以从另外一条边出来。而图 8-14 中的 A、B、C、D 全部都与奇数边关联，因此回路是不存在的。

由上面的例子我们也看到，所谓图（graph）是由结点或称顶点(vertex)和连接结点的边（edge）所构成的图形。使用 V(G)表示图 G 终所有结点的集合，E(G)表示图 G 中所有边的集合。则图 G 可记为<V(G),E(G)>或<V,E>。有 n 个顶点和 m 条边的图记为(n,m)图或称为 n 阶图。

【例 8-7】4 个城市 v1、v2、v3 和 v4。v1 和其他 3 个城市都有道路连接，v2 和 v3 之间有道路连接，画出图并用集合表示该图。

显然结点集合 V={v1，v2，v3，v4}，边集合 E={V1 和 v2 之间的边，v1 和 v3 之间的边，v1 和 v4 之间的边，v2 和 v3 之间的边}。画出的图如 8-15 所示。

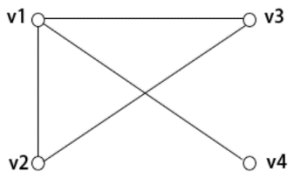


图 8-15 例 8-7 中的图

更一般的，边可以用结点对来表示，或者说用结点 V 的向量积来表示：

V={v1，v2，v3，v4}

E={(v1,v2),(v1,v3),(v1,v4),(v2,v3)}

在图中，如果边不区分起点和终点，这样的边称为无向边。所有边都是无向边的图称为无向图，如图 6-18 就是一个无向图。反之，若边区分起点和终点，则为有向边，所有边都是有向边的图称为有向图。在图中，有向边使用带有箭头的线段表示，有起点指向终点。在集合中则用有序对<v1,v2>来表示，图 8-16 是一个示例。

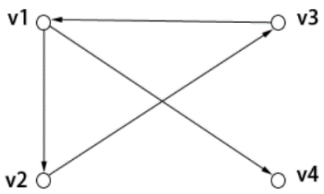


图 8-16 一个有向图的示例

在图 8-16 中：

V={v1，v2，v3，v4}

E={<v1,v2>,<v1,v4>,<v3,v1>,<v2,v3>}

结点的度则是指和结点关联的边的个数。如在图 8-16 中，v1 的度是 3，v2 和 v3 的度是 2，v4 的度是 1。对于有向图，则区分为出度和入度，由结点指向外的边的个数为出度，反之为入度。如图 8-17 中，v1 的出度为 2，入度为 1，v4 的出度为 0，入度为 1。

图在计算机中如何存储，则是人们普遍关心的一个问题。简单的方法是将图用一个二维矩阵来表示，这样的矩阵通常称为邻接矩阵。在此我们不系统讨论，仅以图 8-16 的存储为例来说明。

【例 8-8】将简单有向图（图 8-16）以邻接矩阵的方式存储到计算机中。

要以邻接矩阵的方式存储，首先需要对结点指定一个次序。在此，我们就以结点的下标从小到大为序，排列为 v_1，v_2，v_3，v_4。然后使用一个 4*4 的矩阵来存储该图，矩阵中的元素只有 2 个取值：0 或者 1。对于 2 个结点 v_i 和 v_j，若 v_i 和 v_j 之间存在一条边，则对应的矩阵元素 a_{ij}=1，反之则为 0。图 8-17 示例的有向图存储矩阵如下：

$$\begin{array}{c} \\ v1 \\ v2 \\ v3 \\ v4 \end{array} \begin{array}{c} \begin{array}{cccc} v1 & v2 & v3 & v4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

图 8-17 存储的邻接矩阵

容易看出，矩阵中 1 的个数对应图中边的个数，而对角线的元素则全为 0。

8.4.2 带权图和最短路径

图的问题异常的复杂，甚至是一门完整的学科—图论。在此无法对图有一个完整系统的讨论。而为了使读者对图有进一步的认识，作为一个例子，简单介绍带权图及最短路径的算法，并以此结束对图的讨论。

在处理有关图的实际问题时，往往有值的存在，比如公里数，运费，城市，人口数以及电话部数等。一般这个值称为权值，在图中，将每条边都有一个非负实数对应的图称为带权图或赋权图。这个实数称为这条边的权。根据不同的实际情况，权数的含义可以各不相同。例如，可用权数代表两地之间的实际距离或行车时间，也可用权数代表某工序所需的加工时间等。如图 8-18 所示便是一个带权图。

