数据的逻辑结构和存储结构（物理结构）详解

《数据结构有哪些》一节讲到，数据的存储方式可分为线性表、树和图三种存储结构，而每种存储结构又可细分为顺序存储结构和链式存储结构。数据存储方式如此之多，针对不同类型的数据选择合适的存储方式是至关重要的。

那么，到底如何选择呢？数据存储结构的选择取决于两方面，即数据的逻辑结构和存储结构（又称物理结构）。

逻辑结构

数据的逻辑结构，简单地理解，就是指的数据之间的逻辑关系。

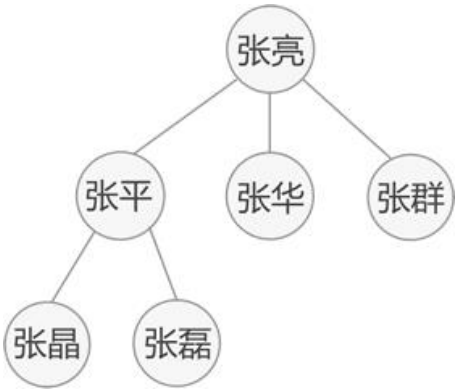


图 1 家庭成员关系图

例如，图 1 显示是一张家庭的成员关系图，从图中可以看到，张平、张华和张群是兄弟，他们的父亲是张亮，其中张平有两个儿子，分别是张晶和张磊。

以上所说，父子、兄弟等这些关系都指的是数据间的逻辑关系，假设我们要存储这样一张家庭成员关系图，不仅要存储张平、张华等数据，还要存储它们之间的关系，两者缺一不可。

> 一组数据成功存储到计算机的衡量标准是要能将其完整的复原。例如图 1 所示的成员关系图，如果所存储的数据能将此成员关系图彻底复原，则说明数据存储成功。

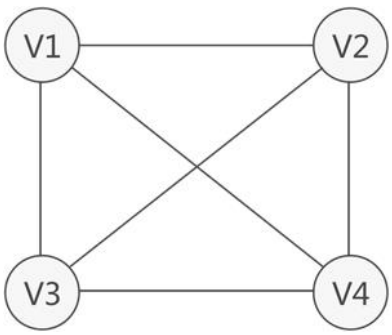


图 2 “多对多”关系示意图

数据之间的逻辑关系可细分为三类，“一对一”、“一对多”和“多对多”：

- “一对一”：类似集合 {1,2,3,...,n} 这类的数据，每个数据的左侧有且仅有一个数据与其相邻（除 1 外）；同样，每个数据的右侧也只有一个数据与其相邻（除 n 外），所有的数据都是如此，就说数据之间是“一对一”的逻辑关系；

- “一对多”：图 1 中的数据就属于“一对多”，因为对于张平来说，有且仅有一个父亲（张亮），但是有 2（多）个孩子；
- “多对多”：拿图 2 来说，从 V1 可以到达 V2、V3、V4，同样，从 V2、V3、V4 也可以到达 V1，对于 V1、V2、V3 和 V4 来说，它们之间就是“多对多”的关系；

通过学习数据结构，我们可以学到 3 种存储结构分别存储这 3 类逻辑关系的数据，换句话说：

1. 线性表用于存储具有“一对一”逻辑关系的数据；
2. 树结构用于存储具有“一对多”关系的数据；
3. 图结构用于存储具有“多对多”关系的数据；

由此，我们可以通过分析数据之间的逻辑关系来决定使用哪种存储结构，但具体使用顺序存储还是链式存储，还要通过数据的物理结构来决定。

存储结构（物理结构）

数据的存储结构，也就是物理结构，指的是数据在物理存储空间上选择集中存放还是分散存放。假设要存储大小为 10G 的数据，则集中存放就如图 3a）所示，分散存放就如图 3b）所示。

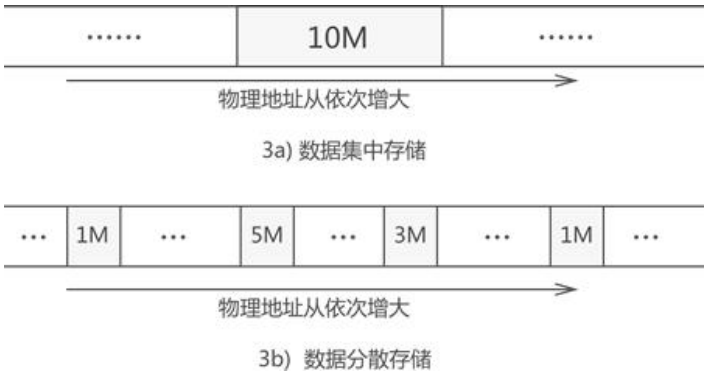


图 3 数据的物理存储方式

如果选择集中存储，就使用顺序存储结构；反之，就使用链式存储。至于如何选择，主要取决于存储设备的状态以及数据的用途。

我们知道，集中存储（底层实现使用的是数组）需要使用一大块连续的物理空间，假设要存储大小为 1G 的数据，若存储设备上没有整块大小超过 1G 的空间，就无法使用顺序存储，此时就要选择链式存储，因为链式存储是随机存储数据，占用的都是存储设备中比较小的存储空间，因此有一定几率可以存储成功。

并且，数据的用途不同，选择的存储结构也不同。将数据进行集中存储有利于后期对数据进行遍历操作，而分散存储更有利于后期增加或删除数据。因此，如果后期需要对数据进行大量的检索（遍历），就选择集中存储；反之，若后期需要对数据做进一步更新（增加或删除），则选择分散存储。

> 至于为什么，我们会在详解两种存储结构时告知大家。

数据结构和算法的关系和区别

由于大量数据结构教程中都将数据结构的知识和算法掺杂起来讲，使很多初学者认为数据结构就是在讲算法，这样理解是不准确的。

数据结构和算法之间完全是两个相互独立的学科，如果非说它们有关系，那也只是互利共赢、“1+1>2”的关系。

最明显的例子，如果你认为数据结构是在讲算法，那么大学我们还学《算法导论》，后者几乎囊括了前者使用的全部算法，有什么必要同时开设这两门课程呢？

我们还可以从分析问题的角度去理清数据结构和算法之间的关系。通常，每个问题的解决都经过以下两个步骤：

1. 分析问题，从问题中提取出有价值的数据，将其存储；
2. 对存储的数据进行处理，最终得出问题的答案；

数据结构负责解决第一个问题，即数据的存储问题。通过前面的学习我们知道，针对数据不同的逻辑结构和物理结构，可以选出最优的数据存储结构来存储数据。

而剩下的第二个问题，属于算法的职责范围。算法，从表面意思来理解，即解决问题的方法。我们知道，评价一个算法的好坏，取决于在解决相同问题的前提下，哪种算法的效率最高，而这里的效率指的就是处理数据、分析数据的能力。

因此我们得出这样的结论，数据结构用于解决数据存储问题，而算法用于处理和分析数据，它们是完全不同的两类学科。

也正因为如此，你可以认为数据结构和算法存在“互利共赢、1+1>2”的关系。在解决问题的过程中，数据结构要配合算法选择最优的存储结构来存储数据，而算法也要结合数据存储的特点，用最优的策略来分析并处理数据，由此可以最高效地解决问题。

图 1 顺序表存储数据示意图

例如，有这样一个问题，计算“1+2+3+4+5”的值。这个问题我们可以这样来分析：

- 计算 1、2、3、4 和 5 的和，首先要选择一种数据存储方式将它们存储起来，通过前面的学习我们知道，数据之间具有“一对一”的逻辑关系，最适合用线性表来存储。结合算法的实现，我们选择顺序表来存储数据（而不是链表），如图 1 所示；
- 接下来，我们选择算法。由于数据集中存放，因此我们可以设计这样一个算法，使用一个初始值为 0 的变量 num 依次同存储的数据做“加”运算，最后得到的新 num 值就是最终结果。

选择顺序表而不是链表的原因，是顺序表遍历数据比链表更高效。后续讲顺序表时会做详细介绍。

顺序表和链表的优缺点（区别、特点）详解

顺序表和链表由于存储结构上的差异，导致它们具有不同的特点，适用于不同的场景。本节就来分析它们的特点，让读者明白 "在什么样的场景中使用哪种存储结构" 更能有效解决问题。

通过系统地学习顺序表和链表我们知道，虽然它们同属于线性表，但数据的存储结构有本质的不同：

- 顺序表存储数据，需预先申请一整块足够大的存储空间，然后将数据按照次序逐一存储，数据之间紧密贴合，不留一丝空隙，如图 1a) 所示；

- 链表的存储方式与顺序表截然相反，什么时候存储数据，什么时候才申请存储空间，数据之间的逻辑关系依靠每个数据元素携带的指针维持，如图 1b) 所示；

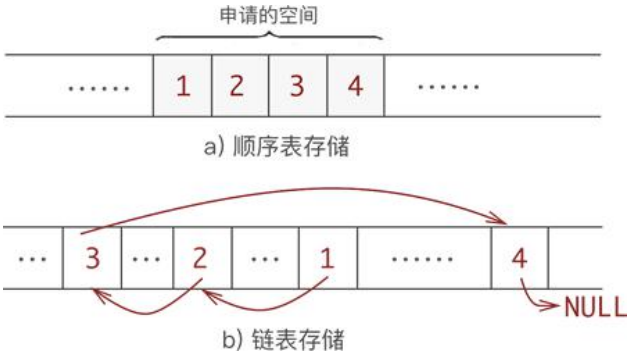


图 1 顺序表和链表的存储结构对比

基于不同的存储结构，顺序表和链表有以下几种不同。

开辟空间的方式

顺序表存储数据实行的是 "一次开辟，永久使用"，即存储数据之前先开辟好足够的存储空间，空间一旦开辟后期无法改变大小（使用动态数组的情况除外）。

而链表则不同，链表存储数据时一次只开辟存储一个节点的物理空间，如果后期需要还可以再申请。

因此，若只从开辟空间方式的角度去考虑，当存储数据的个数无法提前确定，又或是物理空间使用紧张以致无法一次性申请到足够大小的空间时，使用链表更有助于问题的解决。

空间利用率

从空间利用率的角度上看，顺序表的空间利用率显然要比链表高。

这是因为，链表在存储数据时，每次只申请一个节点的空间，且空间的位置是随机的，如图 2 所示：

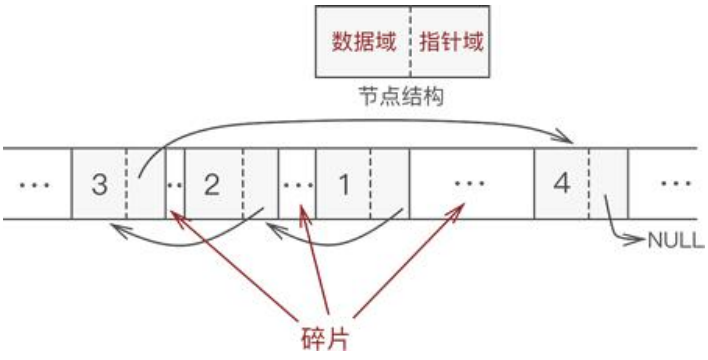


图 2 链表结构易产生碎片

这种申请存储空间的方式会产生很多空间碎片，一定程序上造成了空间浪费。不仅如此，由于链表中每个数据元素都必须携带至少一个指针，因此，链表对所申请空间的利用率也没有顺序表高。

> 空间碎片，指的是某些容量很小（1KB 甚至更小）以致无法得到有效利用的物理空间。

时间复杂度

解决不同类型的问题，顺序表和链表对应的时间复杂度也不同。

根据顺序表和链表在存储结构上的差异，问题类型主要分为以下 2 类：

1. 问题中主要涉及访问元素的操作，元素的插入、删除和移动操作极少；
2. 问题中主要涉及元素的插入、删除和移动，访问元素的需求很少；

第 1 类问题适合使用顺序表。这是因为，顺序表中存储的元素可以使用数组下标直接访问，无需遍历整个表，因此使用顺序表访问元素的时间复杂度为 O(1)；而在链表中访问数据元素，需要从表头依次遍历，直到找到指定节点，花费的时间复杂度为 O(n)；

第 2 类问题则适合使用链表。链表中数据元素之间的逻辑关系靠的是节点之间的指针，当需要在链表中某处插入或删除节点时，只需改变相应节点的指针指向即可，无需大量移动元素，因此链表中插入、删除或移动数据所耗费的时间复杂度为 O(1)；而顺序表中，插入、删除和移动数据可能会牵涉到大量元素的整体移动，因此时间复杂度至少为 O(n)；

综上所述，不同类型的场景，选择合适的存储结构会使解决问题效率成倍数地提高。

静态链表和动态链表区别详解（无师自通）

前面学习了链表，其存储结构如图 1 所示：

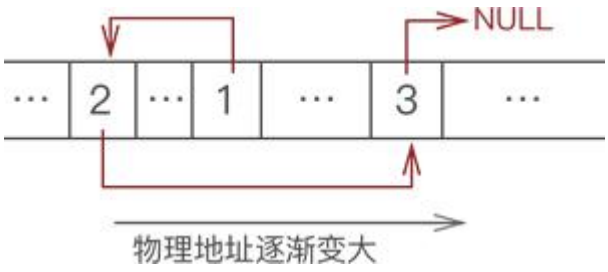


图 1 链表存储结构

类似图 1 这样的链表，它更喜欢人们称它为 "动态链表"。

随后又接触了静态链表。同样是存储图 1 中的数据 {1,2,3}，使用静态链表存储数据的状态如图 2 所示：

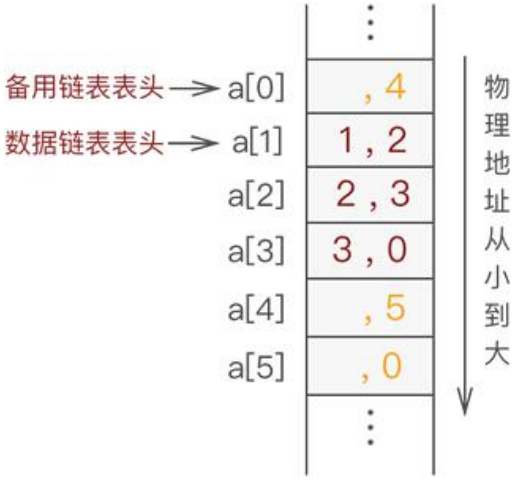


图 2 静态链表存储数据示意图

为了让读者分清动态链表和静态链表，本节来分析一下它们之间的区别和联系。

静态链表和动态链表的共同点是，数据之间"一对一"的逻辑关系都是依靠指针（静态链表中称"游标"）来维持，仅此而已。

静态链表

使用静态链表存储数据，需要预先申请足够大的一整块内存空间，也就是说，静态链表存储数据元素的个数从其创建的那一刻就已经确定，后期无法更改。

比如，如果创建静态链表时只申请存储 10 个数据元素的空间，那么在使用静态链表时，数据的存储个数就不能超过 10 个，否则程序就会发生错误。

不仅如此，静态链表是在固定大小的存储空间内随机存储各个数据元素，这就造成了静态链表中需要使用另一条链表（通常称为"备用链表"）来记录空间存储空间的位置，以便后期分配给新添加元素使用，如图 2 所示。

这意味着，如果你选择使用静态链表存储数据，你需要通过操控两条链表，一条是存储数据，另一条是记录空闲空间的位置。

动态链表

使用动态链表存储数据，不需要预先申请内存空间，而是在需要的时候才向内存申请。也就是说，动态链表存储数据元素的个数是不限的，想存多少就存多少。

同时，使用动态链表的整个过程，你也只需操控一条存储数据的链表。当表中添加或删除数据元素时，你只需要通过 malloc 或 free 函数来申请或释放空间即可，实现起来比较简单。

双向循环链表及创建（C 语言）详解

我们知道，单链表通过首尾连接可以构成单向循环链表，如图 1 所示：

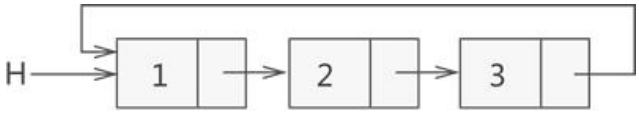


图 1 单向循环链表示意图

同样，双向链表也可以进行首尾连接，构成双向循环链表。如图 2 所示：

图 2 双向循环链表示意图

当问题中涉及到需要 "循环往复" 地遍历表中数据时，就需要使用双向循环链表。例如，前面章节我们对约瑟夫环问题进行了研究，其实约瑟夫环问题有多种玩法，每次顺时针报数后，下一轮可以逆时针报数，然后再顺时针……一直到剩下最后一个人。解决这个问题就需要使用双向循环链表结构。

双向循环链表的创建

创建双向循环链表，只需在创建完成双向链表的基础上，将其首尾节点进行双向连接即可。

C 语言实现代码如下：

```
//创建双向循环链表
line* initLine(line * head){
    head=(line*)malloc(sizeof(line));
    head->prior=NULL;
    head->next=NULL;
    head->data=1;
    line * list=head;
    for (int i=2; i<=3; i++) {
        line * body=(line*)malloc(sizeof(line));
        body->prior=NULL;
        body->next=NULL;
        body->data=i;
        list->next=body;
        body->prior=list;
        list=list->next;
    }
    //通过以上代码，已经创建好双线链表，接下来将链表的首尾节点进行双向连接
    list->next=head;
    head->prior=list;
    return head;
}
```

通过向 main 函数中调用 initLine 函数，就可以成功创建一个存储有 {1,2,3} 数据的双向循环链表，其完整的 C 语言实现代码为：

```
#include <stdio.h>
#include <stdlib.h>
typedef struct line{
    struct line * prior;
    int data;
    struct line * next;
}line;

line* initLine(line * head);
void display(line * head);
int main() {
    line * head=NULL;
    head=initLine(head);
    display(head);
    return 0;
}
//创建双向循环链表
line* initLine(line * head){
    head=(line*)malloc(sizeof(line));
    head->prior=NULL;
    head->next=NULL;
    head->data=1;
    line * list=head;
    for (int i=2; i<=3; i++) {
        line * body=(line*)malloc(sizeof(line));
        body->prior=NULL;
        body->next=NULL;
        body->data=i;

        list->next=body;
        body->prior=list;
        list=list->next;
    }
    //通过以上代码，已经创建好双线链表，接下来将链表的首尾节点进行双向连接
    list->next=head;
```

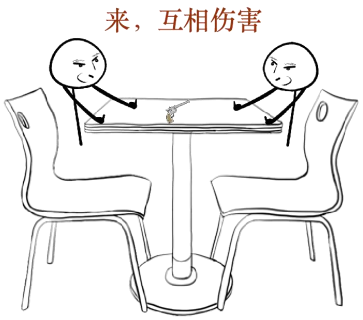
```
    head->prior=list;
    return head;
}

//输出链表的功能函数
void display(line * head){
    line * temp=head;
    //由于是循环链表，所以当遍历指针 temp 指向的下一个节点是 head 时，证明此时已经循环至链表的最后一个节点
    while (temp->next!=head) {
        if (temp->next==NULL) {
            printf("%d\n",temp->data);
        }else{
            printf("%d->",temp->data);
        }
        temp=temp->next;
    }
    //输出循环链表中最后一个节点的值
    printf("%d",temp->data);
}
```

程序输出结果如下：

```
1->2->3
```

数据结构实践项目之俄罗斯轮盘赌小游戏

俄罗斯轮盘赌，想必很多人都听说过，一种残忍的赌博游戏。游戏的道具是一把左轮手枪，其规则也很简单：在左轮手枪中的 6 个弹槽中随意放入一颗或者多颗子弹，在任意旋转转轮之后，关上转轮。游戏的参加者轮流把手枪对着自己，扣动扳机：中枪或是怯场，即为输的一方；坚持到最后的即为胜者。



本节实践项目同轮盘赌类似，游戏规则：n 个参加者排成一个环，每次由主持向左轮手枪中装一颗子弹，并随机转动关上转轮，游戏从第一个人开始，轮流拿枪；中枪者退出赌桌，退出者的下一个人作为第一人开始下一轮游戏。直至最后剩余一个人，即为胜者。要求：模拟轮盘赌的游戏规则，找到游戏的最终胜者。

设计思路

解决类似的问题，使用线性表的顺序存储结构和链式存储结构都能实现，根据游戏规则，在使用链式存储结构时只需使用循环链表即可轻松解决问题。

顺序存储结构模拟轮盘赌

采用顺序存储结构时，同样要在脑海中将数组的首尾进行连接，即当需要从数组中最后一个位置寻找下一个位置时，要能够跳转到数组的第一个位置。（使用取余运算可以解决）

具体实现代码如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef struct gambler{
    int number;
}gambler;

int main(){
    int n;
    int round=1;
    int location=1;
    int shootNum;
    int i,j;
    srand((int)time(0));//设置获得随机数的种子（固定代码，没有这句，随机数是固定不变的）
    printf("输入赌徒的人数：");
    scanf("%d",&n);
    printf("将赌徒依次编号为 1-%d\n",n);
    gambler gamblers[n+1];//存储赌徒编号的数组
    for (i=1;i<=n; i++) {//依次为参加者分配编号
        gamblers[i].number=i;
    }
    //当只剩余一个人时，此场结束
    while (n!=1) {
        printf("第 %d 轮开始，从编号为 %d 的人开始，",round,gamblers[location].number);
        shootNum=rand()%6+1;
        printf("枪在第 %d 次扣动扳机时会响\n",shootNum);
        for (i=location; i<location+shootNum; i++);//找到每轮退出的人的位置（i-1 才是，此处求得的 i 值为下一轮开始的位置）
        i=i%n;//由于参与者排成的是环，所以需要对求得 i 值进行取余处理
```

```
            if (i==1 || i==0) {//当 i=1 或者 i=0 时，实际上指的是位于数组开头和结尾的参与者，需要重新调整 i 的值
                i=n+i;
            }
            printf("编号为 %d 的赌徒退出赌博,剩余赌徒编号依次为：\n",gamblers[i-1].number);
            //使用顺序存储时，如果删除元素，需要将其后序位置的元素进行全部前移
            for (j=i-1; j+1<=n; j++) {
                gamblers[j]=gamblers[j+1];
            }
            n--;//此时参与人数由 n 个人变为 n-1 个人
            for (int k=1; k<=n; k++) {
                printf("%d ",gamblers[k].number);
            }
            printf("\n");
            location=i-1;//location 表示的是下一轮开始的位置
            //同样注意 location 值的范围
            if (location>n) {
                location%=n;
            }
            round++;
        }
        printf("最终胜利的赌徒编号是：%d\n",gamblers[1].number);
}
```

运行结果示例：

```
输入赌徒的人数：5
将赌徒依次编号为 1-5
第 1 轮开始，从编号为 1 的人开始，枪在第 4 次扣动扳机时会响
编号为 4 的赌徒退出赌博,剩余赌徒编号依次为：
1 2 3 5
第 2 轮开始，从编号为 5 的人开始，枪在第 6 次扣动扳机时会响
编号为 1 的赌徒退出赌博,剩余赌徒编号依次为：
2 3 5
第 3 轮开始，从编号为 2 的人开始，枪在第 2 次扣动扳机时会响
编号为 3 的赌徒退出赌博,剩余赌徒编号依次为：
2 5
第 4 轮开始，从编号为 5 的人开始，枪在第 5 次扣动扳机时会响
编号为 5 的赌徒退出赌博,剩余赌徒编号依次为：
2
最终胜利的赌徒编号是：2
```

链式存储结构模拟轮盘赌

采用链式存储结构对于求此类问题是最容易理解的，同时也避免了当参与人数较多时，像顺序存储那样频繁地移动数据。

具体实现代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef enum {false,true} bool;
typedef struct line{
    int No;
    struct line * next;
}line;
//按照赌徒人数，初始化循环链表
void initLine(line ** head,int n){
    *head=(line*)malloc(sizeof(line));
    (*head)->next=NULL;
    (*head)->No=1;
    line * list=*head;
    for (int i=1; i<n; i++) {
        line * body=(line*)malloc(sizeof(line));
        body->next=NULL;
        body->No=i+1;
        list->next=body;
        list=list->next;
    }
    list->next=*head;//将链表成环
}
//输出链表中所有的结点信息
void display(line * head){
    line * temp=head;
    while (temp->next!=head) {
        printf("%d ",temp->No);
        temp=temp->next;
    }
    printf("%d\n",temp->No);
}
```

```
int main() {
    line * head=NULL;
    srand((int)time(0));
    int n,shootNum,round=1;
    printf("输入赌徒人数：");
    scanf("%d",&n);
    initLine(&head,n);
    line* lineNext=head;//用于记录每轮开始的位置
    //仅当链表中只含有一个结点时，即头结点时，退出循环
    while (head->next!=head) {
        printf("第 %d 轮开始，从编号为 %d 的人开始，",round,lineNext->No);
        shootNum=rand()%n+1;
        printf("枪在第 %d 次扣动扳机时会响\n",shootNum);
        line *temp=lineNext;
        //遍历循环链表，找到将要删除结点的上一个结点
        for (int i=1; i<shootNum-1; i++) {
            temp=temp->next;
        }
        //将要删除结点从链表中删除，并释放其占用空间
        printf("编号为 %d 的赌徒退出赌博,剩余赌徒编号依次为：\n",temp->next->No);
        line * del=temp->next;
        temp->next=temp->next->next;
        if (del==head) {
            head=head->next;
        }
        free(del);
        display(head);
        //赋值新一轮开始的位置
        lineNext=temp->next;
        round++;//记录循环次数
    }
    printf("最终胜利的赌徒编号是：%d\n",head->No);
    return 0;
}
```

运行结果示例：

```
输入赌徒人数：5
第 1 轮开始，从编号为 1 的人开始，枪在第 4 次扣动扳机时会响
```

编号为 4 的赌徒退出赌博,剩余赌徒编号依次为:
1 2 3 5
第 2 轮开始，从编号为 5 的人开始，枪在第 3 次扣动扳机时会响
编号为 2 的赌徒退出赌博,剩余赌徒编号依次为:
1 3 5
第 3 轮开始，从编号为 3 的人开始，枪在第 4 次扣动扳机时会响
编号为 3 的赌徒退出赌博,剩余赌徒编号依次为:
1 5
第 4 轮开始，从编号为 5 的人开始，枪在第 4 次扣动扳机时会响
编号为 1 的赌徒退出赌博,剩余赌徒编号依次为:
5
最终胜利的赌徒编号是：5

总结

本节借轮盘赌小游戏，带领大家重新熟悉了线性表的顺序存储结构和链式存储结构，如果你能够根据项目要求自行完成两种结构代码实现的编写工作，恭喜你可以顺利进入下面章节的学习。

数据结构实践项目之进制转换器

进制转换器项目要求：用户提供需要转换的数据和该数据的进制，以及要转换的进制，进制转换器提供给用户最终的正确转换的结果。

转换器实例

例如，用户提供了一个十进制数：10，要求将此数据以二进制形式转换，则通过进制转换器转换的最终结果应该：1010。

提示：此进制转换器可以在 2-36 进制之间对数据进行任意转换。各进制中对应的数字如下表：

对应数字	0 1 2 · · · 8 9	A B C · · · · X Y Z
进制数	0 1 2 · · · 8 9	10 11 12 · · · · 33 34 35

设计思路

当用户给定 2 - 36 进制中的任意一进制数时，最简单的方法是使用顺序存储结构进行存储，即使用字符串数组存储。

转化时，最直接的思路就是先将该数转化为十进制数据，然后再由十进制转化成要求的进制数，最终的结果用栈结构存储（先进后出），这样最终显示给用户的是正常的数据。

实现代码

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

```c
int top=-1;//top 变量时刻表示栈顶元素所在位置
void push(char * a,char elem){
    a[++top]=elem;
}
void pop(char * a){
    if (top==-1) {
        return ;
    }
    //输出时要按照正确的格式显示给用户
    if (a[top]>=10) {
        printf("%c",a[top]+55);
    }else{
        printf("%d",a[top]);
    }
    top--;
}
//将各进制数转换成十进制数
int scaleFun(char * data,int system){
    int k=(int)strlen(data)-1;
    int system_10_data=0;
    int i;
    for (i=k; i>=0; i--) {
        int temp;
        if (data[i]>=48 && data[i]<=57) {
            temp=data[i]-48;
        }else{
            temp=data[i]-55;
        }
        system_10_data+=temp*pow(system, k-i);
    }
    return system_10_data;
}
int main() {
    char data[100];
    printf("进制转换器，请输入原数据的进制（2-36）:");
    int system;
    scanf("%d",&system);
```

```
    getchar();
    printf("请输入要转换的数据：");
    scanf("%s",data);
    getchar();
    int system_10_data=scaleFun(data, system);
    printf("请输入转换后的数据的进制：");
    int newSystem;
    scanf("%d",&newSystem);
    getchar();
    while (system_10_data/newSystem) {
        push(data,system_10_data%newSystem );
        system_10_data/=newSystem;
    }
    push(data,system_10_data%newSystem);
    printf("转换后的结果为：\n");
    while (top!=-1) {
        pop(data);
    }
}
```

运行结果：

```
进制转换器，请输入原数据的进制（2-36）:10
请输入要转换的数据：100
请输入转换后的数据的进制：16
转换后的结果为：
64
```

括号匹配算法及 C 语言实现

在编写代码的时候，经常会用到两种括号：圆括号 “()” 和大括号 “{}” 。不管使用哪种括号，程序编译没有问题的其中一个重要因素就是所使用的括号是否能够匹配上.