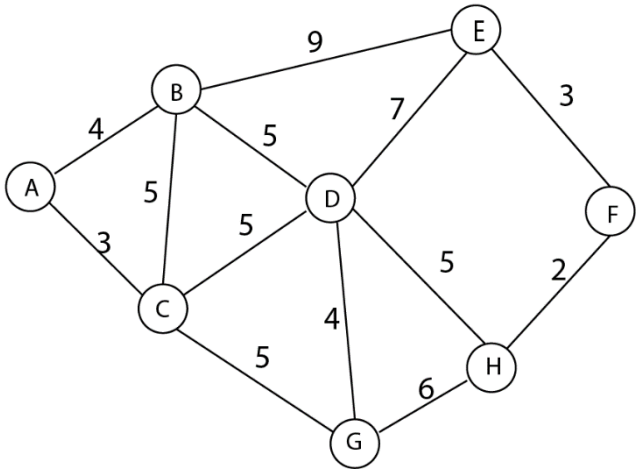


图 8-18 带权图

对图 8-18 所示的无向带权图求最短路径是一个经常遇到的很实际的问题。假设在图中的 A 到 G 点表示 8 个村庄，边表示村庄之间的道路。边上的权值表示距离。现在的问题是从 A 到 F 最短的距离是多少？

求最短路的算法是 E. W. Dijkstra 于 1959 年提出来的，这是至今公认的求最短路径的最好方法，我们称它 Dijkstra 算法。假定给定带权图 G，要求 G 中从 v0 到 v 的最短路径，Dijkstra 算法的基本思想是：

将图 G 中结点集合 V 分成两部分：一部分称为具有 P 标号的集合，另一部分称为具有 T 标号的集合。所谓结点 a 的 P 标号是指从 v0 到 a 的最短路的路长；而结点 b 的 T 标号是指从 v0 到 b 的某条路径的长度。Dijkstra 算法中首先将 v0 取为 P 标号结点，其余的结点均为 T 标号结点，然后逐步地将具有 T 标号的结点改为 P 标号结点，当目的结点也被改为 P 标号时，则找到了从 v0 到 v 的一条最短路径。下面通过一个例子给出实际的算法步骤。

【例 8-9】计算图 8-18 所示的带权图中，从 A 点到 F 点的最短路径。

（1） 首先，将起点 A 划归为 P 标号集合，其余的节点均为 T 结点。A 到 A 的距离为 0，所以 A 的 P 标号为 0。

（2） 更新 T 中节点到 A 的距离，如和 A 相邻（有边连接）则就是边的权值。如和 A 没有直接的边连接则距离是无穷大。

（3） 在 T 中找到一个值最小的节点，并将其划归到 P 集合。此时，计算的结果如图 6-23 所示（C 结点进入 P 集合）：

（4） 根据新进入的 C 节点，更新与 C 相连的结点的值。新值等于 C 的 P 节点值加上到与其相连的结点的距离（边的权值）。更新的算法是如果新值小于原有的值，则用新的值取代，否则保持原有值不变。

（5） 重复步骤（3）和（4）直到目标点进入 P 集合。

图 8-19 到图 8-25 演示了上述过程。

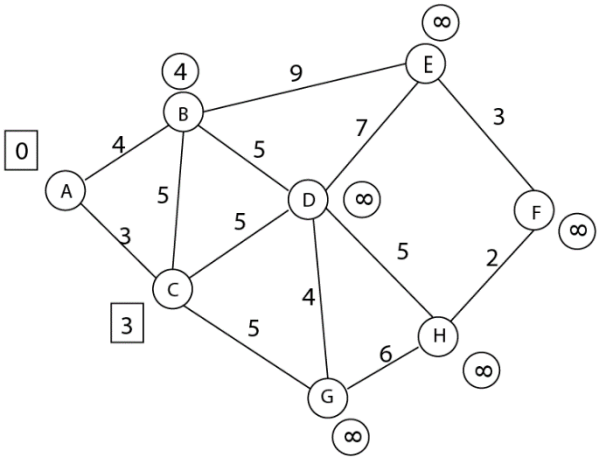


图 6-19

图 6-19 中，A 到 B 的距离为 4，到 C 的距离为 3，到其余结点的距离为无穷大。由于 C 节点的值最小，因此 C 进入 P 集合（P 集以方框表示，T 集用圆圈表示）。

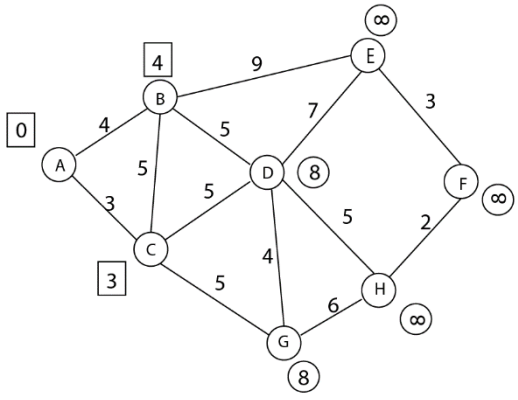


图 8-20

图 8-20 中，节点 C 进入 P 集合后，到 B 的距离为 3+5=8 大于 B 原来的 4，因此 B 的值不变。而到 D 和 G 的值均为 8，均小于原来的无穷大，因此用 8 取代原来的值。之后，在 T 中，B 的值为 4 最小，B 进入 P 集合。

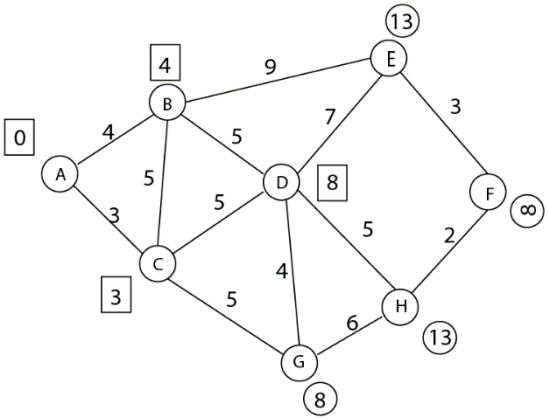


图 8-21

图 21 中，节点 B 进入后更新与 B 连接的 D 和 E 值。其中 D 的值不变，E 为 13。此时 D 和 G 均有最小值 8，任取一进入 P，在此是取的是 D。然后又更新了 H 的值。G 的原值小于 8+4，因此保持不变。