在编写程序时，括号可以嵌套，即： “({()})” 这种形式，但 “({)” 或者 “({}” 都不符合要求。

括号匹配项目要求：给出任意搭配的括号，判断是否匹配。

设计思路

编写程序判断括号匹配问题的时候，使用栈结构会很容易：

- 如果碰到的是左圆括号或者左大括号，直接压栈；

- 如果碰到的是右圆括号或者右大括号，就直接和栈顶元素配对：如果匹配，栈顶元素弹栈；反之，括号不匹配；

实现代码

```
#include <stdio.h>
#include <string.h>
int top=-1;//top 变量时刻表示栈顶元素所在位置
void push(char * a,int elem){
    a[++top]=elem;
}
void pop(char* a){
    if (top==-1) {
        return ;
    }
    top--;
}
char visit(char * a){
    //调取栈顶元素，不等于弹栈，如果栈为空，为使程序不发生错误，返回空字符
    if (top!=-1) {
        return a[top];
    }else{
        return ' ';
    }
}
int main() {
    char a[30];
    char bracket[100];
    printf("请输入括号序列：");
    scanf("%s",bracket);
    getchar();
    int length=(int)strlen(bracket);
    for (int i=0; i<length; i++) {
        //如果是左括号，直接压栈
        if (bracket[i]=='('||bracket[i]=='{') {
            push(a, bracket[i]);
        }else{
            //如果是右边括号，判断与栈顶元素是否匹配，如果匹配，栈顶元素弹栈，程序继续运行；否则，发现括号不匹配，
输出结果直接退出
```

```
            if (bracket[i]==')') {
                if (visit(a)=='(') {
                    pop(a);
                }else{
                    printf("括号不匹配");
                    return 0;
                }
            }else{
                if (visit(a)=='{') {
                    pop(a);
                }else{
                    printf("括号不匹配");
                    return 0;
                }
            }
        }
    }
    //如果所有括号匹配完成，栈内为空，说明所有括号全部匹配成功
    if (top!=-1) {
        printf("括号不匹配");
    }else{
        printf("括号匹配");
    }
}
```

运行结果：

```
请输入括号序列：{}(){
括号不匹配
```

数据结构实践项目之变态的停车场管理系统

实践是检验真理的唯一标准，学习也是如此。本章对栈和队列做了详细的讲解，为了让大家能够学以致用，特推出一个项目供大家练习（包含了本章所有的重要知识点）。

本项目比较烧脑，要求对栈和队列有一定深度的了解，虽有完整代码供大家参考，但是建议先自行完成，然后参照本节给出的完整代码。

项目简介

设停车场是一个可以停放 n 辆汽车的南北方向的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达时间的先后顺序，依次由北向南排列（大门在最南端，最先到达的第一辆车停放在车场的最北端）。

若车场内已停满 n 辆车，那么后来的车只能在门外的便道上等候，一旦有车开走，则排在便道上的第一辆车即可开入；当停车场内某辆车要离开时，在它之后进入的车辆必须先退出车场为它让路，待该辆车开出大门外，其它车辆再按原次序进入车场，每辆停放在车场的车在它离开停车场时必须按它停留的时间长短交纳费用。

项目要求：试为停车场编制按上述要求进行管理的模拟程序。要求程序输出每辆车到达后的停车位置（停车场或便道上），以及某辆车离开停车场时应缴纳的费用和它在停车场内停留的时间。

设计思路

停车场的管理流程如下：

1. 当车辆要进入停车场时，检查停车场是否已满，如果未满则车辆进入停车场；如果停车场已满，则车辆进入便道等候。

2. 当车辆要求出栈时，先让在它之后进入停车场的车辆退出停车场为它让路，再让该车退出停车场，让路的所有车辆再按其原来进入停车场的次序进入停车场。之后，再检查在便道上是否有车等候，有车则让最先等待的那辆车进入停车场。

项目中设计到的数据结构

1. 由于停车场只有一个大门，当停车场内某辆车要离开时，在它之后进入的车辆必须先退出车场为它让路，先进停车场的后退出，后进车场的先退出，符合栈的“后进先出，先进后出”的操作特点，因此，可以用一个栈来模拟停车场。

2. 而当停车场满后，继续来到的其它车辆只能停在便道上，根据便道停车的特点，先排队的车辆先离开便道进入停车场，符合队列的“先进先出，后进后出”的操作特点，因此，可以用一个队列来模拟便道。

3. 排在停车场中间的车辆可以提出离开停车场，并且停车场内在要离开的车辆之后到达的车辆都必须先离开停车场为它让路，然后这些车辆依原来到达停车场的次序进入停车场，因此在前面已设的一个栈和一个队列的基础上，还需要有一个地方保存为了让路离开停车场的车辆，由于先退出停车场的后进入停车场，所以很显然保存让路车辆的场地也应该用一个栈来模拟。

因此，本题求解过程中需用到两个栈和一个队列。栈和队列都既可以用顺序结构实现，也可以用链式结构实现。

程序清单

以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入的输入数据序列进行模拟管理。

每一组输入数据包括三个数据项：汽车“到达”或“离去”信息、汽车牌照号码以及到达或离去的时刻。对每一组输入数据进行操作后的输出信息为：若是车辆到达，则输出汽车在停车场内或便道上的停车位置；若是车辆离去，则输出汽车在停车场内停留的时间和应交纳的费用（在便道上停留的时间不收费，停车场费用按照 1 小时 1.5 元）。

实现代码

```
#include <stdio.h>

#define MAX 3//模拟停车场最多可停的车辆数

//车的必要信息的结构体

typedef struct{
```

```
    int number;
    int arrive_time;
}zanInode;
//进入停车场的处理函数
int push(zanInode * park,int *parktop,zanInode car){
    //如果停车场已满，该车需进入便道等待（先返回 -1 ，在主程序中处理）
    if ((*parktop)>=MAX) {
        printf("停车场已停满！需停到便道上.\n");
        return -1;
    }else{//否则将该车入栈，同时进行输出
        park[(*parktop)]=car;
        printf("该车在停车场的第 %d 的位置上\n",(*parktop)+1);
        (*parktop)++;
        return 1;
    }
}
//车从停车场中退出的处理函数
zanInode pop(zanInode *park,int *parktop,int carnumber,zanInode * location,int *locationtop){
    int i;
    //由于函数本身的返回值需要返回一辆车，所以需要先初始化一个
    zanInode thecar;
    thecar.number=-1;
    thecar.arrive_time=-1;
    //在停车场中找到要出去的车
    for (i=-1; i<(*parktop); i++) {
        if (park[i].number==carnumber) {
            break;
        }
    }
    //如果遍历至最后一辆车，还没有找到，证明停车场中没有这辆车
    if (i==(*parktop)) {
        printf("停车场中没有该车\n");
    }else{//就将该车移出停车场
        //首先将在该车后进入停车场的车全部挪至另一个栈中
        while ((*parktop)>i+1) {
            (*parktop)--;
            location[*locationtop]=park[*parktop];
```

```
            (*locationtop)++;
        }
        //通过以上的循环，可以上该车后的左右车辆全部移开，但是由于该车也要出栈，所以栈顶指针需要下移一个位置，当车进栈时，就直接覆盖掉了
        (*parktop)--;
        thecar=park[*parktop];
        //该车出栈后，还要将之前出栈的所有车，在全部进栈
        while ((*locationtop)>0) {
            (*locationtop)--;
            park[*parktop]=location[*locationtop];
            (*parktop)++;
        }
    }
    return thecar;
}

int main(int argc, const char * argv[]) {
    //停车场的栈
    zanInode park[MAX];
    int parktop=0;//栈顶指针

    //辅助停车场进行挪车的栈
    zanInode location[MAX];
    int locationtop=0;//栈顶指针

    //模拟便道的队列
    zanInode accessroad[100];
    int front,rear;//队头和队尾指针
    front=rear=0;
    char order;//进出停车场的输入命令
    int carNumber;//车牌号
    int arriveTime;//到停车场的时间
    int leaveTime;//离开停车场的时间
    int time;//车在停车场中逗留的时间

    zanInode car;
    printf("有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:\n");
```

```
    while (scanf("%c",&order)) {
        if (order=='#') {
            break;
        }
        switch (order) {
            case 'A':
                printf("登记车牌号(车牌号不能为 -1)及车辆到达时间（按小时为准）：\n");
                scanf("%d %d",&carNumber,&arriveTime);
                car.number=carNumber;
                car.arrive_time=arriveTime;
                //当有车想要进入停车场时，首先试图将该车进入停车场
                int result=push(park, &parktop, car);
                //如果返回值为 -1 ，证明停车场已满，需要停在便道中
                if (result==-1) {//停在便道上
                    accessroad[rear]=car;
                    printf("该车在便道的第 %d 的位置上\n",rear+1-front);
                    rear++;
                }
                break;
            case 'D':
                printf("出停车场的车的车牌号以及离开的时间：\n");
                scanf("%d %d",&carNumber,&leaveTime);
                //当有车需要出停车场时，调用出栈函数
                car=pop(park, &parktop, carNumber, location, &locationtop);
                //如果返回的车的车牌号为-1 ，表明在停车场中没有查找到要查找的车
                if (car.number!=-1) {
                    //停留时间，等于进停车场的时间-
                    time=leaveTime-car.arrive_time;
                    printf("该车停留的时间为：%d 小时,应缴费用为：%f 元\n",time,time*1.5);
                    //一旦有车离开停车场，则在便道中先等待的车就可以进入，进入时需设定车进入的时间
                    if (front!=rear) {
                        car=accessroad[front];
                        printf("在便道上第 1 的位置上，车牌号为：%d 的车进停车场的时间为：\n",car.number);
                        scanf("%d",&car.arrive_time);
                        park[parktop]=car;
                        front++;
                        parktop++;
```

```
                    }else{
                        printf("便道上没有等待车辆，停车场不满！\n");
                    }
                }
                break;
            default:
                break;
        }
        printf("\n 有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:\n");
        scanf("%*[^\n]"); scanf("%*c");//清空缓冲区
    }
    return 0;
}
```

运行结果:

```
有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
A
登记车牌号(车牌号不能为 -1)及车辆到达时间（按小时为准）:
633 6
该车在停车场的第 1 的位置上

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
A
登记车牌号(车牌号不能为 -1)及车辆到达时间（按小时为准）:
634 7
该车在停车场的第 2 的位置上

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
A
登记车牌号(车牌号不能为 -1)及车辆到达时间（按小时为准）:
635 8
该车在停车场的第 3 的位置上

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
A
登记车牌号(车牌号不能为 -1)及车辆到达时间（按小时为准）:
636 9
停车场已停满！需停到便道上.
该车在便道的第 1 的位置上

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
D
```

```
出停车场的车的车牌号以及离开的时间:
633 10
该车停留的时间为：4 小时,应缴费用为：6.000000 元
在便道上第 1 的位置上，车牌号为：636 的车进停车场的时间为:
10

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
D
出停车场的车的车牌号以及离开的时间:
634 10
该车停留的时间为：3 小时,应缴费用为：4.500000 元
便道上没有等待车辆，停车场不满!

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
A
登记车牌号(车牌号不能为 -1)及车辆到达时间（按小时为准）:
637 11
该车在停车场的第 3 的位置上

有车辆进入停车场（A）;有车辆出停车场(D);程序停止（#）:
#
```

扑克牌游戏及 C 语言实现

小时候在刚开始接触扑克牌的时候，最初学会的扑克游戏就是类似于“推小车”这样的无脑游戏，本节带领大家使用学过的知识编写推小车卡牌游戏。

“推小车”扑克牌游戏适合 2-3 个人玩，游戏规则也超级简单：将一副扑克牌平均分成两份，每人拿一份，每个人手中的扑克牌全部反面朝上，叠成一摞。游戏进行时，每个人轮流拿出第一张扑克牌放到桌上，将其排成一竖行。如果打出的牌与桌上某张牌的数字（红桃 5 和黑桃 5 在此游戏中相等）相等，即可将两张相同的牌以及两张中间所夹的所有的牌全部取走，每次取走的一小摞牌都必须放到自己本摞的下面。

游戏过程中，一旦有人手中没有牌，则宣布另一人获胜，同时游戏结束。

设计思路

假设模拟两个人进行该扑克牌游戏。每个人在游戏过程中都是不断地从自己这一摞扑克牌的最上方去取牌，放到桌子上；当发现自己的牌同桌子上的牌相等时，将赢得的牌依次放在自己扑克牌的下方。这是典型的队列的“先进先出”。

而对于桌子而言，就相当于是一个栈。每次放到桌子上的扑克牌，都相当于进栈；当有相同的扑克牌时，相同的扑克牌连通之间的所有的扑克牌则依次出栈。

所以，模拟该扑克牌游戏需要同时使用 2 个队列和 1 个栈。

实现代码

```
#include <stdio.h>
#include <stdlib.h>
struct queue
{
    int data[1000];
    int head;
    int tail;
};
struct stack
{
    int data[10];
    int top;
};
void showCard(struct queue *q,int *book,struct stack *s){
    int t=(*q).data[(*q).head]; //打出一张牌，即从队列 q 的队头取元素（此时还不往桌子的栈里放）
    //判断取出的这张牌是否会赢牌
    if(book[t]==0){ //若不赢牌，只需放到桌子上入栈即可
        (*q).head++;//由于此时牌已经打出，所以队列的队头需要前进
        (*s).top++;
        (*s).data[(*s).top]=t; //再把打出的牌放到桌上，即入栈
        book[t]=1; //标记桌上现在已经有牌面为 t 的牌
    }
    else{
        (*q).head++;//由于此时已经打出去一张牌，所以队头需要 +1
        (*q).data[(*q).tail]=t;//将打出的牌放到手中牌的最后（再入队）
        (*q).tail++;
        //把桌子上赢得的牌依次放到手中牌的最后（依次出栈在入队列的过程）
        while((*s).data[(*s).top]!=t){
            book[(*s).data[(*s).top]]=0;//取消对该牌号的标记
            (*q).data[(*q).tail]=(*s).data[(*s).top];//依次放入队尾
            (*q).tail++;
            (*s).top--;
        }
        //最后别忘了将最后一张相等的牌取出放入队列
        book[(*s).data[(*s).top]]=0;
```

```
        (*q).data[(*q).tail]=(*s).data[(*s).top];
        (*q).tail++;
        (*s).top--;
    }
}
int main()
{
    struct queue q1,q2;//两个队列，分别模拟两个人，假设分别为小王和小李
    struct stack s;//栈，模拟桌子
    int book[14];//为了便于判断桌子上的牌是否有相同的，增加一个数组用于判断
    int i;
    //初始化队列
    q1.head=0; q1.tail=0;
    q2.head=0; q2.tail=0;
    //初始化栈
    s.top=-1;
    //初始化用来标记的数组
    for(i=0;i<=13;i++)
        book[i]=0;
    //假设初期每个人手中仅有 6 张牌，每个人拥有的牌都是随机的，但都在 1-13 之间
    for(i=1;i<=6;i++){
        q1.data[q1.tail]=rand()%13+1;
        q1.tail++;
    }
    for(i=1;i<=6;i++){
        q2.data[q2.tail]=rand()%13+1;
        q2.tail++;
    }
    //仅当其中一个人没有牌时，游戏结束
    while(q1.head<q1.tail && q2.head<q2.tail ){        showCard(&q2, book, &s);//小李出牌
        showCard(&q1, book, &s);//小王出牌
    }
    //游戏结束时，输出最后的赢家以及手中剩余的牌数
    if(q2.head==q2.tail){
        printf("小李赢\n");
        printf("手中还有：%d 张牌",q1.tail-q1.head);
    }
```

```
    else{
        printf("小王赢\n");
        printf("手中还有：%d 张牌",q2.tail-q2.head);
    }
    return 0;
}
```

运行结果：

```
小王赢
手中还有：7 张牌
```

KMP 算法（快速模式匹配算法）C 语言详解

快速模式匹配算法，简称 KMP 算法，是在 BF 算法基础上改进得到的算法。学习 BF 算法我们知道，该算法的实现过程就是 "傻瓜式" 地用模式串（假定为子串的串）与主串中的字符一一匹配，算法执行效率不高。

KMP 算法不同，它的实现过程接近人为进行模式匹配的过程。例如，对主串 A（"ABCABCE"）和模式串 B（"ABCE"）进行模式匹配，如果人为去判断，仅需匹配两次。

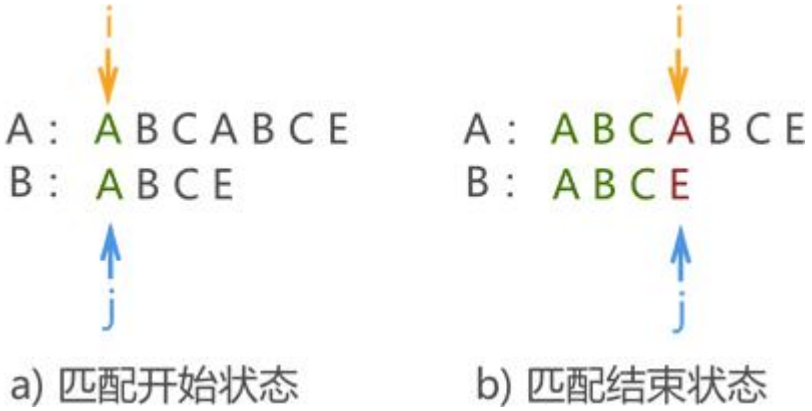


图 1 第一次人为模式匹配

第一次如图 1 所示，最终匹配失败。但在本次匹配过程中，我们可以获得一些信息，模式串中 "ABC" 都和主串对应的字符相同，但模式串中字符 'A' 与 'B' 和 'C' 不同。

因此进行下次模式匹配时，没有必要让串 B 中的 'A' 与主串中的字符 'B' 和 'C' 一一匹配（它们绝不可能相同），而是直接去匹配失败位置处的字符 'A'，如图 2 所示：

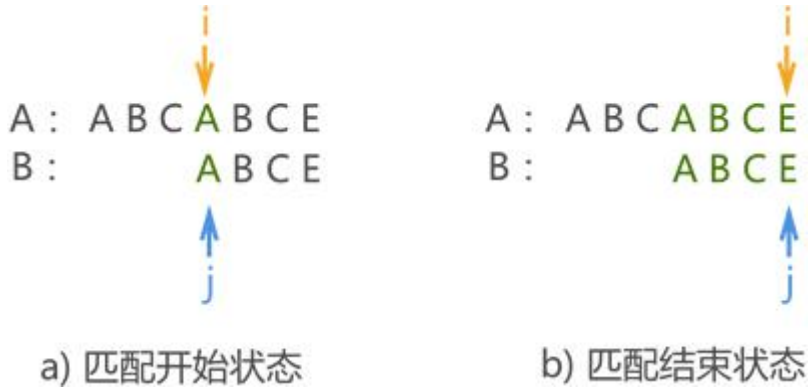


图 2 第二次人为模式匹配

至此，匹配成功。若使用 BF 算法，则此模式匹配过程需要进行 4 次。

由此可以看出，每次匹配失败后模式串移动的距离不一定是 1，某些情况下一次可移动多个位置，这就是 KMP 模式匹配算法。

那么，如何判断匹配失败后模式串向后移动的距离呢?

模式串移动距离的判断

每次模式匹配失败后，计算模式串向后移动的距离是 KMP 算法中的核心部分。

其实，匹配失败后模式串移动的距离和主串没有关系，只与模式串本身有关系。

例如，我们将前面的模式串 B 改为 "ABCAE"，则在第一次模式匹配失败，由于匹配失败位置模式串中字符 'E' 前面有两个字符 'A'，因此，第二次模式匹配应改为如图 3 所示的位置:

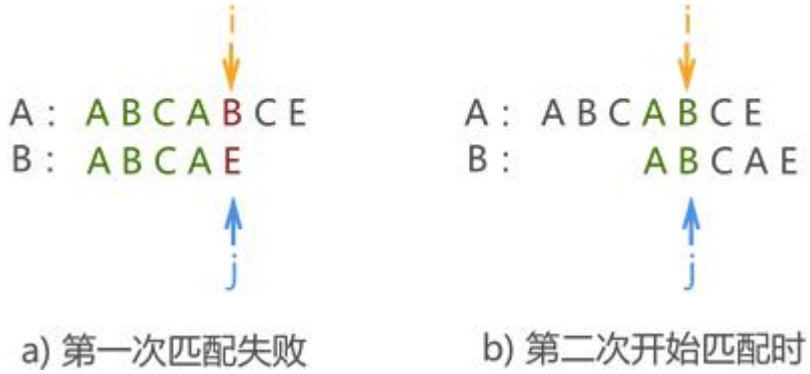


图 3 模式匹配过程示意图

结合图 1、图 2 和图 3 不难看出，模式串移动的距离只和自身有关系，和主串无关。换句话说，不论主串如何变换，只要给定模式串，则匹配失败后移动的距离就已经确定了。

不仅如此，模式串中任何一个字符都可能导致匹配失败，因此串中每个字符都应该对应一个数字，用来表示匹配失败后模式串移动的距离。

注意，这里要转换一下思想，模式串向后移动等价于指针 j 前移，如图 4 中的 a) 和 b)。换句话说，模式串后移相当于对指针 j 重定位。

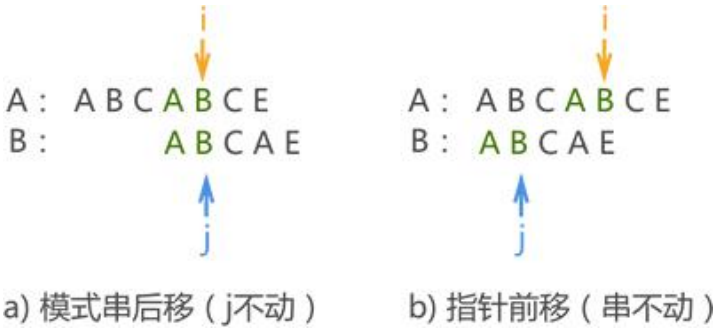


图 4 模式串后移等价于 j 前移

因此，我们可以给每个模式串配备一个数组（例如 next[]），用于存储模式串中每个字符对应指针 j 重定向的位置（也就是存储模式串的数组下标），比如 j=3，则该字符匹配失败后指针 j 指向模式串中第 3 个字符。

模式串中各字符对应 next 值的计算方式是，取该字符前面的字符串（不包含自己），其前缀字符串和后缀字符串相同字符的最大个数再 +1 就是该字符对应的 next 值。

前缀字符串指的是位于模式串起始位置的字符串，例如模式串 "ABCD"，则 "A"、"AB"、"ABC" 以及 "ABCD" 都属于前缀字符串；后缀字符串指的是位于串结尾处的字符串，还拿模式串 "ABCD" 来说，"D"、"CD"、"BCD" 和 "ABCD" 为后缀字符串。

注意，模式串中第一个字符对应的值为 0，第二个字符对应 1 ，这是固定不变的。因此，图 3 的模式串 "ABCAE" 中，各字符对应的 next 值如图 5 所示：

图 5 模式串对应的 next 数组

从图 5 中的数据可以看出，当字符 'E' 匹配失败时，指针 j 指向模式串数组中第 2 个字符，即 'B'，同之前讲解的图 3 不谋而合。

以上所讲 next 数组的实现方式是为了让大家对此数组的功能有一个初步的认识。接下来学习如何用编程的思想实现 next 数组。编程实现 next 数组要解决的主要问题依然是 "如何计算每个字符前面前缀字符串和后缀字符串相同的个数"。