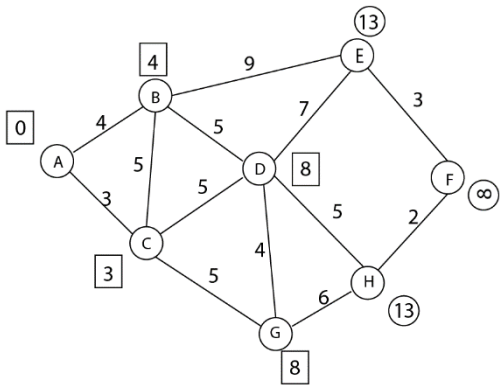


图 8-22 节点 G 的值最小进入 P，H 的值未变

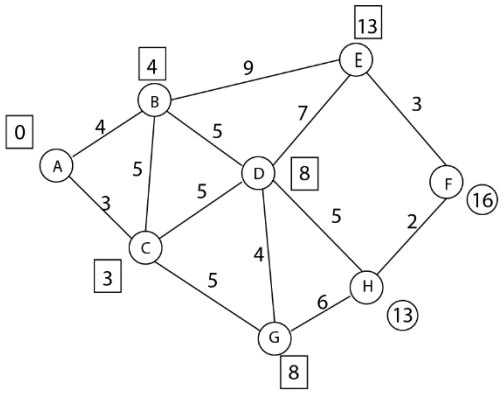


图 8-23 任选 E 进入 P，F 值变为 16

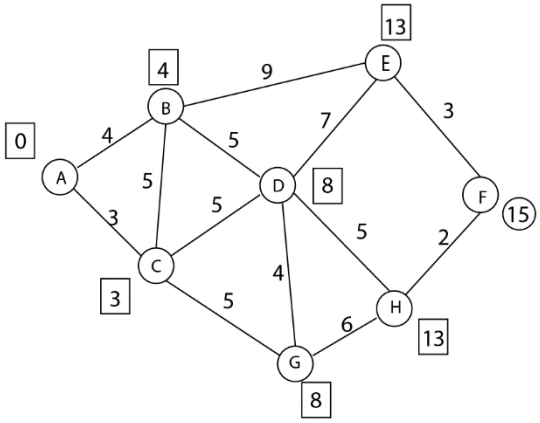


图 8-24 节点 H 进入 P，F 的值变为 15

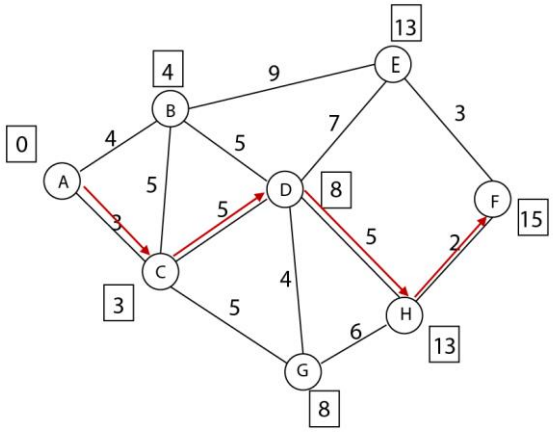


图 8-25

图 8-25 中，终点 F 进入 P 集合，运算结束。从 A 到 F 的最短距离为 15。而事实上，对于每一个 P 中的结点值，计算出了从 A 到该结点的最短距离，如到 E 的最短距离为 13。而找到最短路径的方法是用 F 点的 P 值减去边的权值，倒推回 A 点。如 F 的值 15-2=13 和 H 吻合，而不是 E（因为 15-3=12 不等于 E 的 13）

8.4.3 树的基本概念

树可以看作是一个特殊的有向图。对于一个有向图，如果

（1） 存在一个特殊的节点 r，其入度等于 0；

（2） 除了 r 外的其他结点的入度均为 1。

（3） R 到图中其他结点均有路可达。

满足这样的图称为树。其中入度为 0 的结点称为根，出度为 0 的结点称为叶子。出度不为 0 的结点称为分枝点。如图 8-26 所示。

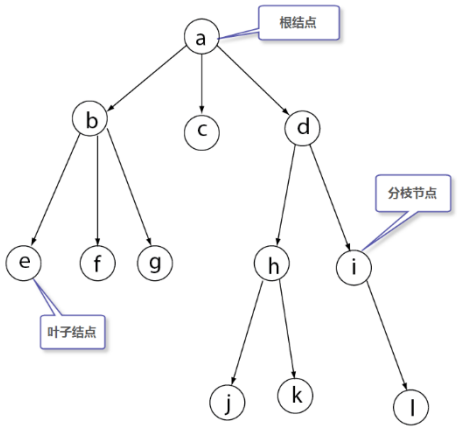


图 8-26 树

在画树的图的时候，由于所有的箭头方向都是一致的，所以箭头常常省略，如图 8-27 所示。树是有层次的，指的是从根到该结点的距离。称距根最远的叶子的层数为树的高度。图 8-27 的树的高度为 3。同一层次之间的结点称为兄弟，上一层次的为父亲，下一层次是儿子，如图 8-27 中对 h 结点的描述。

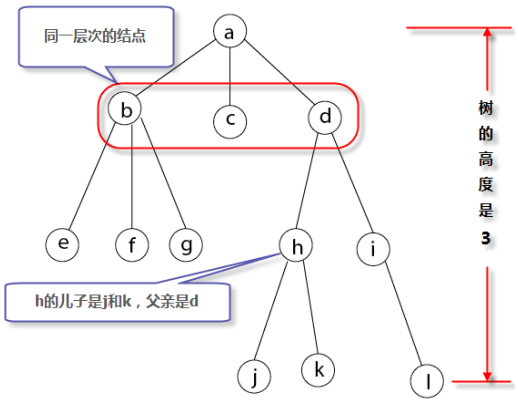


图 8-27 树的高度及层次关系

对于一棵树而言，若所有结点的入度均小于等于 m，则称此树为 m 叉树。如果每个结点的入度都相等且都等于 m，则称此树为完全 m 叉树。在计算机学科经常应用的是二叉树。

8.4.4 二叉树

二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树的每个结点至多只有二棵子树(不存在出度大于 2 的结点)，二叉树的子树有左右之分，次序不能颠倒。