仔细观察图 5，为什么字符 'C' 对应的 next 值为 1？因为字符串 "AB" 前缀字符串和后缀字符串相等个数为 0，0 + 1 = 1。那么，为什么字符 'E' 的 next 值为 2？因为紧挨着该字符之前的 'A' 与模式串开头字符 'A' 相等，1 + 1 = 2。

如果图 5 中模式串为 "ABCABE"，则对应 next 数组应为 [0,1,1,1,2,3]，为什么字符 'E' 的 next 值是 3 ？因为紧挨着该字符前面的 "AB" 与开头的 "AB" 相等，2 + 1 =3。

因此，我们可以设计这样一个算法，刚开始时令 j 指向模式串中第 1 个字符，i 指向第 2 个字符。接下来，对每个字符做如下操作：

如果 i 和 j 指向的字符相等，则 i 后面第一个字符的 next 值为 j+1，同时 i 和 j 做自加 1 操作，为求下一个字符的 next 值做准备，如图 6 所示：

图 6 i 和 j 指向字符相等

上图中可以看到，字符 'a' 的 next 值为 j +1 = 2，同时 i 和 j 都做了加 1 操作。当计算字符 'C' 的 next 值时，还是判断 i 和 j 指向的字符是否相等，显然相等，因此令该字符串的 next 值为 j + 1 = 3，同时 i 和 j 自加 1（此次 next 值的计算使用了上一次 j 的值）。如图 7 所示：

图 7 i 和 j 指向字符仍相等

如上图所示，计算字符 'd' 的 next 时，i 和 j 指向的字符不相等，这表明最长的前缀字符串 "aaa" 和后缀字符串 "aac" 不相等，接下来要判断次长的前缀字符串 "aa" 和后缀字符串 "ac" 是否相等，这一步的实现可以用 j = next[j] 来实现，如图 8 所示：

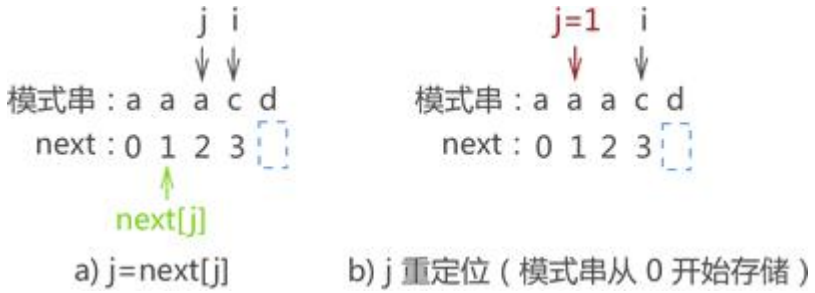


图 8 执行 j=next[j] 操作

从上图可以看到，i 和 j 指向的字符又不相同，因此继续做 j = next[j] 的操作，如图 9 所示：

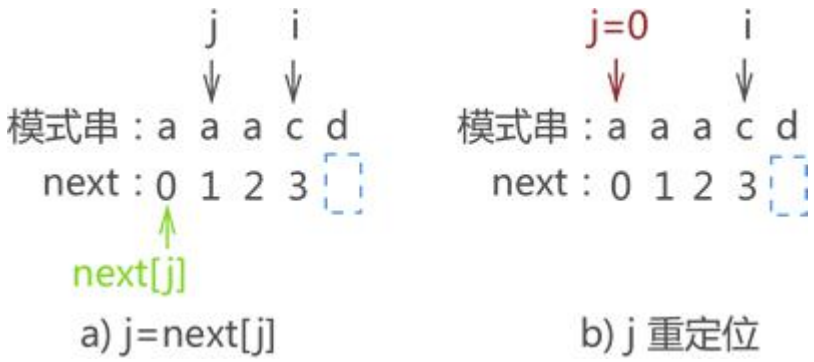


图 9 继续执行 j=next[j] 的操作

可以看到，j = 0 表明字符 'd' 前的前缀字符串和后缀字符串相同个数为 0，因此如果字符 'd' 导致了模式匹配失败，则模式串移动的距离只能是 1。

这里给出使用上述思想实现 next 数组的 C 语言代码：

```
void Next(char*T,int *next){
    next[1]=0;
    next[2]=1;
    int i=2;
    int j=1;
    while (i<strlen(T)) {
        if (j==0||T[i-1]==T[j-1]) {
            i++;
            j++;
            next[i]=j;
        }else{
            j=next[j];
        }
    }
}
```

代码中 j=next[j] 的运用可以这样理解，每个字符对应的 next 值都可以表示该字符前 "同后缀字符串相同的前缀字符串最后一个字符所在的位置"，因此在每次匹配失败后，都可以轻松找到次长前缀字符串的最后一个字符与该字符进行比较。

Next 函数的缺陷

图 10 Next 函数的缺陷

例如，在图 10a) 中，当匹配失败时，Next 函数会由图 10b) 开始继续进行模式匹配，但是从图中可以看到，这样做是没有必要的，纯属浪费时间。

出现这种多余的操作，问题在当 T[i-1]==T[j-1] 成立时，没有继续对 i++ 和 j++ 后的 T[i-1] 和 T[j-1] 的值做判断。改进后的 Next 函数如下所示：

```
void Next(char*T,int *next){
    next[1]=0;
    next[2]=1;
    int i=2;
    int j=1;
    while (i<strlen(T)) {
        if (j==0||T[i-1]==T[j-1]) {
            i++;
            j++;
            if (T[i-1]!=T[j-1]) {
                next[i]=j;
            }
            else{
                next[i]=next[j];
            }
        }else{
            j=next[j];
        }
    }
}
```

使用精简过后的 next 数组在解决例如模式串为 "aaaaaaab" 这类的问题上，会大大提高效率，如图 11 所示，精简前为 next1，精简后为 next2：

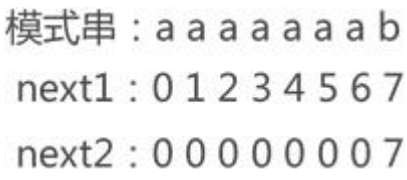


图 11 改进后的 Next 函数

KMP 算法的实现

假设主串 A 为 "ababcabcacbab"，模式串 B 为 "abcac"，则 KMP 算法执行过程为：

- 第一次匹配如图 12 所示，匹配结果失败，指针 j 移动至 next[j] 的位置；

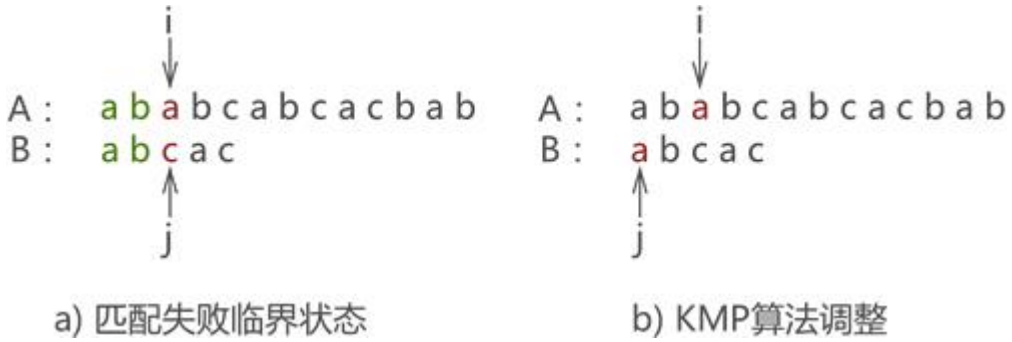


图 12 第一次匹配示意图

- 第二次匹配如图 13 所示，匹配结果失败，依旧执行 j=next[j] 操作：

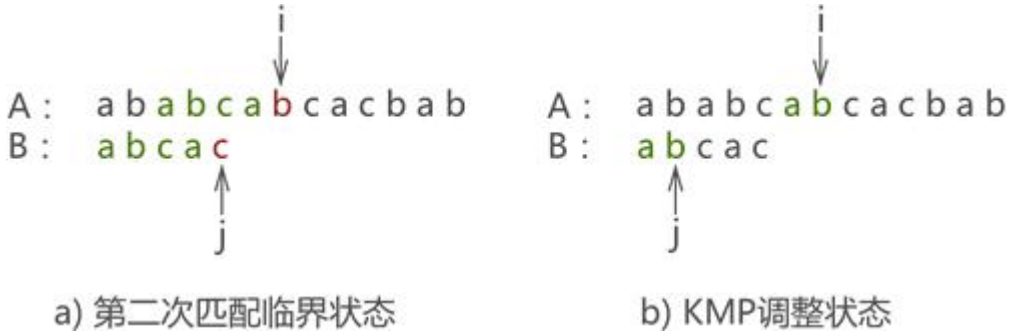


图 13 第二次匹配示意图

- 第三次匹配成功，如图 14 所示：

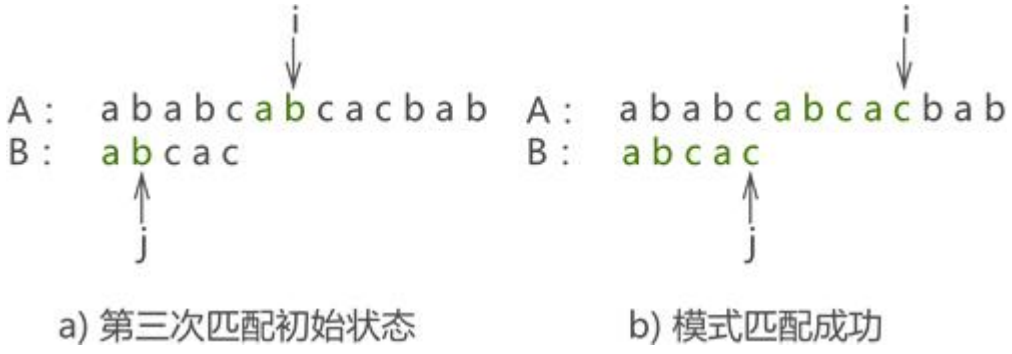


图 14 第三次匹配示意图

很明显，使用 KMP 算法只需匹配 3 次，而同样的问题使用 BF 算法则需匹配 6 次才能完成。

KMP 算法的完整 C 语言实现代码为：

```
#include <stdio.h>
#include <string.h>
void Next(char*T,int *next){
    int i=1;
    next[1]=0;
    int j=0;
    while (i<strlen(T)) {
        if (j==0||T[i-1]==T[j-1]) {
            i++;
            j++;
            next[i]=j;
        }else{
            j=next[j];
        }
    }
}
int KMP(char * S,char * T){
    int next[10];
    Next(T,next);//根据模式串 T,初始化 next 数组
    int i=1;
    int j=1;
    while (i<=strlen(S)&&j<=strlen(T)) {
        //j==0:代表模式串的第一个字符就和当前测试的字符不相等；S[i-1]==T[j-1],如果对应位置字符相等，两种情况下，指向当前测试的两个指针下标 i 和 j 都向后移
        if (j==0 || S[i-1]==T[j-1]) {
            i++;
            j++;
        }
        else{
            j=next[j];//如果测试的两个字符不相等，i 不动，j 变为当前测试字符串的 next 值
        }
    }
    if (j>strlen(T)) {//如果条件为真，说明匹配成功
        return i-(int)strlen(T);
```

```
    }
    return -1;
}
int main() {
    int i=KMP("ababcabcacbab","abcac");
    printf("%d",i);
    return 0;
}
```

运行结果为：

```
6
```

数据结构实践项目之字符过滤系统

字符过滤系统简介：由用户给定一串字符文本和要查询的字符或者字符串，要求系统能够自动检测出该字符或字符串在文本中的位置，同时系统中需设有全部替换功能，必要时将文本中的所有查询到的字符替换成指定字符。

项目实例

例如在字符串文本 “abcabc” 中检测是否含有字符 ‘a’ 时，字符过滤系统应反馈给用户该字符存储于文本中的位置，即 1 和 4，同时询问用户是否将所有的字符 ‘a’ 替换成其它指定字符，假设用户指定字符 ‘a’ 全部转换成字符 ‘d’，系统应将替换后的字符串 “dbcdbc” 反馈给用户。

设计思路

字符过滤系统的设计，其主要功能的实现主要分为两部分：字符串的模式匹配和字符串的替换。有关字符串的模式匹配算法，可以选择 BF 算法或是 KMP 算法，

和之前所接触的算法不同的是，系统是要将文本中所有与指定字符相同的都筛选出来，而不是找出一个就结束。不过换汤不换药，只要在前面的学习中领会了模式匹配的思想，就可以轻松解决该问题。

在实现字符串的替换功能时，没有成型的算法，需要自己设计方案去编码实现。本节有提供了一种实现的思路，建议大家先自己思考，试着独立解决，培养自己的编程解题思路，若实在不会了再借鉴本节给的思路。

完整实现代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 1000
//KMP 算法中的 next 数组
void Next(char*T,int *next){
```

```
    int i=1;
    next[1]=0;
    int j=0;
    while (i<strlen(T)) {
        if (j==0||T[i-1]==T[j-1]) {
            i++;
            j++;
            if (T[i-1]!=T[j-1]) {
                next[i]=j;
            }
            else{
                next[i]=next[j];
            }
        }else{
            j=next[j];
        }
    }
}
//KMP 算法(快速模式匹配)实现函数
void KMP(char * S,char * T,int (*a)[],int *number){
    int next[10];
    Next(T,next);//根据模式串 T,初始化 next 数组
    int i=1;
    int j=1;
    *number=0;
    while (i<=strlen(S)) {
        if (j==0 || S[i-1]==T[j-1]) {
            i++;
            j++;
        }else{
            j=next[j];
        }
        if (j>strlen(T)) {
            (*number)++;
            (*a)[(*number)]=i-(int)strlen(T);
            j=0;
            i--;
```

```
            }
        }
}
//字符串替换算法，oldData：原字符串，selectData 要替换掉的字符串，数组 a 存储的是需替换字符在原字符串中的首地址，number 表示数组 a 的长度， newData 用于存储新字符串， replace 为替换的新字符
void replaceData(char * oldData,int *a,int number,char *replace,char * selectData,char * newData){
    int order=0;//表示 newData 存储字符的位置
    int begin=0;
    for (int i=1; i<=number; i++) {
        //先将替换位置之前的字符完整的复制到新字符串数组中
        for (int j=begin; j<a[i]-1; j++) {
            newData[order]=oldData[j];
            order++;
        }
        //替换字符，用新字符代替
        for (int k=0;k<strlen(replace);k++) {
            newData[order]=replace[k];
            order++;
        }
        //代替完成后，计算出原字符串中隔过该字符串的下一个起始位置
        begin=a[i]+(int)strlen(selectData)-1;
    }
    //要替换位置全部替换完成后，检测是否还有后续字符，若有直接复制
    while(begin<strlen(oldData)) {
        newData[order]=oldData[begin];
        order++;
        begin++;
    }
}
int main() {
    while (1) {
        printf("字符过滤检测系统启动(S),关闭(O)，请选择：\n");
        char s;
        char oldData[SIZE];
        char selectData[SIZE];
        char replace[SIZE];
        char judge;
```

```
        char *newData=(char*)malloc(SIZE*sizeof(char));
        FILE * out;
        scanf("%c",&s);
        getchar();
        if (s=='O') {
            break;
        }else{
            printf("请输入原字符串：\n");
            scanf("%[^\n]",oldData);
            getchar();//用于承接缓冲区冲的换行符
            printf("输入要查找的字符或字符串：\n");
            while (scanf("%s",selectData)) {
                getchar();
                int a[SIZE],number;
                KMP(oldData,selectData,&a,&number);
                if (number==0){
                    printf("未检测到文章中有该字符串！是否重新输入(Y/N)：\n");
                    scanf("%c",&judge);
                    getchar();
                    if (judge=='N') {
                        break;
                    }else{
                        printf("输入要查找的字符或字符串：\n");
                    }
                }else{
                    printf("系统检测到该字符在原字符串中出现 %d 次，起始位置依次是：\n",number);
                    for (int i=1; i<=number; i++) {
                        printf("%d ",a[i]);
                    }
                    printf("\n");
                    printf("是否使用新字符串替换所有的 %s(Y/N)\n",selectData);
                    scanf("%c",&judge);
                    getchar();
                    if (judge=='Y') {
                        printf("请输入用于替换的字符串：\n");
                        scanf("%[^\n]",replace);
                        getchar();
```

```
                replaceData(oldData,a,number,replace,selectData,newData);
                printf("新生成的字符串为: %s\n",newData);
                if((out=fopen("new.txt", "wr"))==NULL){
                    printf("新生成的字符串为%s，写入文件失败",newData);
                }
                if(fputs(newData, out)){
                    printf("已将新字符串写入 new.txt 文件中\n");
                }
                free(newData);
                fclose(out);
            }
            break;
        }
    }
  }
  return 0;
}
```

运行结果：
字符过滤检测系统启动(S),关闭(O)，请选择：
S
请输入原字符串：
hello world!
输入要查找的字符或字符串：
!
系统检测到该字符在原字符串中出现 1 次，起始位置依次是：
12
是否使用新字符串替换所有的 !(Y/N)
Y
请输入用于替换的字符串：
,I love you!
新生成的字符串为: hello world,I love you!
已将新字符串写入 new.txt 文件中
字符过滤检测系统启动(S),关闭(O)，请选择：
O

稀疏矩阵的快速转置（C 语言）算法详解

《稀疏矩阵的转置算法》一节介绍了实现矩阵转置的普通算法，该算法的时间复杂度为 $O(n^2)$。本节给大家介绍一种实现矩阵转置更高效的算法，通常称为稀疏矩阵的快速转置算法。

我们知道，稀疏矩阵的转置需要经历以下 3 步：

1. 将矩阵的行数和列数互换；
2. 将三元组表（存储矩阵）中的 i 列和 j 列互换，实现矩阵的转置；
3. 以 j 列为序，重新排列三元组表中存储各三元组的先后顺序；

稀疏矩阵快速转置算法和普通算法的区别仅在于第 3 步，快速转置能够做到遍历一次三元组表即可完成第 3 步的工作。

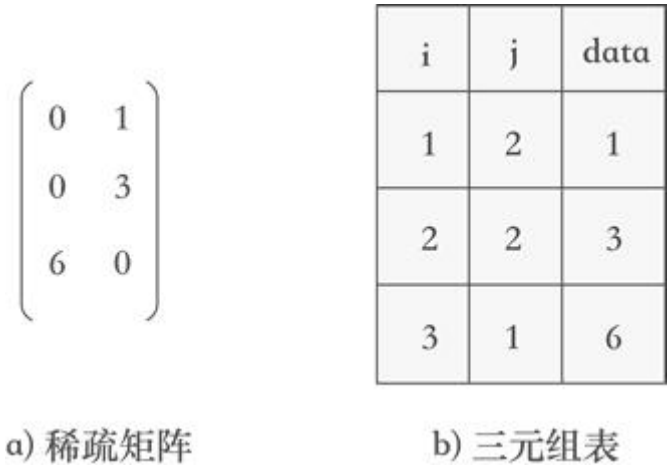


图 1 稀疏矩阵和对应的三元组表

如图 1 所示，此为转置之前的矩阵和对应的三元组表。稀疏矩阵的快速转置是这样的，在普通算法的基础上增设两个数组（假设分别为 array 和 copt）：

- array 数组负责记录原矩阵每一列非 0 元素的个数。以图 1 为例，则对应的 array 数组如图 2 所示：

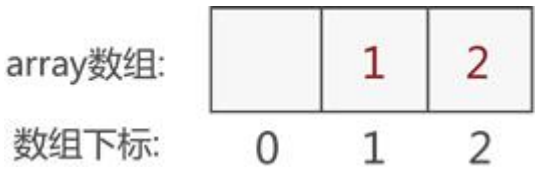


图 2 每一列非 0 元素的个数

图 2 中 array 数组表示，原稀疏矩阵中第一列有 1 个非 0 元素，第二列有 2 个非 0 元素。

- copt 数组用于计算稀疏矩阵中每列第一个非 0 元素在新三元组表中存放的位置。我们通常默认第一列首个非 0 元素存放到新三元组表中的位置为 1，然后通过 cpot[col] = cpot[col-1] + array[col-1] 公式可计算出后续各列首个非 0 元素存放到新三元组表的位置。拿图 1 中的稀疏矩阵来说，它对应的 copt 数组如图 3 所示：

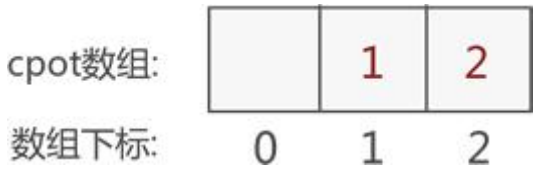


图 3 copt 数组示意图

图 3 中的 copt 数组表示，原稀疏矩阵中第 2 列首个非 0 元素存放到新三元组表的位置为 2。

注意，cpot[col] = cpot[col-1] + array[col-1] 的意思是，后一列首个非 0 元素存放的位置等于前一列首个非 0 元素的存放位置加上该列非 0 元素的个数。由此可以看出，copt 数组才是最终想要的，而 array 数组的设立只是为了帮助我们得到 copt 数组。

这样在实现第 3 步时，根据每个三元组中 j 的数值，可以借助 cpot 数组直接得到此三元组新的存放位置，C 语言实现代码如下：

```
//实现快速转置算法的函数
TSMatrix fastTransposeMatrix(TSMatrix M,TSMatrix T){
    //第 1 步：行和列置换
    T.m=M.n;
    T.n=M.m;
    T.num=M.num;
    if (T.num) {
        //计算 array 数组
        int array[number];
        for (int col=1; col<=M.m; col++) {
            array[col]=0;
        }
        for (int t=0; t<M.num; t++) {
            int j=M.data[t].j;
            array[j]++;
        }

        //创建并初始化 cpot 数组
        int cpot[T.m+1];
        cpot[1]=1;//第一列中第一个非 0 元素的位置默认为 1
        for (int col=2; col<=M.m; col++) {
            cpot[col]=cpot[col-1]+array[col-1];
        }
        //遍历一次即可实现三元组表的转置
        for (int p=0; p<M.num; p++) {
```

```
                //提取当前三元组的列数
                int col=M.data[p].j;
                //根据列数和 cpot 数组，找到当前元素需要存放的位置
                int q=cpot[col];
                //转置矩阵的三元组默认从数组下标 0 开始，而得到的 q 值是单纯的位置，所以要减 1
                T.data[q-1].i=M.data[p].j;
                T.data[q-1].j=M.data[p].i;
                T.data[q-1].data=M.data[p].data;
                //存放完成后，cpot 数组对应的位置要+1，以便下次该列存储下一个三元组
                cpot[col]++;
            }
        }
        return T;
}
```

使用 fastTransposeMatrix 函数实现图 1 中稀疏矩阵转置的 C 语言完整程序为：

```
#include<stdio.h>
#define number 10
typedef struct {
    int i,j;
    int data;
}triple;
typedef struct {
    triple data[number];
    int rpos[number];
    int n,m,num;
}TSMatrix;
//fastTransposeMatrix 放置位置
int main() {
    TSMatrix M;
    M.m=2;
    M.n=3;
    M.num=3;

    M.data[0].i=1;
    M.data[0].j=2;
    M.data[0].data=1;
```

```
    M.data[1].i=2;
    M.data[1].j=2;
    M.data[1].data=3;

    M.data[2].i=3;
    M.data[2].j=1;
    M.data[2].data=6;

    TSMatrix T;
    T=fastTransposeMatrix(M, T);
    printf("转置矩阵三元组表为：\n");
    for (int i=0; i<T.num; i++) {
        printf("(%d,%d,%d)\n",T.data[i].i,T.data[i].j,T.data[i].data);
    }
    return 0;
}
```

程序运行结果为：

```
转置矩阵三元组表为：
(1,3,6)
(2,1,1)
(2,2,3)
```

可以看出，稀疏矩阵快速转置算法的时间复杂度为 $O(n)$。即使在最坏的情况下（矩阵中全部都是非 0 元素），该算法的时间复杂度也才为 $O(n^2)$。

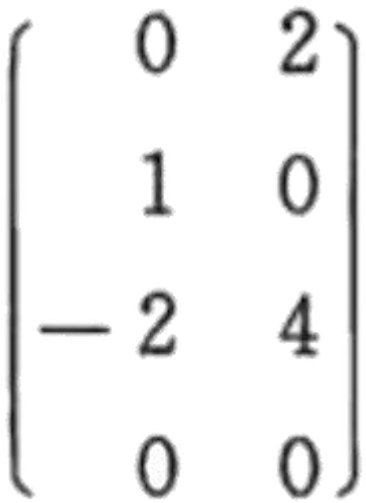
矩阵乘法（行逻辑链接的顺序表）及代码实现

矩阵相乘的前提条件是：乘号前的矩阵的列数要和乘号后的矩阵的行数相等。且矩阵的乘法运算没有交换律，即 A*B 和 B*A 是不一样的。

例如，矩阵 A：

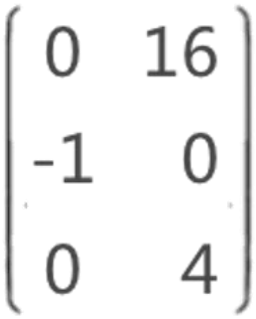
$$\begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

矩阵 B：



由于矩阵 A 的列数和矩阵 B 的行数相等，可以进行 A*B 运算（不能进行 B*A 运算）。计算方法是：用矩阵 A 的第 i 行和矩阵 B 中的每一列 j 对应的数值做乘法运算，乘积——相加，所得结果即为矩阵 C 中第 i 行第 j 列的值。

得到的乘积矩阵 C 为：



例如：$C_{12} = 6$ 是因为：$A_{11}*B_{12} + A_{12}*B_{22} + A_{13}*B_{32} + A_{14}*B_{42}$，即 3*2 + 0*0 + 0*4 + 5*0 = 6 ，因为这是 A 的第 1 行和 B 的第 2 列的乘积和，所以结果放在 C 的第 1 行第 2 列的位置。

例如，A 是 m1*n1 矩阵，B 是 m2*n2 矩阵(前提必须是 n1 == m2)：

```
int C[MAX][MAX];
for (int i=0; i<m1;i++) {
    for (int j=0; j<n2; j++) {
        C[i][j]=0;
        for (int k=0; k<n1; k++) {
            C[i][j]+=A[i][k]*B[k][j];
        }
    }
}
```

普通算法的时间复杂度为 O(m1*n2*n1)。

在稀疏矩阵做乘法运算时，由于本身矩阵中含有的非 0 元素少，普通算法会出现很多 0*0 或者 k*0 或者 0*k （ k 代表非 0 元素值）的情况。下面介绍使用行逻辑链接的顺序表计算矩阵乘积的方法。

行逻辑链接的顺序表解决矩阵乘积算法

对矩阵的乘积进行深度剖析，矩阵 A 和矩阵 B 相乘的运算过程是这样的：

1. 首先，找到矩阵 A 中第一行的非 0 元素，分别是 A_{11} = 3 和 A_{14} = 5；（由于行逻辑链接的顺序表中存储的都是非 0 元素，查找的过程就需要使用记录每行第一个非 0 元素的首地址的数组来完成）

2. 用 3 去和 B 中对应的第一行中的非 0 元素相乘，矩阵 B 中第一行非 0 元素是 B_{12} = 2，所以 3*2 = 6 ，因为 6 是 A_{11} 和 B_{12} 相乘的结果，所以暂时存放在 C_{12} 中；用 5 去和 B 中对应的第 4 行的非 0 元素相乘，由于矩阵 B 中第 4 行没有非 0 元素，所以，第一行的计算结束；

3. 以此类推。

攻克问题难点

现在，解决问题的关键在于，如何知道顺序表中存放的非 0 元素是哪一行的呢?

解决方案：由于使用的是行逻辑链接的顺序表，所以，已经知道了每一个矩阵中的每一行有多少个非 0 元素，而且第一行的第一个非 0 元素的位置一定是 1。

所以，第 n 行的非 0 元素的位置范围是：大于或等于第 n 行第一个元素的位置， 小于第 n+1 行第一个元素的位置（如果是矩阵的最后一行， 小于矩阵中非 0 元素的个数 + 1）。

具体实现代码

```
#include <stdio.h>
#define MAXSIZE 12500
#define MAXRC 100
#define         ElemType int
typedef struct
{
    int i,j;//行，列
    ElemType e;//元素值
}Triple;

typedef struct
{
    Triple    data[MAXSIZE+1];
    int rpos[MAXRC+1];//每行第一个非零元素在 data 数组中的位置
    int mu,nu,tu;//行数，列数，元素个数
```

```
}RLSMatrix;


RLSMatrix MultSMatrix(RLSMatrix A, RLSMatrix B, RLSMatrix C)
{
    //如果矩阵 A 的列数与矩阵 B 的行数不等，则不能做矩阵乘运算
    if(A.nu != B.mu)
        return C;
    C.mu = A.mu;
    C.nu = B.nu;
    C.tu = 0;
    //如果其中任意矩阵的元素个数为零，做乘法元素没有意义，全是 0
    if(A.tu * B.tu == 0)
        return C;
    else
    {
        int arow;
        int ccol;
        //遍历矩阵 A 的每一行
        for(arow=1; arow<=A.mu; arow++)
        {
            //创建一个临时存储乘积结果的数组，且初始化为 0，遍历每次都需要清空
            int ctemp[MAXRC+1] ={};
            C.rpos[arow] = C.tu + 1;
            //根据行数，在三元组表中找到该行所有的非 0 元素的位置
            int tp;
            if(arow < A.mu)
                tp = A.rpos[arow+1];//获取矩阵 A 的下一行第一个非零元素在 data 数组中位置
            else
                tp = A.tu+1;//若当前行是最后一行，则取最后一个元素+1

            int p;
            int brow;
            //遍历当前行的所有的非 0 元素
            for(p=A.rpos[arow]; p<tp; p++)
            {
                brow = A.data[p].j;//取该非 0 元素的列数，便于去 B 中找对应的做乘积的非 0 元素
                int t;
```

```
                // 判断如果对于 A 中非 0 元素，找到矩阵 B 中做乘法的那一行中的所有的非 0 元素
                if(brow < B.mu)
                    t = B.rpos[brow+1];
                else
                    t = B.tu+1;
                int q;
                //遍历找到的对应的非 0 元素，开始做乘积运算
                for(q=B.rpos[brow]; q<t; q++)
                {
                    //得到的乘积结果，每次和 ctemp 数组中相应位置的数值做加和运算
                    ccol = B.data[q].j;
                    ctemp[ccol] += A.data[p].e * B.data[q].e;
                }
            }
            //矩阵 C 的行数等于矩阵 A 的行数，列数等于矩阵 B 的列数，所以，得到的 ctemp 存储的结果，也会在 C 的列数的范围内
            for(ccol=1; ccol<=C.nu; ccol++)
            {
                //由于结果可以是 0，而 0 不需要存储，所以在这里需要判断
                if(ctemp[ccol])
                {
                    //不为 0，则记录矩阵中非 0 元素的个数的变量 tu 要+1；且该值不能超过存放三元素数组的空间大小
                    if(++C.tu > MAXSIZE)
                        return C;
                    else{
                        C.data[C.tu].e = ctemp[ccol];
                        C.data[C.tu].i = arow;
                        C.data[C.tu].j = ccol;
                    }
                }
            }
        }
        return C;
    }
}
int main(int argc, char* argv[])
{
```

```
RLSMatrix M,N,T;

M.tu = 4;
M.mu = 3;
M.nu = 4;

M.rpos[1] = 1;
M.rpos[2] = 3;
M.rpos[3] = 4;

M.data[1].e = 3;
M.data[1].i = 1;
M.data[1].j = 1;

M.data[2].e = 5;
M.data[2].i = 1;
M.data[2].j = 4;

M.data[3].e = -1;
M.data[3].i = 2;
M.data[3].j = 2;

M.data[4].e = 2;
M.data[4].i = 3;
M.data[4].j = 1;

N.tu = 4;
N.mu = 4;
N.nu = 2;

N.rpos[1] = 1;
N.rpos[2] = 2;
N.rpos[3] = 3;
N.rpos[4] = 5;

N.data[1].e = 2;
N.data[1].i = 1;
```

```
    N.data[1].j = 2;

    N.data[2].e = 1;
    N.data[2].i = 2;
    N.data[2].j = 1;

    N.data[3].e = -2;
    N.data[3].i = 3;
    N.data[3].j = 1;

    N.data[4].e = 4;
    N.data[4].i = 3;
    N.data[4].j = 2;

   T= MultSMatrix(M,N,T);
    for (int i=1; i<=T.tu; i++) {
        printf("(%d,%d,%d)\n",T.data[i].i,T.data[i].j,T.data[i].e);
    }
    return 0;
}
```

输出结果：

```
(1,2,6)
(2,1,-1)
(3,2,4)
```

总结

当稀疏矩阵 A_{mn} 和稀疏矩阵 B_{np} 采用行逻辑链接的顺序表做乘法运算时，在矩阵 A 的列数（矩阵 B 的行数） n 不是很大的情况下，算法的时间复杂度相当于 O(m*p)，比普通算法要快很多。

矩阵加法（基于十字链表）及 C 语言代码实现

矩阵之间能够进行加法运算的前提条件是：各矩阵的行数和列数必须相等。

在行数和列数都相等的情况下，矩阵相加的结果就是矩阵中对应位置的值相加所组成的矩阵，例如：

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & -1 \\ 0 & -1 & -1 \\ -1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

图 1 矩阵相加

十字链表法

之前所介绍的都是采用顺序存储结构存储三元组，在类似于矩阵的加法运算中，矩阵中的数据元素变化较大（这里的变化主要为：非 0 元素变为 0，0 变为非 0 元素），就需要考虑采用另一种结构——链式存储结构来存储三元组。

采用链式存储结构存储稀疏矩阵三元组的方法，称为“十字链表法”。

十字链表法表示矩阵

例如，用十字链表法表示矩阵 A ，为：

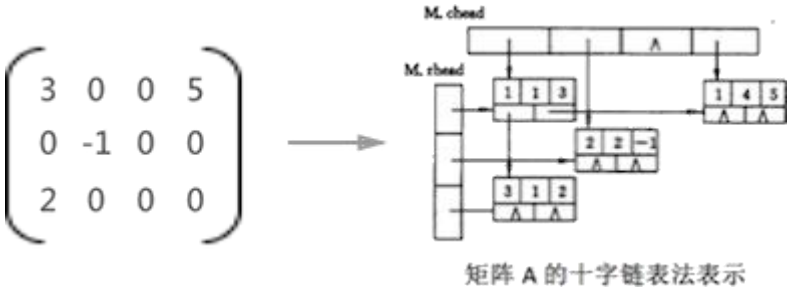


图 2 矩阵用十字链表法表示

由此可见，采用十字链表表示矩阵时，矩阵的每一行和每一个列都可以看作是一个单独的链表，而之所以能够表示矩阵，是因为行链表和列链表都分别存储在各自的数组中

图 2 中：存储行链表的数组称为 rhead 数组；存储列链表的数组称为 chead 数组。

十字链表中的结点

从图 2 中的十字链表表示矩阵的例子可以看到，十字链表中的结点由 5 部分组成：

图 3 十字链表中的结点

指针域 A 存储的是矩阵中结点所在列的下一个结点的地址（称为 “down 域”）；
指针域 B 存储的是矩阵中该结点所在行的下一个结点的地址（称为 “right 域”）；

用结构体自定义表示为：

```
typedef struct OLNode
{
    int i,j,e;                          //矩阵三元组 i 代表行 j 代表列 e 代表当前位置的数据
```

```
    struct OLNode *right,*down;        //指针域 右指针 下指针
}OLNode,*OLink;
```

十字链表的结构

使用十字链表表示一个完整的矩阵，在了解矩阵中各结点的结构外，还需要存储矩阵的行数、列数以及非 0 元素的个数，另外，还需要将各结点链接成的链表存储在数组中。

所以，采用结构体自定义十字链表的结构，为：

```
typedef struct
{
    OLink *rhead,*chead;                //存放各行和列链表头指针的数组
    int mu,nu,tu;                       //矩阵的行数,列数和非零元的个数
}CrossList;
```

十字链表存储矩阵三元组

由于三元组存储的是该数据元素的行标、列标和数值，所以，通过行标和列标，就能在十字链表中唯一确定一个位置。

判断方法为：在同一行中通过列标来判断位置；在同一列中通过行标来判断位置。

首先判断该数据元素 A（例如三元组为：（i，j，k））所在行的具体位置：

- 如果 A 的列标 j 值比该行第一个非 0 元素 B 的 j 值小，说明该数据元素在元素 B 的左侧，这时 A 就成为了该行第一个非 0 元素（也适用于当该行没有非 0 元素的情况，可以一并讨论）
- 如果 A 的列标 j 比该行第一个非 0 元素 B 的 j 值大，说明 A 在 B 的右侧，这时，就需要遍历该行链表，找到插入位置的前一个结点，进行插入。

对应行链表的位置确定之后，判断数据元素 A 在对应列的位置：

1. 如果 A 的行标比该列第一个非 0 元素 B 的行标 i 值还小，说明 A 在 B 的上边，这时 A 就成了该列第一个非 0 元素。（也适用于该列没有非 0 元素的情况）
2. 反之，说明 A 在 B 的下边，这时就需要遍历该列链表，找到要插入位置的上一个数据元素，进行插入。

实现代码：

```
//创建系数矩阵 M,采用十字链表存储表示
CrossList CreateMatrix_OL(CrossList M)
{
    int m,n,t;
    int i,j,e;
    OLNode *p,*q;//定义辅助变量
    scanf("%d%d%d",&m,&n,&t);   //输入矩阵的行列及非零元的数量
    //初始化矩阵的行列及非零元的数量
    M.mu=m;
```

```
    M.nu=n;
    M.tu=t;
    if(!(M.rhead=(OLink*)malloc((m+1)*sizeof(OLink)))||!(M.chead=(OLink*)malloc((n+1)*sizeof(OLink))))
    {
        printf("初始化矩阵失败");
        exit(0);          //初始化矩阵的行列链表
    }
    for(i=1;i<=m;i++)
    {
        M.rhead[i]=NULL;     //初始化行
    }
    for(j=1;j<=n;j++)
    {
        M.chead[j]=NULL;     //初始化列
    }
    for(scanf("%d%d%d",&i,&j,&e);0!=i;scanf("%d%d%d",&i,&j,&e)) //输入三元组  直到行为 0 结束
    {
        if(!(p=(OLNode*)malloc(sizeof(OLNode))))
        {
            printf("初始化三元组失败");
            exit(0);                          //动态生成 p
        }
        p->i=i;
        p->j=j;
        p->e=e;                                    //初始化 p
        if(NULL==M.rhead[i]||M.rhead[i]->j>j)
        {
            p->right=M.rhead[i];
            M.rhead[i]=p;
        }
        else
        {
            for(q=M.rhead[i];(q->right)&&q->right->j<j;q=q->right);
            p->right=q->right;
            q->right=p;
        }
```

```
        if(NULL==M.chead[j]||M.chead[j]->i>i)
        {
            p->down=M.chead[j];
            M.chead[j]=p;
        }
        else
        {
            for (q=M.chead[j];(q->down)&& q->down->i<i;q=q->down);
            p->down=q->down;
            q->down=p;
        }
    }
    return M;
}
```

十字链表解决矩阵相加问题

在解决 “将矩阵 B 加到矩阵 A ” 的问题时，由于采用的是十字链表法存储矩阵的三元组，所以在相加的过程中，针对矩阵 B 中每一个非 0 元素，需要判断在矩阵 A 中相对应的位置，有三种情况：

1. 提取到的 B 中的三元组在 A 相应位置上没有非 0 元素，此时直接加到矩阵 A 该行链表的对应位置上；
2. 提取到的 B 中三元组在 A 相应位置上有非 0 元素，且相加不为 0 ，此时只需要更改 A 中对应位置上的三元组的值即可；
3. 提取到的 B 中三元组在 A 响应位置上有非 0 元素，但相加为 0 ，此时需要删除矩阵 A 中对应结点。

> 提示：算法中，只需要逐个提取矩阵 B 中的非 0 元素，然后判断矩阵 A 中对应位置上是否有非 0 元素，根据不同的情况，相应作出处理。

设指针 pa 和 pb 分别表示矩阵 A 和矩阵 B 中同一行中的结点（ pb 和 pa 都是从两矩阵的第一行的第一个非 0 元素开始遍历），针对上面的三种情况，细分为 4 种处理过程（第一种情况下有两种不同情况）：

1. 当 pa 结点的列值 j > pb 结点的列值 j 或者 pa == NULL （说明矩阵 A 该行没有非 0 元素），两种情况下是一个结果，就是将 pb 结点插入到矩阵 A 中。
2. 当 pa 结点的列值 j < pb 结点的列值 j ，说明此时 pb 指向的结点位置比较靠后，此时需要移动 pa 的位置，找到离 pb 位置最近的非 0 元素，然后在新的 pa 结点的位置后边插入；
3. 当 pa 的列值 j == pb 的列值 j， 且两结点的值相加结果不为 0 ，只需要更改 pa 指向的结点的值即可；
4. 当 pa 的列值 j == pb 的列值 j ，但是两结点的值相加结果为 0 ，就需要从矩阵 A 的十字链表中删除 pa 指向的结点。

实现代码：

```
CrossList AddSMatrix(CrossList M,CrossList N){
    OLNode * pa,*pb;//新增的两个用于遍历两个矩阵的结点
    OLink * hl=(OLink*)malloc(M.nu*sizeof(OLink));//用于存储当前遍历的行为止以上的区域每一个列的最后一个非 0 元素的位置。
    OLNode * pre=NULL;//用于指向 pa 指针所在位置的此行的前一个结点
```

```
    //遍历初期，首先要对 hl 数组进行初始化，指向每一列的第一个非 0 元素
    for (int j=1; j<=M.nu; j++) {
        hl[j]=M.chead[j];
    }
    //按照行进行遍历
    for (int i=1; i<=M.mu; i++) {
        //遍历每一行以前，都要 pa 指向矩阵 M 当前行的第一个非 0 元素；指针 pb 也是如此，只不过遍历对象为矩阵 N
        pa=M.rhead[i];
        pb=N.rhead[i];
        //当 pb 为 NULL 时，说明矩阵 N 的当前行的非 0 元素已经遍历完。
        while (pb!=NULL) {
            //创建一个新的结点，每次都要复制一个 pb 结点，但是两个指针域除外。（复制的目的就是排除指针域的干扰）
            OLNode * p=(OLNode*)malloc(sizeof(OLNode));
            p->i=pb->i;
            p->j=pb->j;
            p->e=pb->e;
            p->down=NULL;
            p->right=NULL;

            //第一种情况
            if (pa==NULL||pa->j>pb->j) {
                //如果 pre 为 NULL，说明矩阵 M 此行没有非 0 元素
                if (pre==NULL) {
                    M.rhead[p->i]=p;
                }else{//由于程序开始时 pre 肯定为 NULL，所以，pre 指向的是第一个 p 的位置，在后面的遍历过程中，p 指向的位置是逐渐向后移动的，所有，pre 肯定会在 p 的前边
                    pre->right=p;
                }
                p->right=pa;
                pre=p;
                //在链接好行链表之后，链接到对应列的列链表中的相应位置
                if (!M.chead[p->j]||M.chead[p->j]->i>p->i) {
                    p->down=M.chead[p->j];
                    M.chead[p->j]=p;
                }else{
                    p->down=hl[p->j]->down;
                    hl[p->j]->down=p;
```

```
                }
                //更新 hl 中的数据
                hl[p->j]=p;
            }else{
                //第二种情况，只需要移动 pa 的位置，继续判断 pa 和 pb 的位置，一定要有 continue
                if (pa->j<pb->j) {
                    pre=pa;
                    pa=pa->right;
                    continue;
                }
                //第三、四种情况，当行标和列标都想等的情况下，需要讨论两者相加的值的问题
                if (pa->j==pb->j) {
                pa->e+=pb->e;
                //如果为 0，摘除当前结点，并释放所占的空间
                if (pa->e==0) {
                    if (pre==NULL) {
                        M.rhead[pa->i]=pa->right;
                    }else{
                        pre->right=pa->right;
                    }
                    p=pa;
                    pa=pa->right;
                    if (M.chead[p->j]==p) {
                        M.chead[p->j]=hl[p->j]=p->down;
                    }else{
                        hl[p->j]->down=p->down;
                    }
                    free(p);
                    }
                }
            }
            pb=pb->right;
        }
    }
    //用于输出矩阵三元组的功能函数
    display(M);
    return M;
```

```
}
```

完整代码演示

```
#include<stdio.h>
#include<stdlib.h>
typedef struct OLNode
{
    int i,j,e;                          //矩阵三元组 i 代表行  j 代表列  e 代表当前位置的数据
    struct OLNode *right,*down;         //指针域  右指针  下指针
}OLNode,*OLink;

typedef struct
{
    OLink *rhead,*chead;                //行和列链表头指针
    int mu,nu,tu;                       //矩阵的行数,列数和非零元的个数
}CrossList;

CrossList CreateMatrix_OL(CrossList M);
CrossList AddSMatrix(CrossList M,CrossList N);

void display(CrossList M);

void main()
{
    CrossList M,N;
    printf("输入测试矩阵 M:\n");
    M=CreateMatrix_OL(M);
    printf("输入测试矩阵 N:\n");
    N=CreateMatrix_OL(N);
    M=AddSMatrix(M,N);
    printf("矩阵相加的结果为:\n");
    display(M);
}

CrossList CreateMatrix_OL(CrossList M)
{
    int m,n,t;
```

```
    int i,j,e;
    OLNode *p,*q;
    scanf("%d%d%d",&m,&n,&t);
    M.mu=m;
    M.nu=n;
    M.tu=t;
    if(!(M.rhead=(OLink*)malloc((m+1)*sizeof(OLink)))||!(M.chead=(OLink*)malloc((n+1)*sizeof(OLink))))
    {
        printf("初始化矩阵失败");
        exit(0);
    }
    for(i=1;i<=m;i++)
    {
        M.rhead[i]=NULL;
    }
    for(j=1;j<=n;j++)
    {
        M.chead[j]=NULL;
    }
    for(scanf("%d%d%d",&i,&j,&e);0!=i;scanf("%d%d%d",&i,&j,&e))       {
        if(!(p=(OLNode*)malloc(sizeof(OLNode))))
        {
            printf("初始化三元组失败");
            exit(0);
        }
        p->i=i;
        p->j=j;
        p->e=e;
        if(NULL==M.rhead[i]||M.rhead[i]->j>j)
        {
            p->right=M.rhead[i];
            M.rhead[i]=p;
        }
        else
        {
            for(q=M.rhead[i];(q->right)&&q->right->j<j;q=q->right);
            p->right=q->right;
```

```
                q->right=p;
            }

            if(NULL==M.chead[j]||M.chead[j]->i>i)
            {
                p->down=M.chead[j];
                M.chead[j]=p;
            }
            else
            {
                for (q=M.chead[j];(q->down)&& q->down->i<i;q=q->down);
                p->down=q->down;
                q->down=p;
            }
        }
        return M;
}

CrossList AddSMatrix(CrossList M,CrossList N){
        OLNode * pa,*pb;
        OLink * hl=(OLink*)malloc(M.nu*sizeof(OLink));
        OLNode * pre=NULL;
        for (int j=1; j<=M.nu; j++) {
            hl[j]=M.chead[j];
        }
        for (int i=1; i<=M.mu; i++) {
            pa=M.rhead[i];
            pb=N.rhead[i];
            while (pb!=NULL) {
                OLNode * p=(OLNode*)malloc(sizeof(OLNode));
                p->i=pb->i;
                p->j=pb->j;
                p->e=pb->e;
                p->down=NULL;
                p->right=NULL;
                if (pa==NULL||pa->j>pb->j) {
                    if (pre==NULL) {
```

```
                M.rhead[p->i]=p;
            }else{
                pre->right=p;
            }
            p->right=pa;
            pre=p;
            if (!M.chead[p->j]||M.chead[p->j]->i>p->i) {
                p->down=M.chead[p->j];
                M.chead[p->j]=p;
            }else{
                p->down=hl[p->j]->down;
                hl[p->j]->down=p;
            }
            hl[p->j]=p;
        }else{
            if (pa->j<pb->j) {
                pre=pa;
                pa=pa->right;
                continue;
            }
            if (pa->j==pb->j) {
            pa->e+=pb->e;
            if (pa->e==0) {
                if (pre==NULL) {
                    M.rhead[pa->i]=pa->right;
                }else{
                    pre->right=pa->right;
                }
                p=pa;
                pa=pa->right;
                if (M.chead[p->j]==p) {
                    M.chead[p->j]=hl[p->j]=p->down;
                }else{
                    hl[p->j]->down=p->down;
                }
                free(p);
                }
```

```
                }
            }
            pb=pb->right;
        }
    }
    display(M);
    return M;
}

void display(CrossList M){
    printf("输出测试矩阵:\n");
    printf("M:\n--------------------\ni\tj\te\n--------------------\n");
    for (int i=1;i<=M.nu;i++)
    {
        if (NULL!=M.chead[i])
        {
            OLink p=M.chead[i];
            while (NULL!=p)
            {
                printf("%d\t%d\t%d\n",p->i,p->j,p->e);
                p=p->down;
            }
        }
    }
}
```

运行结果：

```
输入测试矩阵 M:
3 3 3
1 2 1
2 1 1
3 3 1
0 0 0
输入测试矩阵 N:
3 3 4
1 2 -1
1 3 1
2 3 1
3 1 1
```

```
0 0 0
矩阵相加的结果为:
输出测试矩阵:
M:
----------------------
i j e
----------------------
2 1 1
3 1 1
1 3 1
2 3 1
3 3 1
```

总结

使用十字链表法解决稀疏矩阵的压缩存储的同时，在解决矩阵相加的问题中，对于某个单独的结点来说，算法的时间复杂度为一个常数（全部为选择结构），算法的整体的时间复杂度取决于两矩阵中非 0 元素的个数。

广义表的深度和长度（C 语言）详解

前面学习了广义表及其对应的存储结构，本节来学习如何计算广义表的深度和长度，以及如何编写相应的 C 语言实现程序。

广义表的长度

广义表的长度，指的是广义表中所包含的数据元素的个数。

由于广义表中可以同时存储原子和子表两种类型的数据，因此在计算广义表的长度时规定，广义表中存储的每个原子算作一个数据，同样每个子表也只算作是一个数据。

例如，在广义表 {a,{b,c,d}} 中，它包含一个原子和一个子表，因此该广义表的长度为 2。

再比如，广义表 {{a,b,c}} 中只有一个子表 {a,b,c}，因此它的长度为 1。

前面我们用 $LS=\{a_1,a_2,...,a_n\}$ 来表示一个广义表，其中每个 a_i 都可用来表示一个原子或子表，其实它还可以表示广义表 LS 的长度为 n。

> 广义表规定，空表 {} 的长度为 0。

在编程实现求广义表长度时，由于广义表的存储使用的是链表结构，且有以下两种方式（如图 1 所示）：

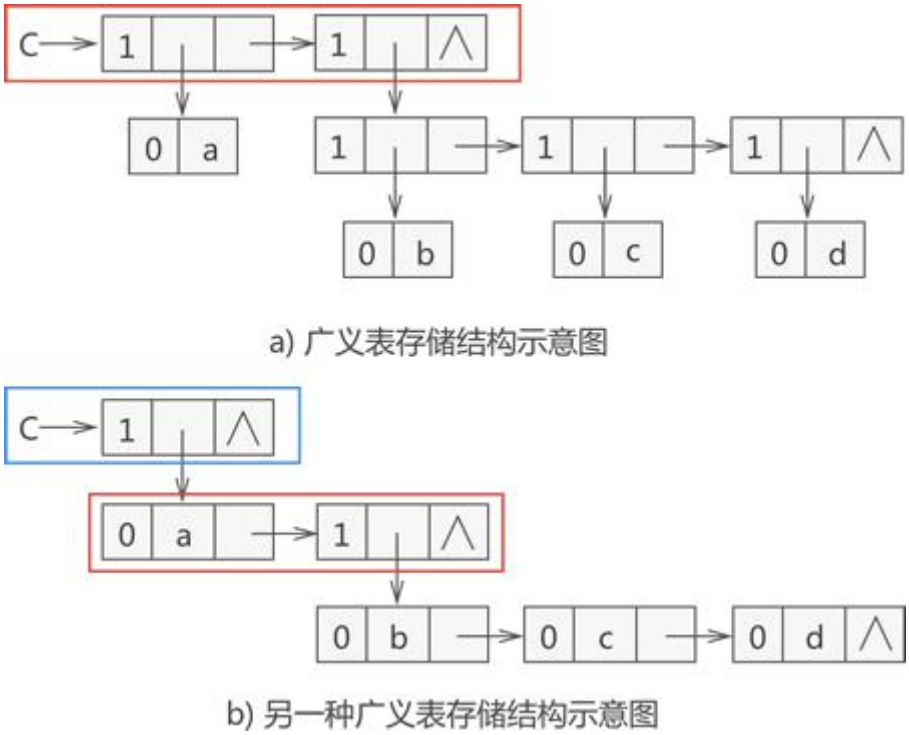


图 1 存储 {a,{b,c,d}} 的两种方式

对于图 1a) 来说，只需计算最顶层（红色标注）含有的节点数量，即可求的广义表的长度。

同理，对于图 1b) 来说，由于其最顶层（蓝色标注）表示的此广义表，而第二层（红色标注）表示的才是该广义表中包含的数据元素，因此可以通过计算第二层中包含的节点数量，即可求得广义表的长度。

由于两种算法的实现非常简单，这里只给出计算图 1a) 中广义表长度的 C 语言实现代码：