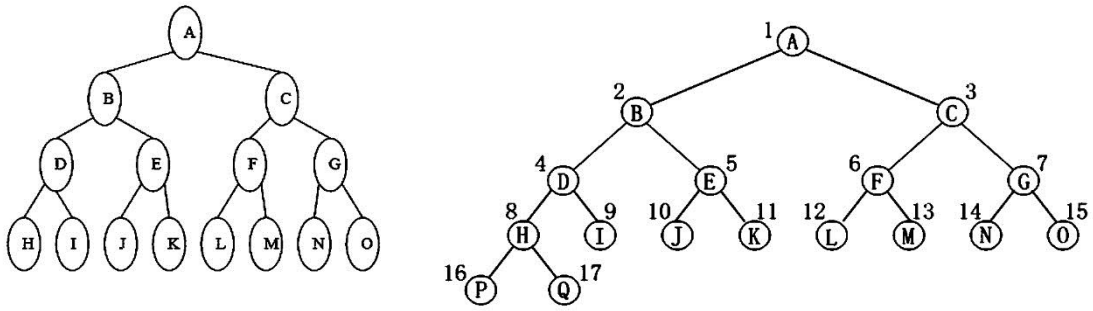
二叉树具有如下性质：

- 二叉树的第 i 层至多有 2^{i-1} 个结点；
- 深度为 k 的二叉树至多有 2^{k-1} 个结点；

- 二叉树的结点个数可以为 0；
- 二叉树的结点有左、右之分。

一棵深度为 k，且有 2^{k-1} 个节点的二叉树，称为**满二叉树**（Full Binary Tree）。这种树的特点是每一层上的节点数都是最大节点数。

而对于深度为 K 的，有 N 个结点的二叉树，当且仅当其每一个结点都与深度为 K 的满二叉树中编号从 1 至 n 的结点一一对应时称之为完全二叉树（Complete Binary Tree）。也就是说，若一棵二叉树至多只有最下面的两层上的结点的度数可以小于 2，并且最下层上的结点都集中在该层最左边的若干位置上，则此二叉树成为完全二叉树。具有 n 个节点的完全二叉树的深度为 $\log_2 n+1$。深度为 k 的完全二叉树，至少有 2^{k-1} 个节点，至多有 2^k-1 个节点。图 8-28（a）（b）分别是满二叉树和完全二叉树的示例。



（a）满二叉树　　（b）完全二叉树

图 8-28 满二叉树和完全二叉树

8.4.5 树的遍历

树的遍历是树的一种重要的运算。所谓遍历是指对树中所有结点的信息的访问，即依次对树中每个结点访问一次且仅访问一次。树的遍历有 2 种方式：先根的遍历方式，即先访问树的根节点，然后依次先根的访问根的每一颗子树；后根遍历，即依次后根的访问根的每一颗子树，最后访问根节点。

对于图 8-29 的树，采用先根的方式，节点访问的次序依次为：

a b e f g c d h j k i l

采用后根的方式，节点访问的次序依次为：

e f g b c j k h l i d a

习 题

1. 什么是数据的线性存储结构，什么是数据的非线性存储结构？
2. 简述线性表的操作。
3. 假设电话号码本由人名和一个电话号码组成，设计一个线性表，存储 7 个人的电话号码簿。
4. 设栈 S 中存储的是字符数据，自栈底到栈顶依次为 A，C，D。经过 2 次出栈操作并

将 E 压入栈，此时栈中的数据是什么？

5. 使用栈，检查表达式（2+3）*a*（3+b）/（2*（12+8）的括号是否匹配。
6. 编写程序，输入一行文本，然后使用栈逆序显示该行文本。
7. 编写程序，用栈来判断一个字符串是否为回文（即顺读和倒读都相同的字符串）。程序忽略字符串中的大小写，空格和标点符号。
8. 设计一个队列，将整数 3，4，5 进入队列，打印该队列，将队列的前 2 个元素出队，随后将 11 和 12 入队，再次打印队列。
9. 对于图 8-12 的循环队列，在该图的基础上，将 1，2，3，4，5 入队，并将 2 个元素出队后，画出队列目前的状态。
10. 将书中 8-19 的带权图使用邻接矩阵的方式存储到计算机中，试写出该矩阵。（提示：是一个对角线元素为 0 的对称矩阵）。