```
#include <stdio.h>
#include <stdlib.h>
typedef struct GLNode{
    int tag;//标志域
    union{
        char atom;//原子结点的值域
        struct{
            struct GLNode * hp,*tp;
        }ptr;//子表结点的指针域，hp 指向表头；tp 指向表尾
    };
}*Glist;
Glist creatGlist(Glist C){
    //广义表 C
    C=(Glist)malloc(sizeof(Glist));
    C->tag=1;
    //表头原子'a'
```

```
    C->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.hp->tag=0;
    C->ptr.hp->atom='a';
    //表尾子表（b,c,d）,是一个整体
    C->ptr.tp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.tp=NULL;
    //开始存放下一个数据元素（b,c,d）,表头为‘b’，表尾为（c,d）
    C->ptr.tp->ptr.hp->tag=1;
    C->ptr.tp->ptr.hp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.hp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.hp->atom='b';
    C->ptr.tp->ptr.hp->ptr.tp=(Glist)malloc(sizeof(Glist));
    //存放子表(c,d)，表头为 c，表尾为 d
    C->ptr.tp->ptr.hp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->atom='c';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp=(Glist)malloc(sizeof(Glist));
    //存放表尾 d
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->atom='d';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.tp=NULL;
    return C;
}
int GlistLength(Glist C){
    int Number=0;
    Glist P=C;
    while(P){
        Number++;
        P=P->ptr.tp;
    }
    return Number;
}
```

```
int main(){
    Glist C = creatGlist(C);
    printf("广义表的长度为：%d",GlistLength(C));
    return 0;
}
```

程序运行结果为：

```
广义表的长度为：2
```

广义表的深度

广义表的深度，可以通过观察该表中所包含括号的层数间接得到。

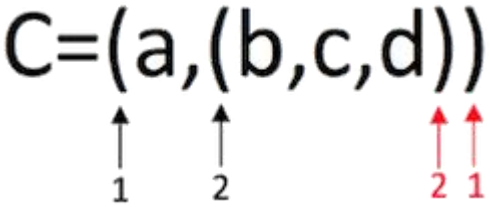


图 2 广义表示意图

从图 2 中可以看到，此广义表从左往右数有两层左括号（从右往左数也是如此），因此该广义表的深度为 2。

再比如，广义表 {{1,2},{3,{4,5}}} 中，子表 {1,2} 和 {3,{4,5}} 位于同层，此广义表中包含 3 层括号，因此深度为 3。

以上观察括号的方法需将广义表当做字符串看待，并借助栈存储结构实现，这里不做重点介绍。

编写程序计算广义表的深度时，以图 1a) 中的广义表为例，可以采用递归的方式：

- 依次遍历广义表 C 的每个节点，若当前节点为原子（tag 值为 0），则返回 0；若为空表，则返回 1；反之，则继续遍历该子表中的数据元素。
- 设置一个初始值为 0 的整形变量 max，每次递归过程返回时，令 max 与返回值进行比较，并取较大值。这样，当整个广义表递归结束时，max+1 就是广义表的深度。

其实，每次递归返回的值都是当前所在的子表的深度，原子默认深度为 0，空表默认深度为 1。

C 语言实现代码如下：

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

typedef struct GLNode{
    int tag;//标志域
    union{
```

```
        char atom;//原子结点的值域
        struct{
            struct GLNode * hp,*tp;
        }ptr;//子表结点的指针域，hp 指向表头；tp 指向表尾
    };
}*Glist,GNode;

Glist creatGlist(Glist C){
    //广义表 C
    C=(Glist)malloc(sizeof(GNode));
    C->tag=1;
    //表头原子‘a’
    C->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.hp->tag=0;
    C->ptr.hp->atom='a';
    //表尾子表（b,c,d）,是一个整体
    C->ptr.tp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.tp=NULL;
    //开始存放下一个数据元素（b,c,d）,表头为‘b’，表尾为（c,d）
    C->ptr.tp->ptr.hp->tag=1;
    C->ptr.tp->ptr.hp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.hp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.hp->atom='b';
    C->ptr.tp->ptr.hp->ptr.tp=(Glist)malloc(sizeof(GNode));
    //存放子表(c,d)，表头为 c，表尾为 d
    C->ptr.tp->ptr.hp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->atom='c';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp=(Glist)malloc(sizeof(GNode));
    //存放表尾 d
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->atom='d';
```

```
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.tp=NULL;
    return C;
}
int GlistDepth(Glist C){
    //如果表 C 为空表时，直接返回长度 1；
    if (!C) {
        return 1;
    }
    //如果表 C 为原子时，直接返回 0；
    if (C->tag==0) {
        return 0;
    }
    int max=0;//设置表 C 的初始长度为 0；
    for (Glist pp=C; pp; pp=pp->ptr.tp) {
        int dep=GlistDepth(pp->ptr.hp);
        if (dep>max) {
            max=dep;//每次找到表中遍历到深度最大的表，并用 max 记录
        }
    }
    //程序运行至此处，表明广义表不是空表，由于原子返回的是 0，而实际长度是 1，所以，此处要+1；
    return max+1;
}
int main(int argc, const char * argv[]) {
    Glist C=creatGlist(C);
    printf("广义表的深度为：%d",GlistDepth(C));
    return 0;
}
```

程序运行结果：

```
广义表的深度为：2
```

浅谈二叉树的（4 种）遍历算法

遍历二叉树可以算作是对树存储结构做的最多的操作，既是重点，也是难点。本节将从初学者的角度给大家分析一下 4 种遍历二叉树算法的由来。

遍历二叉树的算法

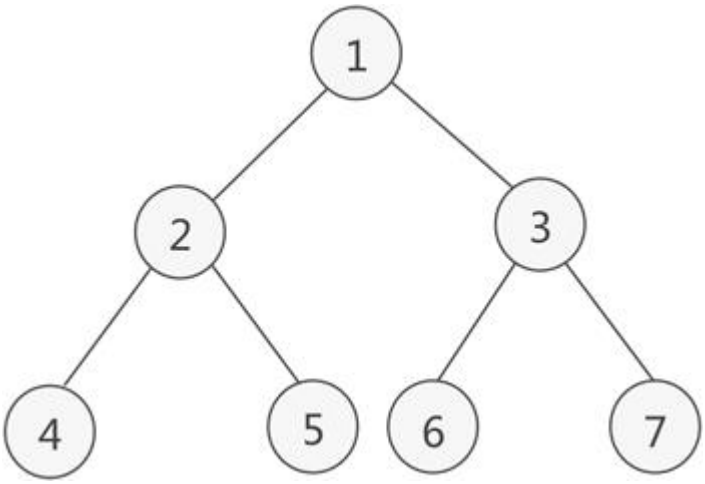


图 1 二叉树示意图

图 1 是一棵二叉树，对于初学者而言，遍历这棵二叉树无非有以下两种方式。

层次遍历

前面讲过，树是有层次的，拿图 1 来说，该二叉树的层次为 3。通过对树中各层的节点从左到右依次遍历，即可实现对正棵二叉树的遍历，此种方式称为层次遍历。

比如，对图 1 中二叉树进行层次遍历，遍历过程如图 2 所示：

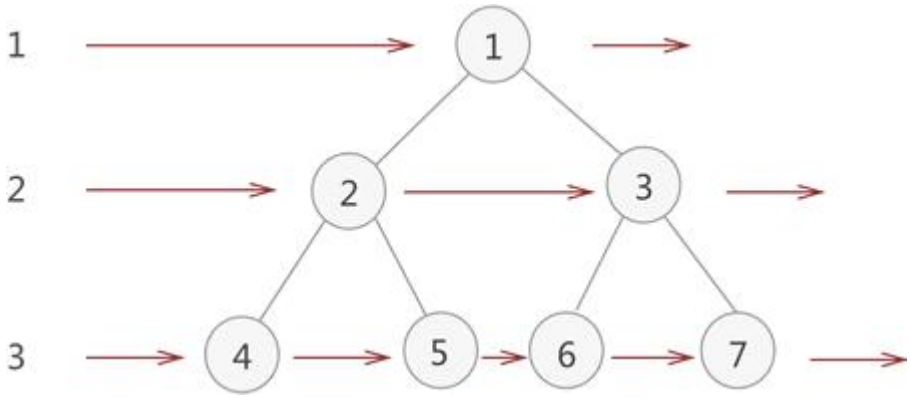


图 2 层次遍历二叉树示意图

普通遍历

其实，还有一种更普通的遍历二叉树的思想，即按照 "从上到下，从左到右" 的顺序遍历整棵二叉树。

还拿图 1 中的二叉树举例，其遍历过程如图 3 所示：

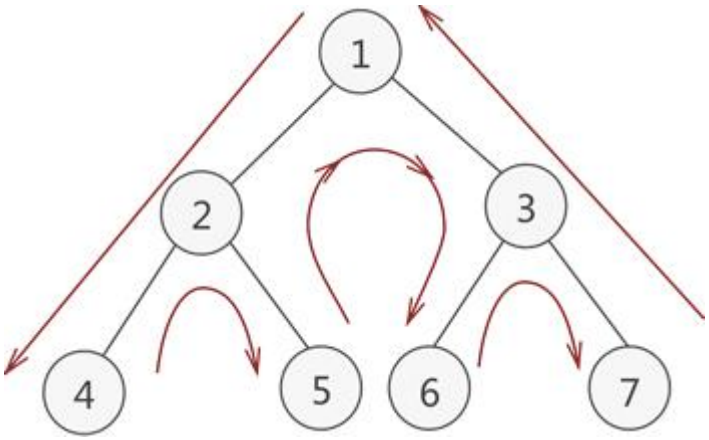


图 3 普通方式遍历二叉树

以上仅是从初学者的角度，对遍历二叉树的过程进行了分析。接下来我们从程序员的角度再对以上两种遍历方式进行剖析。

> 这里，我们要建立一个共识，即成功遍历二叉树的标志是能够成功访问到二叉树中所有的节点。

二叉树遍历算法再剖析

首先观察图 2 中的层次遍历，整个遍历过程只经过各个节点一次，因此在层次遍历过程，每经过一个节点，都必须立刻访问该节点，否则错失良机，后续无法再对其访问。

若对图 1 中二叉树进行层次遍历，则访问树中节点的次序为：

```
1 2 3 4 5 6 7
```

而普通遍历方式则不同，通过观察图 3 可以看到，整个遍历二叉树的过程中，每个节点都被经过了 3 次（虽然叶子节点看似只经过了 2 次，但实际上可以看做是 3 次）。以图 3 中的节点 2 为例，如图 4 所示，它被经过了 3 次。

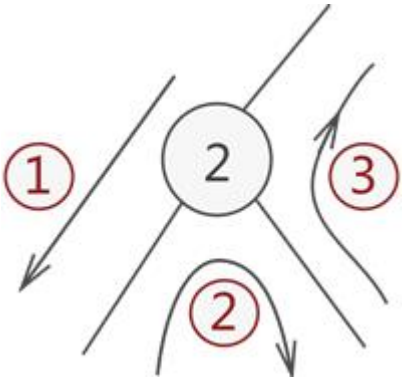


图 4 遍历节点 2 的过程示意图

因此，在编程实现时，我们可以设定真正访问各个节点的时机，换句话说，我们既可以在第一次经过各节点时就执行访问程序，也可以在第二次经过各节点时访问，甚至可以在最后一次经过各节点时访问。

这也就引出了以下 3 种遍历二叉树的算法：

1. 先序遍历：每遇到一个节点，先访问，然后再遍历其左右子树（对应图 4 中的 ①）；

2. 中序遍历：第一次经过时不访问，等遍历完左子树之后再访问，然后遍历右子树（对应图 4 中的 ②）；

3. 后序遍历：第一次和第二次经过时都不访问，等遍历完该节点的左右子树之后，最后访问该节点（对应图 4 中的 ③）；

以图 1 中的二叉树为例，其先序遍历算法访问节点的先后次序为：

```
1 2 4 5 3 6 7
```

中序遍历算法访问节点的次序为：

```
4 2 5 1 6 3 7
```

后序遍历访问节点的次序为：

```
4 5 2 6 7 3 1
```

以上就是二叉树 4 种遍历算法的由来，其各个算法的具体实现过程其代码实现后续章节会详解介绍。

线索二叉树的创建及遍历(C 语言实现)

通过前面对二叉树的学习，了解到二叉树本身是一种非线性结构，采用任何一种遍历二叉树的方法，都可以得到树中所有结点的一个线性序列。在这个序列中，除第一个结点外，每个结点都有自己的直接前趋；除最后一个结点外，每个结点都有一个直接后继。

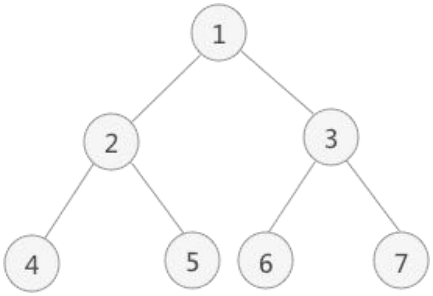


图 1 满二叉树

例如，图 1 采用先序遍历的方法得到的结点序列为：`1 2 4 5 3 6 7`，在这个序列中，结点 2 的直接前趋结点为 1，直接后继结点为 4。

什么是线索二叉树

如果算法中多次涉及到对二叉树的遍历，普通的二叉树就需要使用栈结构做重复性的操作。

线索二叉树不需要如此，在遍历的同时，使用二叉树中空闲的内存空间记录某些结点的前趋和后继元素的位置（不是全部）。这样在算法后期需要遍历二叉树时，就可以利用保存的结点信息，提高了遍历的效率。使用这种方法构建的二叉树，即为“线索二叉树”。

线索二叉树的结点结构

如果在二叉树中想保存每个结点前趋和后继所在的位置信息，最直接的想法就是改变结点的结构，即添加两个指针域，分别指向该结点的前趋和后继。

但是这种方式会降低树存储结构的存储密度。而对于二叉树来讲，其本身还有很多未利用的空间。

> 存储密度指的是数据本身所占的存储空间和整个结点结构所占的存储量之比。

每一棵二叉树上，很多结点都含有未使用的指向 NULL 的指针域。除了度为 2 的结点，度为 1 的结点，有一个空的指针域；叶子结点两个指针域都为 NULL。

规律：在有 n 个结点的二叉链表中必定存在 n+1 个空指针域。

线索二叉树实际上就是使用这些空指针域来存储结点之间前趋和后继关系的一种特殊的二叉树。

线索二叉树中，如果结点有左子树，则 lchild 指针域指向左孩子，否则 lchild 指针域指向该结点的直接前趋；同样，如果结点有右子树，则 rchild 指针域指向右孩子，否则 rchild 指针域指向该结点的直接后继。

为了避免指针域指向的结点的意义混淆，需要改变结点本身的结构，增加两个标志域，如图 2 所示。

图 2 线索二叉树中的结点结构

LTag 和 RTag 为标志域。实际上就是两个布尔类型的变量：

- LTag 值为 0 时，表示 lchild 指针域指向的是该结点的左孩子；为 1 时，表示指向的是该结点的直接前趋结点；
- RTag 值为 0 时，表示 rchild 指针域指向的是该结点的右孩子；为 1 时，表示指向的是该结点的直接后继结点。

结点结构代码实现：

```
#define TElemType int//宏定义，结点中数据域的类型
//枚举，Link 为 0，Thread 为 1
typedef enum PointerTag{
    Link,
    Thread
}PointerTag;
//结点结构构造
typedef struct BiThrNode{
    TElemType data;//数据域
    struct BiThrNode* lchild,*rchild;//左孩子，右孩子指针域

    PointerTag Ltag,Rtag;//标志域，枚举类型
}BiThrNode,*BiThrTree;
```

表示二叉树时，使用图 2 所示的结点结构构成的二叉链表，被称为线索链表；构建的二叉树称为线索二叉树。

> 线索链表中的“线索”，指的是链表中指向结点前趋和后继的指针。二叉树经过某种遍历方法转化为线索二叉树的过程称为线索化。

对二叉树进行线索化

将二叉树转化为线索二叉树，实质上是在遍历二叉树的过程中，将二叉链表中的空指针改为指向直接前趋或者直接后继的线索。

线索化的过程即为在遍历的过程中修改空指针的过程。

在遍历过程中，如果当前结点没有左孩子，需要将该结点的 lchild 指针指向遍历过程中的前一个结点，所以在遍历过程中，设置一个指针（名为 pre ），时刻指向当前访问结点的前一个结点。

代码实现（拿中序遍历为例）：

```
//中序对二叉树进行线索化
void InThreading(BiThrTree p){
    //如果当前结点存在
    if (p) {
        InThreading(p->lchild);//递归当前结点的左子树，进行线索化
        //如果当前结点没有左孩子，左标志位设为 1，左指针域指向上一结点 pre
        if (!p->lchild) {
            p->Ltag=Thread;
            p->lchild=pre;
        }
        //如果 pre 没有右孩子，右标志位设为 1，右指针域指向当前结点。
        if (!pre->rchild) {
            pre->Rtag=Thread;
            pre->rchild=p;
        }
        pre=p;//线索化完左子树后，让 pre 指针指向当前结点
        InThreading(p->rchild);//递归右子树进行线索化
    }
}
```

注意：中序对二叉树进行线索化的过程中，在两个递归函数中间的运行程序，和之前介绍的中序遍历二叉树的输出函数的作用是相同的。

将中间函数移动到两个递归函数之前，就变成了前序对二叉树进行线索化的过程；后序线索化同样如此。

使用线索二叉树遍历

图 3 中是一个按照中序遍历建立的线索二叉树。其中，实线表示指针，指向的是左孩子或者右孩子。虚线表示线索，指向的是该结点的直接前趋或者直接后继。

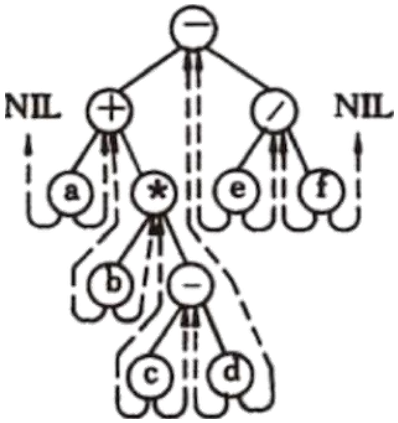


图 3 线索二叉树

使用线索二叉树时，会经常遇到一个问题，如图 3 中，结点 b 的直接后继直接通过指针域获得，为结点 * ；而由于结点 * 的度为 2 ，无法利用指针域指向后继结点，整个链表断掉了。当在遍历过程，遇到这种问题是解决的办法就是：寻找先序、中序、后序遍历的规律，找到下一个结点。

在先序遍历过程中，如果结点因为有右孩子导致无法找到其后继结点，如果结点有左孩子，则后继结点是其左孩子；否则，就一定是右孩子。拿图 3 举例，结点 + 的后继结点是其左孩子结点 a ，如果结点 a 不存在的话，就是结点 * 。

在中序遍历过程中，结点的后继是遍历其右子树时访问的第一个结点，也就是右子树中位于最左下的结点。例如图 3 中结点 * ，后继结点为结点 c ，是其右子树中位于最左边的结点。反之，结点的前趋是左子树最后访问的那个结点。

后序遍历中找后继结点需要分为 3 种情况：

1. 如果该结点是二叉树的根，后继结点为空；
2. 如果该结点是父结点的右孩子（或者是左孩子，但是父结点没有右孩子），后继结点是父结点；
3. 如果该结点是父结点的左孩子，且父结点有右子树，后继结点为父结点的右子树在后序遍历列出的第一个结点。

使用后序遍历建立的线索二叉树，在真正使用过程中遇到链表的断点时，需要访问父结点，所以在初步建立二叉树时，宜采用三叉链表做存储结构。

遍历线索二叉树非递归代码实现：

```
//中序遍历线索二叉树
void InOrderThraverse_Thr(BiThrTree p)
{
    while(p)
    {
        //一直找左孩子，最后一个为中序序列中排第一的
        while(p->Ltag == Link){
            p = p->lchild;
        }
        printf("%c ", p->data);   //操作结点数据
        //当结点右标志位为 1 时，直接找到其后继结点
        while(p->Rtag == Thread && p->rchild !=NULL){
```

```
            p = p->rchild;
            printf("%c ", p->data);
        }
        //否则，按照中序遍历的规律，找其右子树中最左下的结点，也就是继续循环遍历
        p = p->rchild;
    }
}
```

整节完整代码（可运行）

```
#include <stdio.h>
#include <stdlib.h>
#define TElemType char//宏定义，结点中数据域的类型
//枚举，Link 为 0，Thread 为 1
typedef enum {
    Link,
    Thread
}PointerTag;
//结点结构构造
typedef struct BiThrNode{
    TElemType data;//数据域
    struct BiThrNode* lchild,*rchild;//左孩子，右孩子指针域
    PointerTag Ltag,Rtag;//标志域，枚举类型
}BiThrNode,*BiThrTree;
BiThrTree pre=NULL;
//采用前序初始化二叉树
//中序和后序只需改变赋值语句的位置即可
void CreateTree(BiThrTree * tree){
    char data;
    scanf("%c",&data);
    if (data!='#'){
        if (!((*tree)=(BiThrNode*)malloc(sizeof(BiThrNode)))){
            printf("申请结点空间失败");
            return;
        }else{
            (*tree)->data=data;//采用前序遍历方式初始化二叉树
            CreateTree(&((*tree)->lchild));//初始化左子树
            CreateTree(&((*tree)->rchild));//初始化右子树
```

```
        }
    }else{
        *tree=NULL;
    }
}
//中序对二叉树进行线索化
void InThreading(BiThrTree p){
    //如果当前结点存在
    if (p) {
        InThreading(p->lchild);//递归当前结点的左子树，进行线索化
        //如果当前结点没有左孩子，左标志位设为 1，左指针域指向上一结点 pre
        if (!p->lchild) {
            p->Ltag=Thread;
            p->lchild=pre;
        }
        //如果 pre 没有右孩子，右标志位设为 1，右指针域指向当前结点。
        if (pre&&!pre->rchild) {
            pre->Rtag=Thread;
            pre->rchild=p;
        }
        pre=p;//pre 指向当前结点
        InThreading(p->rchild);//递归右子树进行线索化
    }
}
//中序遍历线索二叉树
void InOrderThraverse_Thr(BiThrTree p)
{
    while(p)
    {
        //一直找左孩子，最后一个为中序序列中排第一的
        while(p->Ltag == Link){
            p = p->lchild;
        }
        printf("%c ", p->data);   //操作结点数据
        //当结点右标志位为 1 时，直接找到其后继结点
        while(p->Rtag == Thread && p->rchild !=NULL)
        {
```

```
            p = p->rchild;
            printf("%c ", p->data);
        }
        //否则，按照中序遍历的规律，找其右子树中最左下的结点，也就是继续循环遍历
        p = p->rchild;
    }
}
int main() {
    BiThrTree t;
    printf("输入前序二叉树:\n");
    CreateTree(&t);
    InThreading(t);
    printf("输出中序序列:\n");
    InOrderThraverse_Thr(t);
    return 0;
}
```

运行结果

```
输入前序二叉树:
124###35##6##
输出中序序列:
4 2 1 5 3 6
```

双向线索二叉树的建立及 C 语言实现

通过前一节对线索二叉树的学习，其中，在遍历使用中序序列创建的线索二叉树时，对于其中的每个结点，即使没有线索的帮助下，也可以通过中序遍历的规律找到直接前趋和直接后继结点的位置。

也就是说，建立的线索二叉链表可以从两个方向对结点进行中序遍历。通过前一节的学习，线索二叉链表可以从第一个结点往后逐个遍历。但是起初由于没有记录中序序列中最后一个结点的位置，所以不能实现从最后一个结点往前逐个遍历。

双向线索链表的作用就是可以让线索二叉树从两个方向实现遍历。

双向线索二叉树的实现过程

在线索二叉树的基础上，额外添加一个结点。此结点的作用类似于链表中的头指针，数据域不起作用，只利用两个指针域（由于都是指针，标志域都为 0 ）。

左指针域指向二叉树的树根，确保可以正方向对二叉树进行遍历；同时，右指针指向线索二叉树形成的线性序列中的最后一个结点。

这样，二叉树中的线索链表就变成了双向线索链表，既可以从第一个结点通过不断地找后继结点进行遍历，也可以从最后一个结点通过不断找前趋结点进行遍历。

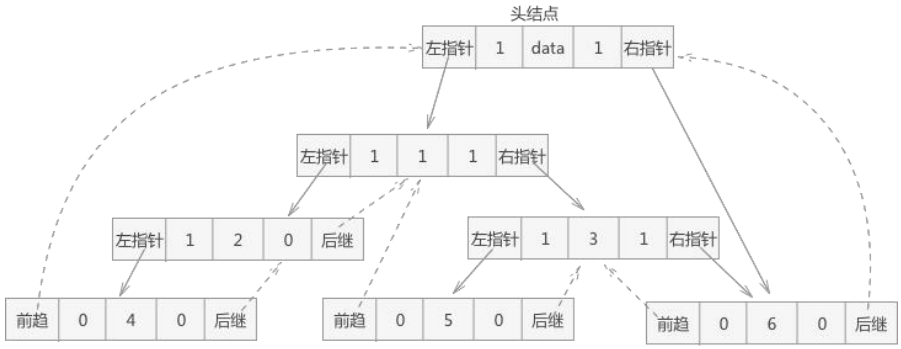


图 1 双向线索二叉链表

代码实现：

```
//建立双向线索链表
void InOrderThread_Head(BiThrTree *h, BiThrTree t)
{
    //初始化头结点
    (*h) = (BiThrTree)malloc(sizeof(BiThrNode));
    if((*h) == NULL){
        printf("申请内存失败");
        return ;
    }
    (*h)->rchild = *h;
    (*h)->Rtag = Link;
    //如果树本身是空树
    if(!t){
        (*h)->lchild = *h;
        (*h)->Ltag = Link;
    }
    else{
        pre = *h;//pre 指向头结点
        (*h)->lchild = t;//头结点左孩子设为树根结点
        (*h)->Ltag = Link;
        InThreading(t);//线索化二叉树，pre 结点作为全局变量，线索化结束后，pre 结点指向中序序列中最后一个结点
        pre->rchild = *h;
        pre->Rtag = Thread;
        (*h)->rchild = pre;
    }
```

```
}
```

双向线索二叉树的遍历

双向线索二叉树遍历时，如果正向遍历，就从树的根结点开始。整个遍历过程结束的标志是：当从头结点出发，遍历回头结点时，表示遍历结束。

```
//中序正向遍历双向线索二叉树
void InOrderThraverse_Thr(BiThrTree h)
{
    BiThrTree p;
    p = h->lchild;                  //p 指向根结点
    while(p != h)
    {
        while(p->Ltag == Link)     //当 ltag = 0 时循环到中序序列的第一个结点
        {
            p = p->lchild;
        }
        printf("%c ", p->data);   //显示结点数据，可以更改为其他对结点的操作
        while(p->Rtag == Thread && p->rchild != h)
        {
            p = p->rchild;
            printf("%c ", p->data);
        }

        p = p->rchild;                  //p 进入其右子树
    }
}
```

逆向遍历线索二叉树的过程即从头结点的右指针指向的结点出发，逐个寻找直接前趋结点，结束标志同正向遍历一样：

```
//中序逆方向遍历线索二叉树
void InOrderThraverse_Thr(BiThrTree h){
    BiThrTree p;
    p=h->rchild;
    while (p!=h) {
        while (p->Rtag==Link) {
            p=p->rchild;
        }
```

```
            printf("%c",p->data);
            //如果 lchild 为线索，直接使用，输出
            while (p->Ltag==Thread && p->lchild !=h) {
                p=p->lchild;
                printf("%c",p->data);
            }
            p=p->lchild;
        }
}
```

完整代码实现

```
#include <stdio.h>
#include <stdlib.h>
#define TElemType char//宏定义，结点中数据域的类型
//枚举，Link 为 0，Thread 为 1
typedef enum {
    Link,
    Thread
}PointerTag;
//结点结构构造
typedef struct BiThrNode{
    TElemType data;//数据域
    struct BiThrNode* lchild,*rchild;//左孩子，右孩子指针域
    PointerTag Ltag,Rtag;//标志域，枚举类型
}BiThrNode,*BiThrTree;

BiThrTree pre=NULL;

//采用前序初始化二叉树
//中序和后序只需改变赋值语句的位置即可
void CreateTree(BiThrTree * tree){
    char data;
    scanf("%c",&data);
    if (data!='#'){
        if (!((*tree)=(BiThrNode*)malloc(sizeof(BiThrNode)))){
            printf("申请结点空间失败");
            return;
```

```
        }else{
            (*tree)->data=data;//采用前序遍历方式初始化二叉树
            CreateTree(&((*tree)->lchild));//初始化左子树
            CreateTree(&((*tree)->rchild));//初始化右子树
        }
    }else{
        *tree=NULL;
    }
}
//中序对二叉树进行线索化
void InThreading(BiThrTree p){
    //如果当前结点存在
    if (p) {
        InThreading(p->lchild);//递归当前结点的左子树，进行线索化
        //如果当前结点没有左孩子，左标志位设为 1，左指针域指向上一结点 pre
        if (!p->lchild) {
            p->Ltag=Thread;
            p->lchild=pre;
        }
        //如果 pre 没有右孩子，右标志位设为 1，右指针域指向当前结点。
        if (pre&&!pre->rchild) {
            pre->Rtag=Thread;
            pre->rchild=p;
        }
        pre=p;//pre 指向当前结点
        InThreading(p->rchild);//递归右子树进行线索化
    }
}
//建立双向线索链表
void InOrderThread_Head(BiThrTree *h, BiThrTree t)
{
    //初始化头结点
    (*h) = (BiThrTree)malloc(sizeof(BiThrNode));
    if((*h) == NULL){
        printf("申请内存失败");
        return ;
    }
```

```
    (*h)->rchild = *h;
    (*h)->Rtag = Link;
    //如果树本身是空树
    if(!t){
        (*h)->lchild = *h;
        (*h)->Ltag = Link;
    }
    else{
        pre = *h;//pre 指向头结点
        (*h)->lchild = t;//头结点左孩子设为树根结点
        (*h)->Ltag = Link;
        InThreading(t);//线索化二叉树，pre 结点作为全局变量，线索化结束后，pre 结点指向中序序列中最后一个结点
        pre->rchild = *h;
        pre->Rtag = Thread;
        (*h)->rchild = pre;
    }
}
//中序正向遍历双向线索二叉树
void InOrderThraverse_Thr(BiThrTree h)
{
    BiThrTree p;
    p = h->lchild;                  //p 指向根结点
    while(p != h)
    {
        while(p->Ltag == Link)     //当 ltag = 0 时循环到中序序列的第一个结点
        {
            p = p->lchild;
        }
        printf("%c ", p->data);   //显示结点数据，可以更改为其他对结点的操作
        while(p->Rtag == Thread && p->rchild != h)
        {
            p = p->rchild;
            printf("%c ", p->data);
        }

        p = p->rchild;                  //p 进入其右子树
    }
```

```
}
int main() {
    BiThrTree t;
    BiThrTree h;
    printf("输入前序二叉树:\n");
    CreateTree(&t);
    InOrderThread_Head(&h, t);
    printf("输出中序序列:\n");
    InOrderThraverse_Thr(h);
    return 0;
}
```

运行结果：

```
输入前序二叉树:
124###35##6##
输出中序序列:
4 2 1 5 3 6
```

程序中只调用了正向遍历线索二叉树的代码，如果逆向遍历，直接替换逆向遍历的函数代码到程序中即可。

森林转化为二叉树（详解版）

前面介绍了普通树转化为二叉树的孩子兄弟表示法，本节来学习如何将森林转化为一整棵二叉树。

森林，指的是由 n（n>=2）棵互不相交的树组成的集合，如图 1 所示。

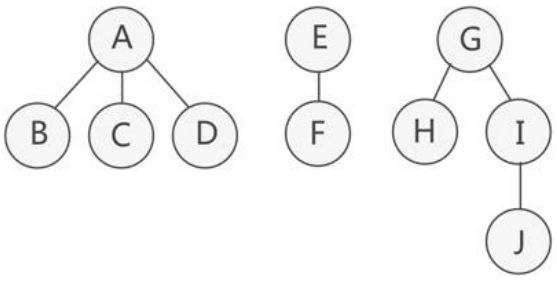


图 1 森林示意图

在某些实际场景中，为了便于操作具有森林结构的数据，往往需要将森林转化为一整棵二叉树。

我们知道，任意一棵普通树都可以转化为二叉树，而森林是由多棵普通树构成的，因此自然也可以转化为二叉树，其转化方法是：

1. 首先将森林中所有的普通树各自转化为二叉树；
2. 将森林中第一棵树的树根作为整个森林的树根，其他树的根节点看作是第一棵树根节点的兄弟节点，采用孩子兄弟表示法将所有树进行连接；

例如，将图 2a）中的森林转化为二叉树，则以上两个转化过程分别对应图 2 中的 b）和 c）：

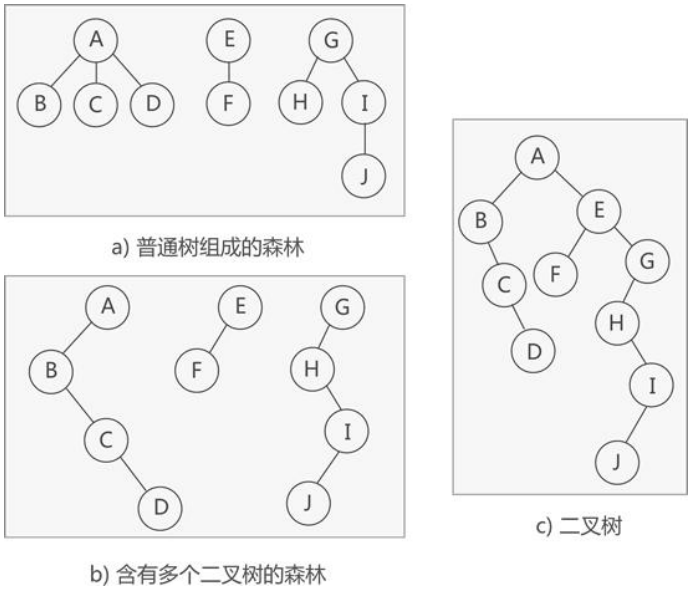


图 2 森林转化为二叉树的过程示意图

如图 2 所示，先将森林包含的所有普通树各自转化为二叉树，然后将其他树的根节点看作为第一棵二叉树的兄弟节点，采用孩子兄弟表示法进行连接。

森林转化为二叉树，更多的是为了对森林中的节点做遍历操作。前面讲过，遍历二叉树有 4 种方法，分别是层次遍历、先序遍历、中序遍历和后序遍历。转化前的森林与转化后的二叉树相比，其层次遍历和后序遍历的访问节点顺序不同，而前序遍历和中序遍历访问节点的顺序是相同的。

以图 1 中的森林为例，其转化后的二叉树为图 2c)，两者比较，其先序遍历访问节点的顺序都是 `A B C D E F G H I J`；同样，中序遍历访问节点的顺序也相同，都是 `B C D A F E H J I G`。而后序遍历和层次遍历访问节点的顺序是不同的。

提示，由二叉树转化为森林的过程也就是森林转化二叉树的逆过程，也就是图 2 中由 c) 到 b) 再到 a) 的过程。

数据结构实践项目之移动迷宫小游戏(初级版)

《移动迷宫》游戏简介：迷宫只有两个门，一个入口，一个出口。一个骑士骑马从入口走进迷宫，迷宫中设置有很多墙壁，对前进方向造成障碍。骑士需要在迷宫中寻找通路以到达出口。

本游戏的迷宫是“移动”的，每次骑士进入迷宫时，迷宫的入口、出口，甚至是迷宫中设置的障碍都是不同的。

设计思路

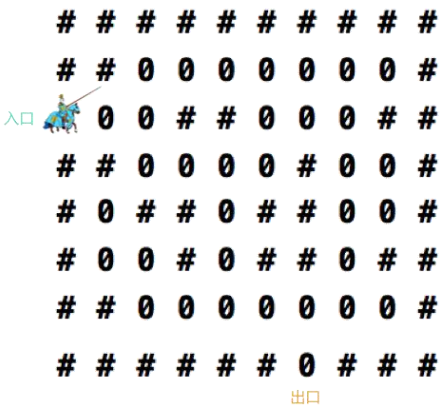
解决类似的问题，使用回溯法是最行之有效的解题方法。骑士从入口开始，不断地对周围的道路进行试探：若能走通，则进入该位置，继续对周围进行试探；反之，则后退一步，继续寻求其他的可行路径。

通过不停地对可行道路进行试探，结果有两种：

- 骑士最终找到了一条通往出口的道路；
- 试探结束，没有通往出口的道路，骑士最终只能被迫返回入口，继续等到迷宫的下一次变化（程序结束）。

实例分析

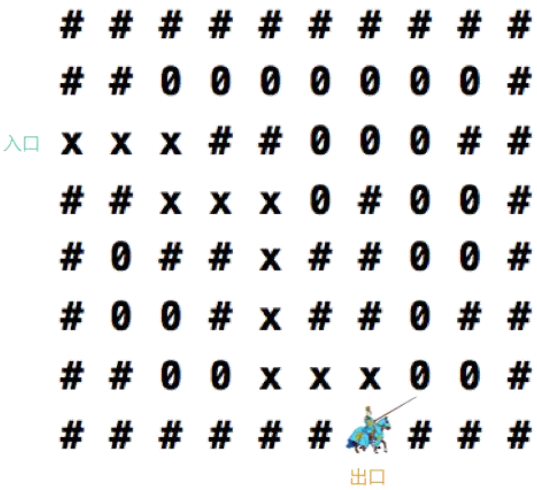
假设迷宫为一块长为 10 ，宽为 8 的矩形区域，其中随机设置了入口、出口和该区域内可供通行的道路，如下图所示：



提示：迷宫中，‘0’ 表示道路，‘#’ 表示障碍。

当骑士处于入口的位置时，他会前后左右的进行探索式前进，当他发现前方道路可行时，即坐标为（2，1）的通路，此时骑士会快速移动至该位置，进行以该位置为中心的再次探索式前进。

通过骑士不断地探索，对于该实例中列举的迷宫，骑士最终可以找到一条通往出口的道路，如下图所示：



提示：迷宫中，新增的‘X’表示骑士走过的道路（找出一条通路即可）。

完整实现代码

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef enum {false,true} bool;

//迷宫本身是一个 8 行 10 列的矩形

int ROWS=8;

int COLS=10;

//初始化迷宫，随机设置迷宫出入口，同时在迷宫中随机设置可行道路。

void mazeGenerator(char [][COLS],int *,int *,int *,int *);
```

```
//使用回溯法从入口处不断地尝试找到出口的路径
void mazeTraversal(char maze[ROWS][COLS],int row,int col,int entryRow,int entryCol,int exitrow,int exitcol);
//迷宫的输出函数
void printMaze(const char[][COLS]);
//判断每一次移动是否有效
bool validMove(const char [][COLS],int,int);

int main()
{
    printf("*********移动迷宫小项目（数据结构就该这么学）*********\n");
    char maze[ROWS][COLS];
    int xStart,yStart,x,y;
    srand(time(0));//种下随机种子数，每次运行种下不同的种子，后序通过 rand()函数获得的随机数就不同。
    //通过一个嵌套循环，先将迷宫中各个地方设置为死路('#'表示为墙，表示此处不可通过)
    for(int loop=0;loop<ROWS;++loop ){
        for(int loop2=0;loop2<COLS;++loop2 ){
            maze[loop][loop2]='#';
        }
    }
    //初始化迷宫，即在迷宫中随机设置出口、入口和中间的道路，用'0'表示。通过此函数，可同时得到入口的坐标
    mazeGenerator(maze,&xStart,&yStart,&x,&y);

    printf("迷宫入口位置坐标为(%d,%d);出口位置坐标为：(%d，%d);\n",xStart+1,yStart+1,x+1,y+1);
    printf("迷宫设置如下（'#'表示墙，'0'表示通路）：\n");
    printMaze(maze);//输出一个初始化好的迷宫
    //使用回溯法，通过不断地进行尝试，试图找到一条通往出口的路。
    mazeTraversal(maze,xStart,yStart,xStart,yStart,x,y);
}
//由于迷宫整体布局为矩形，有四条边，在初始化迷宫的出口和入口时，随机选择不同的两条边作为设置出口和入口的边
void mazeGenerator(char maze[][COLS],int *xPtr,int *yPtr,int *exitx,int *exity){
    int a,x,y,entry,exit;
    do {
        entry=rand()%4;
        exit=rand()%4;
    }while(entry==exit);
    // 确定入口位置，0 代表选择的为左侧的边，1 代表为上边，2 代表为右侧的边，3 代表为下边
    if(entry==0){
```

```
            *xPtr=1+rand()%(ROWS-2);
            *yPtr=0;
            maze[*xPtr][*yPtr]='0';
    }else if(entry==1){
            *xPtr=0;
            *yPtr=1+rand()%(COLS-2);
            maze[*xPtr][*yPtr]='0';
    }else if(entry==2){
            *xPtr=1+rand()%(ROWS-2);
            *yPtr=COLS-1;
            maze[*xPtr][*yPtr]='0';
    }else{
            *xPtr=ROWS-1;
            *yPtr=1+rand()%(COLS-2);
            maze[*xPtr][*yPtr]='0';
    }
    //确定出口位置
    if(exit==0){
            a=1+rand()%(ROWS-2);
            *exitx=a;
            *exity=0;
            maze[a][0]='0';}
    else if(exit==1){
            a=1+rand()%(COLS-2);
            *exitx=0;
            *exity=a;
            maze[0][a]='0';}
    else if(exit==2){
            a=1+rand()%(ROWS-2);
            *exitx=a;
            *exity=COLS-1;
            maze[a][COLS-1]='0';}
    else{
            a=1+rand()%(COLS-2);
            *exitx=ROWS-1;
            *exity=a;
            maze[ROWS-1][a]='0';
```

```
    }
    //在迷宫中央设置多出不同的随机通路
    for(int loop=1;loop<(ROWS-2)*(COLS-2);++loop) {
        x=1+rand()%(ROWS-2);
        y=1+rand()%(COLS-2);
        maze[x][y]='0';}
}

void mazeTraversal(char maze[ROWS][COLS],int row,int col,int entryRow,int entryCol,int exitrow,int exitcol){
    //由于从入口处进入，为了区分走过的通路和没走过的通路，将走过的通路设置为‘x’，
    maze[row][col]='x';
    static bool judge=false;//设置一个判断变量，判断在入口位置是否有通路存在。
    static int succ=0;//用于统计从入口到出口的可行通路的条数
    if (row==exitrow && col==exitcol) {
        printf("成功走出迷宫，道路图如下：\n");
        printMaze(maze);
        succ++;
        return;
    }
    //判断当前位置的下方是否为通路
    if (validMove(maze, row+1, col)) {
        judge=true;//证明起码有路存在，下面证明是否有可通往出口的路
        mazeTraversal(maze, row+1, col,entryRow,entryCol,exitrow,exitcol);//以下方的位置为起点继续尝试
    }
    //判断当前位置的右侧是否为通路
    if (validMove(maze, row, col+1)) {
        judge=true;
        mazeTraversal(maze, row, col+1,entryRow,entryCol,exitrow,exitcol);
    }
    //判断当前位置的上方是否为通路
    if (validMove(maze, row-1, col)) {
        judge=true;
        mazeTraversal(maze, row-1, col,entryRow,entryCol,exitrow,exitcol);
    }
    //判断当前位置的左侧是否为通路
    if (validMove(maze, row, col-1)) {
        judge=true;
```

```
            mazeTraversal(maze, row, col-1,entryRow,entryCol,exitrow,exitcol);
        }
        //如果 judge 仍为假，说明在入口处全部被墙包围，无路可走
        if (judge==false) {
            printf("入口被封死，根本无路可走！\n");
            printMaze(maze);
        }
        //如果 judge 为真，但是 succ 值为 0，且最终又回到了入口的位置，证明所有的尝试工作都已完成，但是没有发现通往出口的路
        else if(judge==true && row==entryRow && col==entryCol && succ==0){
            printf("尝试了所有道路，出口和入口之间没有通路！\n");
            printMaze(maze);
        }
}
//有效移动，即证明该位置处于整个迷宫的矩形范围内，且该位置是通路，不是墙，也从未走过
bool validMove(const char maze[][COLS],int r,int c){
    return(r>=0&&r<=ROWS-1&&c>=0&&c<=COLS-1&&maze[r][c]!='#'&& maze[r][c]!='x');
}
//输出迷宫
void printMaze(const char maze[][COLS] ){
    for(int x=0;x<ROWS;++x){
        for(int y=0;y<COLS;++y){
            printf("%c ",maze[x][y]);
        }
        printf("\n");
    }
    printf("\n");
}
```

该程序由于每次运行产生不同的迷宫，所以每次运行结果不同，可自行运行，查看结果，这里不再进行描述。

总结

通过练习《移动迷宫》小游戏，旨在让大家熟悉回溯法的解题思路。

回溯 PK 递归回忆：比如说你在面对一个二叉路口，不知道要走哪条，此时就要做尝试（尝试这个动作就是一个函数）你选择先尝试左边这条，往左边走，走着走着发现又有一个二叉路口，此时你需要上一次尝试的过程中要再做一次尝试，即在函数内再调用一次函数，这是递归。但是如果你发现这条路走不通，就知道上一个二岔路你选择错了，此时你回到原来的岔路口选择右边，这就是回溯（回溯使用递归的思想实现的一种算法结构）。

普里姆算法(Prim 算法)求最小生成树

通过前面的学习，对于含有 n 个顶点的连通图来说可能包含有多种生成树，例如图 1 所示：

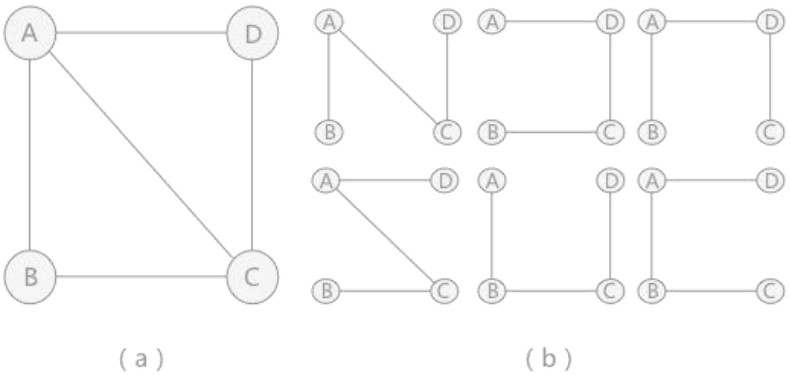


图 1 连通图的生成树

图 1 中的连通图和它相对应的生成树，可以用于解决实际生活中的问题：假设 A、B、C 和 D 为 4 座城市，为了方便生产生活，要为这 4 座城市建立通信。对于 4 个城市来讲，本着节约经费的原则，只需要建立 3 个通信线路即可，就如图 1（b）中的任意一种方式。

在具体选择采用（b）中哪一种方式时，需要综合考虑城市之间间隔的距离，建设通信线路的难度等各种因素，将这些因素综合起来用一个数值表示，当作这条线路的权值。

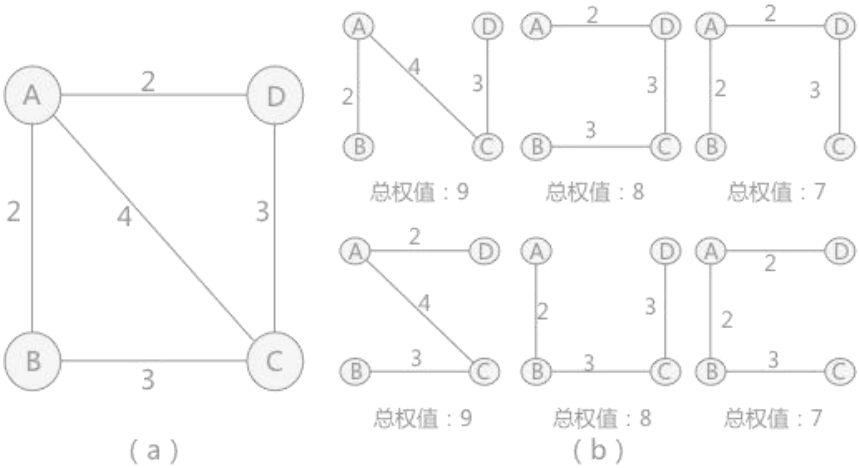


图 2 无向网

假设通过综合分析，城市之间的权值如图 2（a）所示，对于（b）的方案中，选择权值总和为 7 的两种方案最节约经费。

这就是本节要讨论的最小生成树的问题，简单得理解就是给定一个带有权值的连通图（连通网），如何从众多的生成树中筛选出权值总和最小的生成树，即为该图的最小生成树。

给定一个连通网，求最小生成树的方法有：普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法。

普里姆算法

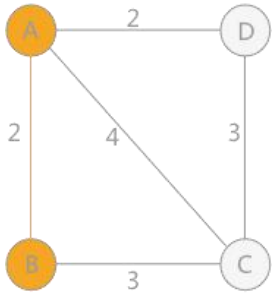
普里姆算法在找最小生成树时，将顶点分为两类，一类是在查找的过程中已经包含在树中的（假设为 A 类），剩下的是另一类（假设为 B 类）。

对于给定的连通网，起始状态全部顶点都归为 B 类。在找最小生成树时，选定任意一个顶点作为起始点，并将之从 B 类移至 A

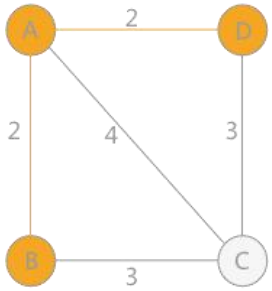
类；然后找出 B 类中到 A 类中的顶点之间权值最小的顶点，将之从 B 类移至 A 类，如此重复，直到 B 类中没有顶点为止。所走过的顶点和边就是该连通图的最小生成树。

例如，通过普里姆算法查找图 2（a）的最小生成树的步骤为：

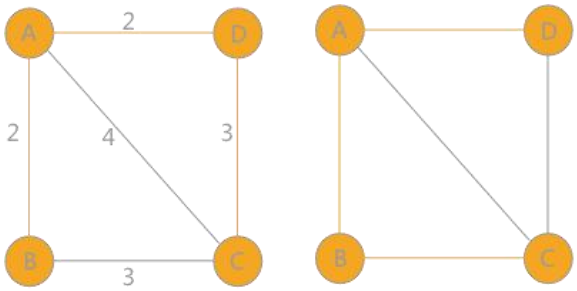
假如从顶点 A 出发，顶点 B、C、D 到顶点 A 的权值分别为 2、4、2，所以，对于顶点 A 来说，顶点 B 和顶点 D 到 A 的权值最小，假设先找到的顶点 B：



继续分析顶点 C 和 D，顶点 C 到 B 的权值为 3，到 A 的权值为 4；顶点 D 到 A 的权值为 2，到 B 的权值为无穷大（如果之间没有直接通路，设定权值为无穷大）。所以顶点 D 到 A 的权值最小：



最后，只剩下顶点 C，到 A 的权值为 4，到 B 的权值和到 D 的权值一样大，为 3。所以该连通图有两个最小生成树：



具体实现代码为：

```
#include <stdio.h>

#include <stdlib.h>

#define VertexType int

#define VRType int

#define MAX_VERtEX_NUM 20
```

```
#define InfoType char
#define INFINITY 65535
typedef struct {
    VRType adj;                                 //对于无权图，用 1 或 0 表示是否相邻；对于带权图，直接为权值。
    InfoType * info;                            //弧额外含有的信息指针
}ArcCell,AdjMatrix[MAX_VERtEX_NUM][MAX_VERtEX_NUM];

typedef struct {
    VertexType vexs[MAX_VERtEX_NUM];            //存储图中顶点数据
    AdjMatrix arcs;                             //二维数组，记录顶点之间的关系
    int vexnum,arcnum;                          //记录图的顶点数和弧（边）数
}MGraph;

//根据顶点本身数据，判断出顶点在二维数组中的位置
int LocateVex(MGraph G,VertexType v){
    int i=0;
    //遍历一维数组，找到变量 v
    for (; i<G.vexnum; i++) {
        if (G.vexs[i]==v) {
            return i;
        }
    }
    return -1;
}
//构造无向网
void CreateUDN(MGraph* G){
    scanf("%d,%d",&(G->vexnum),&(G->arcnum));
    for (int i=0; i<G->vexnum; i++) {
        scanf("%d",&(G->vexs[i]));
    }
    for (int i=0; i<G->vexnum; i++) {
        for (int j=0; j<G->vexnum; j++) {
            G->arcs[i][j].adj=INFINITY;
            G->arcs[i][j].info=NULL;
        }
    }
    for (int i=0; i<G->arcnum; i++) {
```

```
            int v1,v2,w;
            scanf("%d,%d,%d",&v1,&v2,&w);
            int m=LocateVex(*G, v1);
            int n=LocateVex(*G, v2);
            if (m==-1 ||n==-1) {
                printf("no this vertex\n");
                return;
            }
            G->arcs[n][m].adj=w;
            G->arcs[m][n].adj=w;
        }
}
//辅助数组，用于每次筛选出权值最小的边的邻接点
typedef struct {
    VertexType adjvex;//记录权值最小的边的起始点
    VRType lowcost;//记录该边的权值
}closedge[MAX_VERtEX_NUM];
closedge theclose;//创建一个全局数组，因为每个函数中都会使用到
//在辅助数组中找出权值最小的边的数组下标，就可以间接找到此边的终点顶点。
int minimun(MGraph G,closedge close){
    int min=INFINITY;
    int min_i=-1;
    for (int i=0; i<G.vexnum; i++) {
        //权值为 0，说明顶点已经归入最小生成树中；然后每次和 min 变量进行比较，最后找出最小的。
        if (close[i].lowcost>0 && close[i].lowcost < min) {
            min=close[i].lowcost;
            min_i=i;
        }
    }
    //返回最小权值所在的数组下标
    return min_i;
}
//普里姆算法函数，G 为无向网，u 为在网中选择的任意顶点作为起始点
void miniSpanTreePrim(MGraph G,VertexType u){
    //找到该起始点在顶点数组中的位置下标
    int k=LocateVex(G, u);
```

```
    //首先将与该起始点相关的所有边的信息：边的起始点和权值，存入辅助数组中相应的位置，例如（1，2）边，adjvex 为 0，lowcost 为 6，存入 theclose[1]中，辅助数组的下标表示该边的顶点 2
    for (int i=0; i<G.vexnum; i++) {
        if (i !=k) {
            theclose[i].adjvex=k;
            theclose[i].lowcost=G.arcs[k][i].adj;
        }
    }
    //由于起始点已经归为最小生成树，所以辅助数组对应位置的权值为 0，这样，遍历时就不会被选中
    theclose[k].lowcost=0;
    //选择下一个点，并更新辅助数组中的信息
    for (int i=1; i<G.vexnum; i++) {
        //找出权值最小的边所在数组下标
        k=minimun(G, theclose);
        //输出选择的路径
        printf("v%d v%d\n",G.vexs[theclose[k].adjvex],G.vexs[k]);
        //归入最小生成树的顶点的辅助数组中的权值设为 0
        theclose[k].lowcost=0;
        //信息辅助数组中存储的信息，由于此时树中新加入了一个顶点，需要判断，由此顶点出发，到达其它各顶点的权值是否比之前记录的权值还要小，如果还小，则更新
        for (int j=0; j<G.vexnum; j++) {
            if (G.arcs[k][j].adj<theclose[j].lowcost) {
                theclose[j].adjvex=k;
                theclose[j].lowcost=G.arcs[k][j].adj;
            }
        }
    }
    printf("\n");
}
int main(){
    MGraph G;
    CreateUDN(&G);
    miniSpanTreePrim(G, 1);
}
```

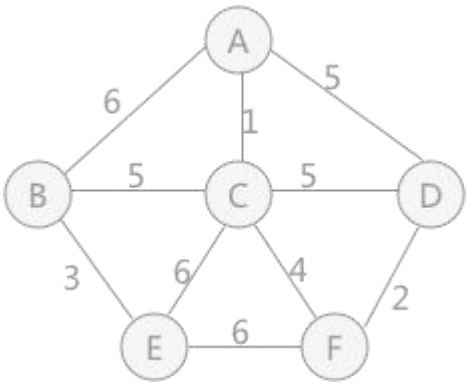


图 3 无向网

使用普里姆算法找图 3 所示无向网的最小生成树的测试数据为：

```
6,10
1
2
3
4
5
6
1,2,6
1,3,1
1,4,5
2,3,5
2,5,3
3,4,5
3,5,6
3,6,4
4,6,2
5,6,6
运行结果为：
v1 v3
v3 v6
v6 v4
v3 v2
v2 v5
```

普里姆算法的运行效率只与连通网中包含的顶点数相关，而和网所含的边数无关。所以普里姆算法适合于解决边稠密的网，该算法运行的时间复杂度为：$O(n^2)$。

如果连通网中所含边的绸密度不高，则建议使用克鲁斯卡尔算法求最小生成树（下节详细介绍）。

克鲁斯卡尔算法(Kruskal 算法)求最小生成树

上一节介绍了求最小生成树之普里姆算法。该算法从顶点的角度为出发点，时间复杂度为 $O(n^2)$，更适合与解决边的绸密度更高的连通网。

本节所介绍的克鲁斯卡尔算法，从边的角度求网的最小生成树，时间复杂度为 O(eloge)。和普里姆算法恰恰相反，更适合于求边稀疏的网的最小生成树。

对于任意一个连通网的最小生成树来说，在要求总的权值最小的情况下，最直接的想法就是将连通网中的所有边按照权值大小进行升序排序，从小到大依次选择。

由于最小生成树本身是一棵生成树，所以需要时刻满足以下两点：

- 生成树中任意顶点之间有且仅有一条通路，也就是说，生成树中不能存在回路；
- 对于具有 n 个顶点的连通网，其生成树中只能有 n-1 条边，这 n-1 条边连通着 n 个顶点。

连接 n 个顶点在不产生回路的情况下，只需要 n-1 条边。

所以克鲁斯卡尔算法的具体思路是：将所有边按照权值的大小进行升序排序，然后从小到大一一判断，条件为：如果这个边不会与之前选择的所有边组成回路，就可以作为最小生成树的一部分；反之，舍去。直到具有 n 个顶点的连通网筛选出来 n-1 条边为止。筛选出来的边和所有的顶点构成此连通网的最小生成树。

判断是否会产生回路的方法为：在初始状态下给每个顶点赋予不同的标记，对于遍历过程的每条边，其都有两个顶点，判断这两个顶点的标记是否一致，如果一致，说明它们本身就处在一棵树中，如果继续连接就会产生回路；如果不一致，说明它们之间还没有任何关系，可以连接。

假设遍历到一条由顶点 A 和 B 构成的边，而顶点 A 和顶点 B 标记不同，此时不仅需要将顶点 A 的标记更新为顶点 B 的标记，还需要更改所有和顶点 A 标记相同的顶点的标记，全部改为顶点 B 的标记。

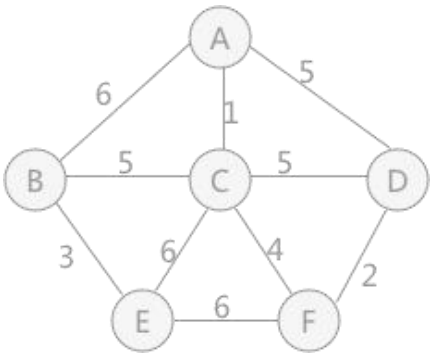
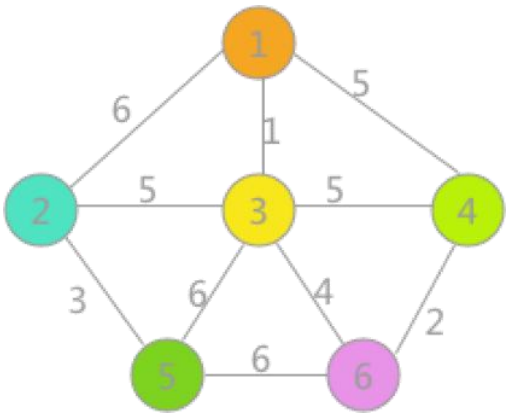


图 1 连通网

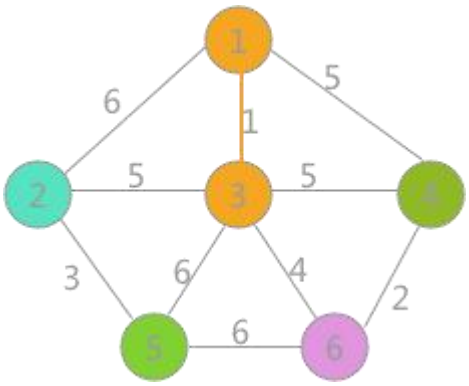
例如，使用克鲁斯卡尔算法找图 1 的最小生成树的过程为：

首先，在初始状态下，对各顶点赋予不同的标记（用颜色区别），如下图所示：



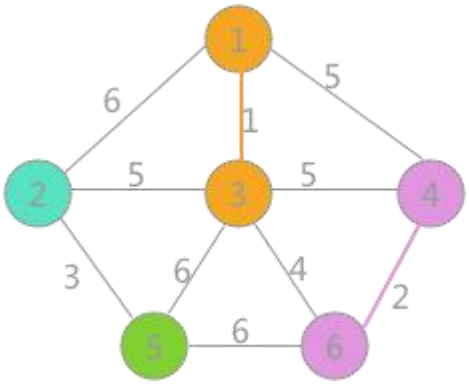
(1)

对所有边按照权值的大小进行排序，按照从小到大的顺序进行判断，首先是（1，3），由于顶点 1 和顶点 3 标记不同，所以可以构成生成树的一部分，遍历所有顶点，将与顶点 3 标记相同的全部更改为顶点 1 的标记，如（2）所示：



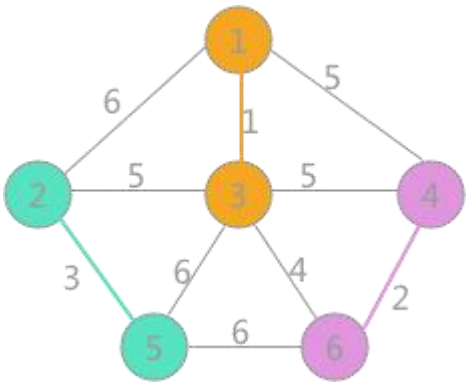
(2)

其次是（4，6）边，两顶点标记不同，所以可以构成生成树的一部分，更新所有顶点的标记为：



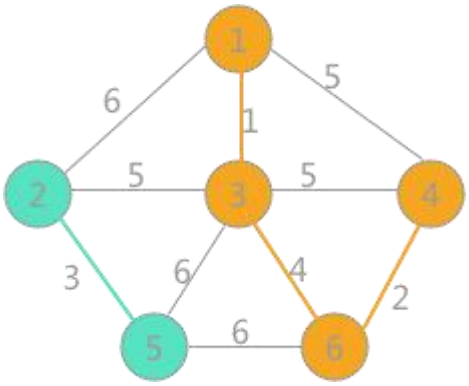
(3)

其次是（2，5）边，两顶点标记不同，可以构成生成树的一部分，更新所有顶点的标记为：



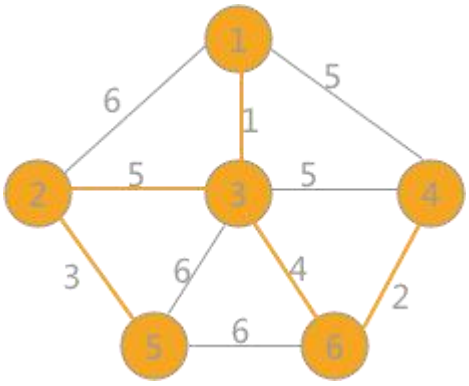
(4)

然后最小的是（3，6）边，两者标记不同，可以连接，遍历所有顶点，将与顶点 6 标记相同的所有顶点的标记更改为顶点 1 的标记：



(5)

继续选择权值最小的边，此时会发现，权值为 5 的边有 3 个，其中（1，4）和（3，4）各自两顶点的标记一样，如果连接会产生回路，所以舍去，而（2，3）标记不一样，可以选择，将所有与顶点 2 标记相同的顶点的标记全部改为同顶点 3 相同的标记：



(6)

当选取的边的数量相比与顶点的数量小 1 时，说明最小生成树已经生成。所以最终采用克鲁斯卡尔算法得到的最小生成树为 (6) 所示。

实现代码：

```
#include "stdio.h"
#include "stdlib.h"
#define MAX_VERtEX_NUM 20
#define VertexType int
typedef struct edge{
    VertexType initial;
    VertexType end;
    VertexType weight;
}edge[MAX_VERtEX_NUM];
//定义辅助数组
typedef struct {
    VertexType value;//顶点数据
    int sign;//每个顶点所属的集合
}assist[MAX_VERtEX_NUM];

assist assists;

//qsort 排序函数中使用，使 edges 结构体中的边按照权值大小升序排序
int cmp(const void *a,const void*b){
    return   ((struct edge*)a)->weight-((struct edge*)b)->weight;
}
//初始化连通网
void CreateUDN(edge *edges,int *vexnum,int *arcnum){
    printf("输入连通网的边数：\n");
    scanf("%d %d",&(*vexnum),&(*arcnum));
    printf("输入连通网的顶点：\n");
    for (int i=0; i<(*vexnum); i++) {
        scanf("%d",&(assists[i].value));
        assists[i].sign=i;
    }
    printf("输入各边的起始点和终点及权重：\n");
    for (int i=0 ; i<(*arcnum); i++) {
        scanf("%d,%d,%d",&(*edges)[i].initial,&(*edges)[i].end,&(*edges)[i].weight);
    }
}
```

```
//在 assists 数组中找到顶点 point 对应的位置下标
int Locatevex(int vexnum,int point){
    for (int i=0; i<vexnum; i++) {
        if (assists[i].value==point) {
            return i;
        }
    }
    return -1;
}
int main(){

    int arcnum,vexnum;
    edge edges;
    CreateUDN(&edges,&vexnum,&arcnum);
    //对连通网中的所有边进行升序排序，结果仍保存在 edges 数组中
    qsort(edges, arcnum, sizeof(edges[0]), cmp);
    //创建一个空的结构体数组，用于存放最小生成树
    edge minTree;
    //设置一个用于记录最小生成树中边的数量的常量
    int num=0;
    //遍历所有的边
    for (int i=0; i<arcnum; i++) {
        //找到边的起始顶点和结束顶点在数组 assists 中的位置
        int initial=Locatevex(vexnum, edges[i].initial);
        int end=Locatevex(vexnum, edges[i].end);
        //如果顶点位置存在且顶点的标记不同，说明不在一个集合中，不会产生回路
        if (initial!=-1&& end!=-1&&assists[initial].sign!=assists[end].sign) {
            //记录该边，作为最小生成树的组成部分
            minTree[num]=edges[i];
            //计数+1
            num++;
            //将新加入生成树的顶点标记全不更改为一样的
            for (int k=0; k<vexnum; k++) {
                if (assists[k].sign==assists[end].sign) {
                    assists[k].sign=assists[initial].sign;
                }
            }
```

```
            //如果选择的边的数量和顶点数相差 1，证明最小生成树已经形成，退出循环
            if (num==vexnum-1) {
                break;
            }
        }
    }
    //输出语句
    for (int i=0; i<vexnum-1; i++) {
        printf("%d,%d\n",minTree[i].initial,minTree[i].end);
    }
    return 0;
}
```

测试数据：

```
输入连通网的边数：
6 10
输入连通网的顶点：
1
2
3
4
5
6
输入各边的起始点和终点及权重：
1,2,6
1,3,1
1,4,5
2,3,5
2,5,3
3,4,5
3,5,6
3,6,4
4,6,2
5,6,6
1,3
4,6
2,5
3,6
2,3
```

拓扑排序算法及 C 语言实现

拓扑排序指的是将有向无环图（又称“DAG”图）中的顶点按照图中指定的先后顺序进行排序。

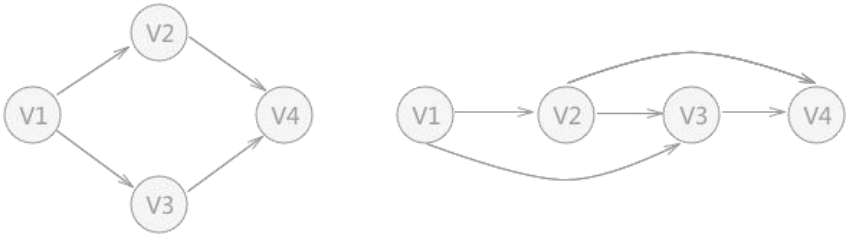


图 1 有向无环图

例如，图 1 中的两个图都是有向无环图，都可以使用拓扑排序对图中的顶点进行排序，两个图形的区别是：左图中的 V2 和 V3 之间没有明确的前后顺序；而右图中任意两个顶点之间都有前后顺序。

所以，左图中顶点之间的关系被称为“偏序”关系；右图中顶点之间的关系被称为”全序“关系。

在有向无环图中，弧的方向代表着顶点之间的先后次序，例如从 V1 指向 V2 的弧表示在进行排序时 V1 在前， V2 在后。

全序是偏序的一种特殊情况。对于任意一个有向无环图来说，通过拓扑排序得到的序列首先一定是偏序，如果任意两个顶点都具有前后顺序，那么此序列是全序。

拓扑排序的方法

对有向无环图进行拓扑排序，只需要遵循两个原则：

1. 在图中选择一个没有前驱的顶点 V；
2. 从图中删除顶点 V 和所有以该顶点为尾的弧。

例如，在对图 1 中的左图进行拓扑排序时的步骤如图 2 所示：

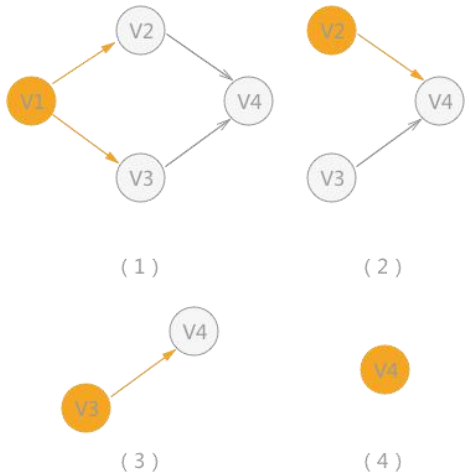


图 2 拓扑排序

有向无环图如果顶点本身具有某种实际意义，例如用有向无环图表示大学期间所学习的全部课程，每个顶点都表示一门课程，有向边表示课程学习的先后次序，例如要先学《程序设计基础》和《离散数学》，然后才能学习《数据结构》。所以用来表示某种活动间的优先关系的有向图简称为“AOV 网”。

进行拓扑排序时，首先找到没有前驱的顶点 V1，如图 2（1）所示；在删除顶点 V1 及以 V1 作为起点的弧后，继续查找没有前驱的顶点，此时， V2 和 V3 都符合条件，可以随机选择一个，例如图 2（2） 所示，选择 V2 ，然后继续重复以上的操作，直至最后找不到没有前驱的顶点。

所以，针对图 2 来说，拓扑排序最后得到的序列有两种：

- V1 -> V2 -> V3 -> V4
- V1 -> V3 -> V2 -> V4

> 如果顶点之间只是具有偏序关系，那么拓扑排序的结果肯定不唯一；如果顶点之间是全序关系，那么拓扑排序得到的序列唯一。

拓扑排序的 C 语言实现

在编写程序解决拓扑排序的问题时，大致思路为：首先通过邻接表将 AOV 网进行存储，由于拓扑排序的整个过程中，都是以顶点的入度为依据进行排序，所以需要根据建立的邻接表统计出各顶点的入度。

在得到各顶点的入度后，首先找到入度为 0 的顶点作为拓扑排序的起始点，然后查找以该顶点为起始点的所有顶点，如果入度为 1，说明如果删除前一个顶点后，该顶点的入度为 0，为拓扑排序的下一个对象。

实现代码：

```
#include <stdio.h>
#include <stdlib.h>
#define    MAX_VERTEX_NUM 20//最大顶点个数
#define    VertexType int//顶点数据的类型
typedef enum{false,true} bool;
typedef struct ArcNode{
    int adjvex;//邻接点在数组中的位置下标
    struct ArcNode * nextarc;//指向下一个邻接点的指针
}ArcNode;

typedef struct VNode{
    VertexType data;//顶点的数据域
    ArcNode * firstarc;//指向邻接点的指针
}VNode,AdjList[MAX_VERTEX_NUM];//存储各链表头结点的数组

typedef struct {
    AdjList vertices;//图中顶点及各邻接点数组
    int vexnum,arcnum;//记录图中顶点数和边或弧数
}ALGraph;
//找到顶点对应在邻接表数组中的位置下标
```

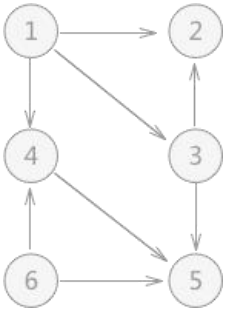
```
int LocateVex(ALGraph G,VertexType u){
    for (int i=0; i<G.vexnum; i++) {
        if (G.vertices[i].data==u) {
            return i;
        }
    }
    return -1;
}
//创建 AOV 网，构建邻接表
void CreateAOV(ALGraph **G){
    *G=(ALGraph*)malloc(sizeof(ALGraph));

    scanf("%d,%d",&((*G)->vexnum),&((*G)->arcnum));
    for (int i=0; i<(*G)->vexnum; i++) {
        scanf("%d",&((*G)->vertices[i].data));
        (*G)->vertices[i].firstarc=NULL;
    }
    VertexType initial,end;
    for (int i=0; i<(*G)->arcnum; i++) {
        scanf("%d,%d",&initial,&end);

        ArcNode *p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=LocateVex(*(*G), end);
        p->nextarc=NULL;

        int locate=LocateVex(*(*G), initial);
        p->nextarc=(*G)->vertices[locate].firstarc;
        (*G)->vertices[locate].firstarc=p;
    }
}
//结构体定义栈结构
typedef struct stack{
    VertexType data;
    struct stack * next;
}stack;
//初始化栈结构
void initStack(stack* *S){
```

```
    (*S)=(stack*)malloc(sizeof(stack));
    (*S)->next=NULL;
}
//判断链表是否为空
bool StackEmpty(stack S){
    if (S.next==NULL) {
        return true;
    }
    return false;
}
//进栈，以头插法将新结点插入到链表中
void push(stack *S,VertexType u){
    stack *p=(stack*)malloc(sizeof(stack));
    p->data=u;
    p->next=NULL;
    p->next=S->next;
    S->next=p;
}
//弹栈函数，删除链表首元结点的同时，释放该空间，并将该结点中的数据域通过地址传值给变量 i;
void pop(stack *S,VertexType *i){
    stack *p=S->next;
    *i=p->data;
    S->next=S->next->next;
    free(p);
}
//统计各顶点的入度
void FindInDegree(ALGraph G,int indegree[]){
    //初始化数组，默认初始值全部为 0
    for (int i=0; i<G.vexnum; i++) {
        indegree[i]=0;
    }
    //遍历邻接表，根据各链表中结点的数据域存储的各顶点位置下标，在 indegree 数组相应位置+1
    for (int i=0; i<G.vexnum; i++) {
        ArcNode *p=G.vertices[i].firstarc;
        while (p) {
            indegree[p->adjvex]++;
            p=p->nextarc;
```

```
        }
    }
}
void TopologicalSort(ALGraph G){
    int indegree[G.vexnum];//创建记录各顶点入度的数组
    FindInDegree(G,indegree);//统计各顶点的入度
    //建立栈结构，程序中使用的是链表
    stack *S;
    initStack(&S);
    //查找度为 0 的顶点，作为起始点
    for (int i=0; i<G.vexnum; i++) {
        if (!indegree[i]) {
            push(S, i);
        }
    }
    int count=0;
    //当栈为空，说明排序完成
    while (!StackEmpty(*S)) {
        int index;
        //弹栈，并记录栈中保存的顶点所在邻接表数组中的位置
        pop(S,&index);
        printf("%d",G.vertices[index].data);
        ++count;
        //依次查找跟该顶点相链接的顶点，如果初始入度为 1，当删除前一个顶点后，该顶点入度为 0
        for (ArcNode *p=G.vertices[index].firstarc; p; p=p->nextarc) {
            VertexType k=p->adjvex;
            if (!(--indegree[k])) {
                //顶点入度为 0，入栈
                push(S, k);
            }
        }
    }
    //如果 count 值小于顶点数量，表明该有向图有环
    if (count<G.vexnum) {
        printf("该图有回路");
        return;
    }
```

```
}


int main(){
    ALGraph *G;
    CreateAOV(&G);//创建 AOV 网
    TopologicalSort(*G);//进行拓扑排序
    return  0;
}
```

例如使用上述完整代码解决下图的有向无环图时，拓扑排序的结果为：



运行效果：

```
6,8
1
2
3
4
5
6
1,2
1,4
1,3
3,2
3,5
4,5
6,4
6,5
6 1 4 3 2 5
```

迪杰斯特拉算法详解及 C 语言实现

如今出行已经不需要再为找不着路而担心了，车上有车载导航，手机中有导航 App。只需要确定起点和终点，导航会自动规划出可行的距离最短的道路。这是最短路径在人们实际生活中最典型的应用。

在一个网 (有权图) 中，求一个顶点到另一个顶点的最短路径的计算方式有两种：迪杰斯特拉 (Dijkstra 算法) 和弗洛伊德 (Floyd)

算法。迪杰斯特拉算法计算的是有向网中的某个顶点到其余所有顶点的最短路径；弗洛伊德算法计算的是任意两顶点之间的最短路径。

最短路径算法既适用于有向网，也同样适用于无向网。本节将围绕有向网讲解迪杰斯特拉算法的具体实现。

迪杰斯特拉（Dijkstra 算法）

迪杰斯特拉算法计算的是从网中一个顶点到其它顶点之间的最短路径问题。

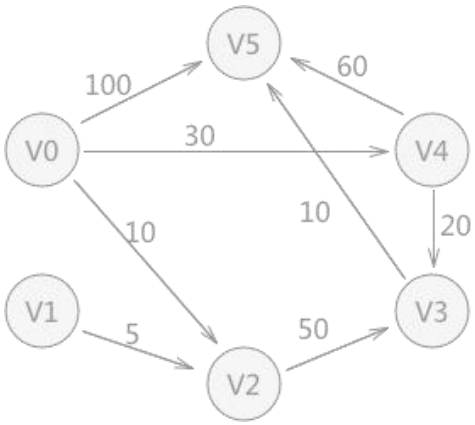


图 1 带权有向图

如图 1 所示是一个有向网，在计算 V0 到其它所有顶点之间的最小路径时，迪杰斯特拉算法的计算方式为：

从 V0 出发，由于可以直接到达 V2 和 V5 ，而其它顶点和 V0 之间没有弧的存在，所以之间的距离设定为无穷大，可以得到下面这个表格：

	V1	V2	V3	V4	V5
V0	∞	10	∞	30	100
涉及到的边	V0-V1	V0-V2	V0-V3	V0-V4	V0-V5

从表格中可以看到，V0 到 V2 的距离最近，所以迪杰斯特拉算法设定 V0-V2 为 V0 到 V2 之间的最短路径，最短路径的权值和为 10。

已经判断 V0-V2 是最短路径，所以以 V2 为起始点，判断 V2 到除了 V0 以外的其余各点之间的距离，如果对应的权值比前一张表格中记录的数值小，就说明网中有一条更短的路径，直接更新表格；反之表格中的数据不变。可以得到下面这个表格：

	V1	V2	V3	V4	V5
V0	∞	10	60	30	100
	V0-V1	V0-V2	V0V2-V3	V0-V4	V0-V5

例如，表格中 V0 到 V3 的距离，发现当通过 V2 到达 V3 的距离比之前的 ∞ 要小，所以更新表格。

更新之后，发现 V0-V4 的距离最近，设定 V0 到 V4 的最短路径的值为 30。之后从 V4 出发，判断到未确定最短路径的其它顶点的距离，继续更新表格：

	V1	V2	V3	V4	V5
V0	∞	10	50	30	90
	V0-V1	V0-V2	V0-V4-V3	V0-V4	V0-V4-V5

更新后确定从 V0 到 V3 的最短路径为 V0-V4-V3，权值为 50。然后从 V3 出发，继续判断：

	V1	V2	V3	V4	V5
V0	∞	10	50	30	60
	V0-V1	V0-V2	V0-V4-V3	V0-V4	V0-V4-V3-V5

对于 V5 来说，通过 V0-V4-V3-V5 的路径要比之前的权值 90 还要小，所以更新表格，更新后可以看到，V0-V5 的距离此时最短，可以确定 V0 到 V5 的最短路径为 60。

最后确定 V0-V1 的最短路径，由于从 V0 无法到达 V1 ，最终设定 V0 到 V1 的最短路径为 ∞（无穷大）。

在确定了 V0 与其他所有顶点的最短路径后，迪杰斯特拉算法才算结束。

> 事例中借用了图 1 的有向网对迪杰斯特拉算法进行了讲解，实际上无向网中的最短路径问题也可以使用迪杰斯特拉算法解决，解决过程和上述过程完全一致。

迪杰斯特拉算法的代码实现

```
#include <stdio.h>
#define MAX_VERtEX_NUM 20                    //顶点的最大个数
#define VRType int                           //表示弧的权值的类型
#define VertexType int                       //图中顶点的数据类型
#define INFINITY 65535
typedef struct {
    VertexType vexs[MAX_VERtEX_NUM];         //存储图中顶点数据
    VRType arcs[MAX_VERtEX_NUM][MAX_VERtEX_NUM];                    //二维数组，记录顶点之间的关系
    int vexnum,arcnum;                       //记录图的顶点数和弧（边）数
}MGraph;

typedef int PathMatrix[MAX_VERtEX_NUM];      //用于存储最短路径中经过的顶点的下标
typedef int ShortPathTable[MAX_VERtEX_NUM]; //用于存储各个最短路径的权值和

//根据顶点本身数据，判断出顶点在二维数组中的位置
int LocateVex(MGraph * G,VertexType v){
    int i=0;
```

```
    //遍历一维数组，找到变量 v
    for (; i<G->vexnum; i++) {
        if (G->vexs[i]==v) {
            break;
        }
    }
    //如果找不到，输出提示语句，返回-1
    if (i>G->vexnum) {
        printf("no such vertex.\n");
        return -1;
    }
    return i;
}
//构造有向网
void CreateUDG(MGraph *G){
    scanf("%d,%d",&(G->vexnum),&(G->arcnum));
    for (int i=0; i<G->vexnum; i++) {
        scanf("%d",&(G->vexs[i]));
    }
    for (int i=0; i<G->vexnum; i++) {
        for (int j=0; j<G->vexnum; j++) {
            G->arcs[i][j]=INFINITY;
        }
    }
    for (int i=0; i<G->arcnum; i++) {
        int v1,v2,w;
        scanf("%d,%d,%d",&v1,&v2,&w);
        int n=LocateVex(G, v1);
        int m=LocateVex(G, v2);
        if (m==-1 ||n==-1) {
            printf("no this vertex\n");
            return;
        }
        G->arcs[n][m]=w;
    }
}
//迪杰斯特拉算法，v0 表示有向网中起始点所在数组中的下标
```

```
void ShortestPath_Dijkstra(MGraph G,int v0,PathMatrix *p,ShortPathTable *D){
    int final[MAX_VERtEX_NUM];//用于存储各顶点是否已经确定最短路径的数组
    //对各数组进行初始化
    for (int v=0; v<G.vexnum; v++) {
        final[v]=0;
        (*D)[v]=G.arcs[v0][v];
        (*p)[v]=0;
    }
    //由于以 v0 位下标的顶点为起始点，所以不用再判断
    (*D)[v0]=0;
    final[v0]=1;
    int k = 0;
    for (int i=0; i<G.vexnum; i++) {
        int min=INFINITY;
        //选择到各顶点权值最小的顶点，即为本次能确定最短路径的顶点
        for (int w=0; w<G.vexnum; w++) {
            if (!final[w]) {
                if ((*D)[w]<min) {
                    k=w;
                    min=(*D)[w];
                }
            }
        }
        //设置该顶点的标志位为 1，避免下次重复判断
        final[k]=1;
        //对 v0 到各顶点的权值进行更新
        for (int w=0; w<G.vexnum; w++) {
            if (!final[w]&&(min+G.arcs[k][w]<(*D)[w])) {
                (*D)[w]=min+G.arcs[k][w];
                (*p)[w]=k;//记录各个最短路径上存在的顶点
            }
        }
    }
}
int main(){
    MGraph G;
    CreateUDG(&G);
```

```
    PathMatrix P;
    ShortPathTable D;
    ShortestPath_Dijkstra(G, 0, &P, &D);
    for (int i=1; i<G.vexnum; i++) {
        printf("V%d - V%d 的最短路径中的顶点有(逆序)：",0,i);
        printf(" V%d",i);
        int j=i;
        //由于每一段最短路径上都记录着经过的顶点，所以采用嵌套的方式输出即可得到各个最短路径上的所有顶点
        while (P[j]!=0) {
            printf(" V%d",P[j]);
            j=P[j];
        }
        printf(" V0\n");
    }
    printf("源点到各顶点的最短路径长度为:\n");
    for (int i=1; i<G.vexnum; i++) {
        printf("V%d - V%d : %d \n",G.vexs[0],G.vexs[i],D[i]);
    }
    return 0;
}
```

运行以上代码，计算如图 1 所示的有向网，运行结果为：

```
6,8
0
1
2
3
4
5
0,5,100
0,4,30
0,2,10
1,2,5
2,3,50
3,5,10
4,3,20
4,5,60
V0 - V1 的最短路径中的顶点有：  V1 V0
V0 - V2 的最短路径中的顶点有：  V2 V0
V0 - V3 的最短路径中的顶点有：  V3 V4 V0
```

```
V0 - V4 的最短路径中的顶点有：  V4 V0
V0 - V5 的最短路径中的顶点有：  V5 V3 V4 V0
源点到各顶点的最短路径长度为:
V0 - V1 : 65535
V0 - V2 : 10
V0 - V3 : 50
V0 - V4 : 30
V0 - V5 : 60
```

总结

迪杰斯特拉算法解决的是从网中的一个顶点到所有其它顶点之间的最短路径，算法整体的时间复杂度为 O(n2)。但是如果需要求任意两顶点之间的最短路径，使用迪杰斯特拉算法虽然最终虽然也能解决问题，但是大材小用，相比之下使用弗洛伊德算法解决此类问题会更合适。

弗洛伊德算法实现的具体过程下一节会作详细介绍。

弗洛伊德算法详解及 C 语言实现

通过前一节对迪杰斯特拉算法的学习，主要解决从网（带权图）中某一顶点计算到其它顶点之间的最短路径问题。如果求有向网中每一对顶点之间的最短路径，使用迪杰斯特拉算法的解决思路是：以每一个顶点为源点，执行迪杰斯特拉算法。这样可以求得每一对顶点之间的最短路径。

本节介绍另外一种解决算法：弗洛伊德算法，该算法相比于使用迪杰斯特拉算法在解决此问题上的时间复杂度虽然相同，都为 $O(n^3)$，但是弗洛伊德算法的实现形式更简单。

弗洛伊德的核心思想是：对于网中的任意两个顶点（例如顶点 A 到顶点 B）来说，之间的最短路径不外乎有 2 种情况：

1. 直接从顶点 A 到顶点 B 的弧的权值为顶点 A 到顶点 B 的最短路径；
2. 从顶点 A 开始，经过若干个顶点，最终达到顶点 B，期间经过的弧的权值和为顶点 A 到顶点 B 的最短路径。

所以，弗洛伊德算法的核心为：对于从顶点 A 到顶点 B 的最短路径，拿出网中所有的顶点进行如下判断：

```
Dis（A，K）+ Dis（K，B）< Dis（A，B）
```

其中，K 表示网中所有的顶点；Dis（A，B） 表示顶点 A 到顶点 B 的距离。

也就是说，拿出所有的顶点 K，判断经过顶点 K 是否存在一条可行路径比直达的路径的权值小，如果式子成立，说明确实存在一条权值更小的路径，此时只需要更新记录的权值和即可。

任意的两个顶点全部做以上的判断，最终遍历完成后记录的最终的权值即为对应顶点之间的最短路径。

完整实例

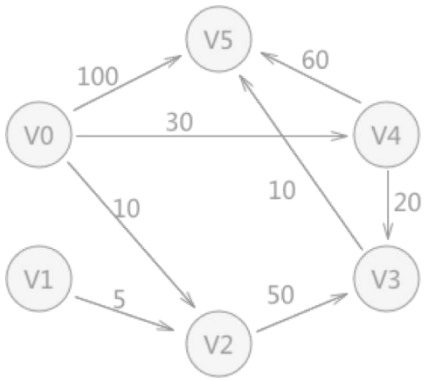


图 1 带权图

例如，在使用弗洛伊德算法计算图 1 中的任意两个顶点之间的最短路径时，具体实施步骤为：

首先，记录顶点之间初始的权值，如下表所示：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	∞	30	100
V1	∞	∞	5	∞	∞	∞
V2	∞	∞	∞	50	∞	∞
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	60
V5	∞	∞	∞	∞	∞	∞

依次遍历所有的顶点，假设从 V0 开始，将 V0 作为中间点，看每对顶点之间的距离值是否会更小。最终 V0 对于每对顶点之间的距离没有任何改善。

对于 V0 来说，由于该顶点只有出度，没有入度，所以没有作为中间点的可能。同理，V1 也没有可能。

将 V2 作为每对顶点的中间点，有影响的为 （V0，V3） 和 （V1，V3）：

例如，（V0，V3）权值为无穷大，而（V0，V2）+（V2，V3）= 60，比之前的值小，相比而言后者的路径更短；同理 （V1，V3）也是如此。

更新的表格为：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	60	30	100
V1	∞	∞	5	55	∞	∞
V2	∞	∞	∞	50	∞	∞
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	60
V5	∞	∞	∞	∞	∞	∞

以 V3 作为中间顶点遍历各队顶点，更新后的表格为：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	60	30	70
V1	∞	∞	5	55	∞	65
V2	∞	∞	∞	50	∞	60
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	30
V5	∞	∞	∞	∞	∞	∞

以 V4 作为中间顶点遍历各队顶点，更新后的表格为：

	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	50	30	60
V1	∞	∞	5	55	∞	65
V2	∞	∞	∞	50	∞	60
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	30
V5	∞	∞	∞	∞	∞	∞

而对于顶点 V5 来说，和顶点 V0 和 V1 相类似，所不同的是，V5 只有入度，没有出度，所以对各队顶点的距离不会产生影响。最终采用弗洛伊德算法求得的各个顶点之间的最短路径如上图所示。

弗洛伊德算法的具体实现

```
#include <stdio.h>

#define MAX_VERtEX_NUM 20                    //顶点的最大个数

#define VRType int                           //表示弧的权值的类型
```

```
#define VertexType int                          //图中顶点的数据类型
#define INFINITY 65535
typedef struct {
    VertexType vexs[MAX_VERtEX_NUM];            //存储图中顶点数据
    VRType arcs[MAX_VERtEX_NUM][MAX_VERtEX_NUM];                    //二维数组，记录顶点之间的关系
    int vexnum,arcnum;                          //记录图的顶点数和弧（边）数
}MGraph;

typedef int PathMatrix[MAX_VERtEX_NUM][MAX_VERtEX_NUM];       //用于存储最短路径中经过的顶点的下标
typedef int ShortPathTable[MAX_VERtEX_NUM][MAX_VERtEX_NUM]; //用于存储各个最短路径的权值和

//根据顶点本身数据，判断出顶点在二维数组中的位置
int LocateVex(MGraph * G,VertexType v){
    int i=0;
    //遍历一维数组，找到变量 v
    for (; i<G->vexnum; i++) {
        if (G->vexs[i]==v) {
            break;
        }
    }
    //如果找不到，输出提示语句，返回-1
    if (i>G->vexnum) {
        printf("no such vertex.\n");
        return -1;
    }
    return i;
}
//构造有向网
void CreateUDG(MGraph *G){
    scanf("%d,%d",&(G->vexnum),&(G->arcnum));
    for (int i=0; i<G->vexnum; i++) {
        scanf("%d",&(G->vexs[i]));
    }
    for (int i=0; i<G->vexnum; i++) {
        for (int j=0; j<G->vexnum; j++) {
            G->arcs[i][j]=INFINITY;
        }
```

```
    }
    for (int i=0; i<G->arcnum; i++) {
        int v1,v2,w;
        scanf("%d,%d,%d",&v1,&v2,&w);
        int n=LocateVex(G, v1);
        int m=LocateVex(G, v2);
        if (m==-1 ||n==-1) {
            printf("no this vertex\n");
            return;
        }
        G->arcs[n][m]=w;
    }
}
//弗洛伊德算法，其中 P 二维数组存放各对顶点的最短路径经过的顶点，D 二维数组存储各个顶点之间的权值
void ShortestPath_Floyed(MGraph G,PathMatrix *P,ShortPathTable *D){
    //对 P 数组和 D 数组进行初始化
    for (int v=0; v<G.vexnum; v++) {
        for (int w=0; w<G.vexnum; w++) {
            (*D)[v][w]=G.arcs[v][w];
            (*P)[v][w]=-1;
        }
    }
    //拿出每个顶点作为遍历条件
    for (int k=0; k<G.vexnum; k++) {
        //对于第 k 个顶点来说，遍历网中任意两个顶点，判断间接的距离是否更短
        for (int v=0; v<G.vexnum; v++) {
            for (int w=0; w<G.vexnum; w++) {
                //判断经过顶点 k 的距离是否更短，如果判断成立，则存储距离更短的路径
                if ((*D)[v][w] > (*D)[v][k] + (*D)[k][w]) {
                    (*D)[v][w]=(*D)[v][k] + (*D)[k][w];
                    (*P)[v][w]=k;
                }
            }
        }
    }
}
int main(){
```

```
    MGraph G;
    CreateUDG(&G);
    PathMatrix P;
    ShortPathTable D;
    ShortestPath_Floyed(G, &P, &D);
    for (int i=0; i<G.vexnum; i++) {
        for (int j=0; j<G.vexnum; j++) {
            printf("%d ",P[i][j]);
        }
        printf("\n");
    }
    for (int i=0; i<G.vexnum; i++) {
        for (int j=0; j<G.vexnum; j++) {
            printf("%d ",D[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

运行结果为（以图 1 为例）：

```
6,8
0
1
2
3
4
5
0,5,100
0,4,30
0,2,10
1,2,5
2,3,50
3,5,10
4,3,20
4,5,60
-1 -1 -1 4 -1 4
-1 -1 -1 2 -1 3
-1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 3
```

```
-1 -1 -1 -1 -1 -1
65535 65535 10 50 30 60
65535 65535 5 55 65535 65
65535 65535 65535 50 65535 60
65535 65535 65535 65535 65535 10
65535 65535 65535 20 65535 30
65535 65535 65535 65535 65535 65535
```

其中，输出结果中 65535 表示该位置所表示的两顶点之间的距离为无穷大。

数据结构实践项目之移动迷宫小游戏(升级版)

在上一章的《移动迷宫》小游戏中，使用回溯法帮助骑士在迷宫中找到了通往出口的一条通路，但是骑士并不太满意，他又提出了更高的需求。

《移动迷宫》升级版游戏简介：迷宫只有两个门，一个入口，一个出口。一个骑士骑马从入口走进迷宫，迷宫中设置有很多墙壁，对前进方向造成障碍。先需要你从迷宫中找到一条最短的通路，将行走路线和行走的最短距离告知骑士。

设计思路

类似于寻找最短路径这样的问题，最直接的方法就是使用迪杰斯特拉算法和弗洛伊德算法。

两种算法面对的数据结构是图，但是迷宫是在二维数组中进行存储的，所以如果使用前面两种算法的话，需要首先将二维数组转化为图的存储形式。

二维数组转化成图

如下图所示，此为 3*3 迷宫：

S	-	#
-	-	-
#	#	E

图 1 移动迷宫

提示：S 为入口，E 为出口，# 为墙壁，- 为通路。

在编写程序向计算机中输入该迷宫的数据时，宜使用二维数组进行存储。但是无论是迪杰斯特拉算法还是弗洛伊德算法，其处理对象都是有向网或者无向网。迷宫中并不涉及到具体的方向，所以需要将存储迷宫的二维数组转化为无向网。

无向网的存储方式也是用二维数组来实现，将迷宫中所有的顶点看作是图中的顶点，对于上图的迷宫来说，共有 9 个顶点，所以转化为无向网时，需要用 9*9 的一个二维数组来表示。

在转化时，从迷宫的左上角（上图的 S 开始），一行一行的进行转化，对于每个顶点来说，只需要判断其右侧和相邻的下方顶

点是否为通路，如果是通路，转化为图中的直接体现就是两顶点之间有线连接。

例如，上图中的 S 其右侧和下方的顶点都是 - ，骑士可以通过，那在图中的表现就是 S 同其右侧顶点和下方顶点之间存储通路，如下图所示：

对于图 1 中的二维数组，其完全转化为图，如下图所示（每个顶点用其二维数组中的坐标来表示，00 表示第 0 行第 0 列）：

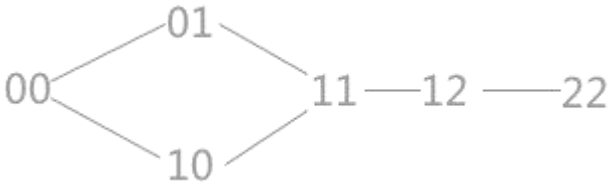


图 1 中的二维数组转化为图的存储表示如下图所示：

	S	-	#	-	-	-	#	#	E
S	0	1	0	1	0	0	0	0	0
-	1	0	0	0	1	0	0	0	0
#	0	0	0	0	0	0	0	0	0
-	1	0	0	0	1	0	0	0	0
-	0	1	0	1	0	1	0	0	0
-	0	0	0	0	1	0	0	0	1
#	0	0	0	0	0	0	0	0	0
#	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	1	0	0	0

提示：1 表示有通路，0 表示没有通路，# 由于表示墙壁，同其它任何顶点之间都没有通路。

迪杰斯特拉算法

使用迪杰斯特拉算法求迷宫的最短路径，其完整代码如下：

```
#include <stdio.h>
#define MAX_VERtEX_NUM 1000                    //顶点的最大个数
#define VRType int                             //表示弧的权值的类型
#define VertexType char                        //图中顶点的数据类型
#define INFINITY 65535
typedef enum{false,true} bool;
typedef struct {
    VertexType vexs[MAX_VERtEX_NUM];           //存储图中顶点数据
```

```
    VRType arcs[MAX_VERtEX_NUM][MAX_VERtEX_NUM];                    //二维数组，记录顶点之间的关系
    int vexnum,arcnum;                          //记录图的顶点数和弧（边）数
}MGraph;
typedef int PathMatrix[MAX_VERtEX_NUM];       //用于存储最短路径中经过的顶点的下标
typedef int ShortPathTable[MAX_VERtEX_NUM]; //用于存储各个最短路径的权值和

//迪杰斯特拉算法，v0 表示有向网中起始点所在数组中的下标
void ShortestPath_Dijkstra(MGraph G,int v0,PathMatrix *p,ShortPathTable *D){
    int final[MAX_VERtEX_NUM];//用于存储各顶点是否已经确定最短路径的数组
    //对各数组进行初始化
    for (int v=0; v<G.vexnum; v++) {
        final[v]=0;
        (*D)[v]=G.arcs[v0][v];
        (*p)[v]=0;
    }
    //以起点为下标的顶点为起始点，所以不用再判断
    (*D)[v0]=0;
    final[v0]=1;
    int k = 0;
    for (int i=0; i<G.vexnum; i++) {
        int min=INFINITY;
        //选择到各顶点权值最小的顶点，即为本次能确定最短路径的顶点
        for (int w=0; w<G.vexnum; w++) {
            if (!final[w]) {
                if ((*D)[w]<min) {
                    k=w;
                    min=(*D)[w];
                }
            }
        }
        //设置该顶点的标志位为 1，避免下次重复判断
        final[k]=1;
        //对从起点到各顶点的权值进行更新
        for (int w=0; w<G.vexnum; w++) {
            if (!final[w]&&(min+G.arcs[k][w]<(*D)[w])) {
                (*D)[w]=min+G.arcs[k][w];
                (*p)[w]=k;//记录各个最短路径上存在的顶点
```

```
            }
        }
    }
}
//在将二维数组转化为图的过程中，需要判断当前的点是否越界或者是否为通路
bool canUsed(int i,int j,int n,int m,char a[][110]){
    if (a[i][j]!='#' && i>=0 && i<n && j>=0 && j<m) {
        return true;
    }
    return false;
}
int main(){
    char a[110][110];
    int n,m;
    scanf("%d %d",&n,&m);
    getchar();
    MGraph G;
    G.vexnum=0;
    G.arcnum=0;
    //记录入口在图的顶点数组中的位置下标
    int start =0;
    //记录出口在图的顶点数组中的位置下标
    int exit=0;
    //初始化记录图的边的二维数组，假设各个边的长度为无穷大，即两顶点之间没有边
    for (int i=0; i<n*m; i++) {
        for (int j=0; j<n*m; j++) {
            G.arcs[i][j]=INFINITY;
        }
    }
    //输入二维数组，同时记录入口和出口的位置
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            scanf("%c",&a[i][j]);
            G.vexs[i*m+j]=a[i][j];
            G.vexnum++;
            if (a[i]
                [j]=='S') {
```

```
                start=i*m+j;
            }else if(a[i][j]=='E'){
                exit=i*m+j;
            }
        }
        getchar();//作用是为了读取缓存区中的换行符（因为迷宫是一行一行输入到内存中的）
    }
    //将二维数组转换为无向图，在转换时，从二维数组的左上角开始，每次判断当前顶点的右侧和下侧是否为通路，这样所有的通路就可以转换为无向图中的边。
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            //首先判断当前点是否为通路
            if (canUsed(i, j, n, m, a)) {
                if (canUsed(i+1, j, n, m, a)) {
                    //设定两顶点之间的边的权值为 1
                    G.arcs[i*m+j][(i+1)*m+j]=1;
                    G.arcs[(i+1)*m+j][i*m+j]=1;
                    G.arcnum++;
                }
                if (canUsed(i, j+1, n, m, a)) {
                    G.arcs[i*m+j][i*m+j+1]=1;
                    G.arcs[i*m+j+1][i*m+j]=1;
                    G.arcnum++;
                }
            }
        }
    }
    PathMatrix P;
    ShortPathTable D;
    //进行迪杰斯特拉算法
    ShortestPath_Dijkstra(G,start, &P, &D);
    //如果最终记录的权值和还是无穷大，证明，入口和出口之间没有通路
    if (D[exit]==INFINITY) {
        printf("-1");
    }else{
        printf("入口到出口的最短路径长度为:\n");
        printf("%d\n",D[exit]);
```

```
            printf("入口到出口的最短路径为(逆序):\n");
            printf("(%d,%d) ",exit/m,exit%m);
            while (P[exit]!=0) {
                printf("(%d,%d) ",P[exit]/m,P[exit]%m);
                exit=P[exit];
            }
            printf("(%d,%d)\n",start/m,start%m);
        }
    return 0;
}
```

运行结果：

程序输入：

3 3

S-#

##E

程序输出：

入口到出口的最短路径长度为:

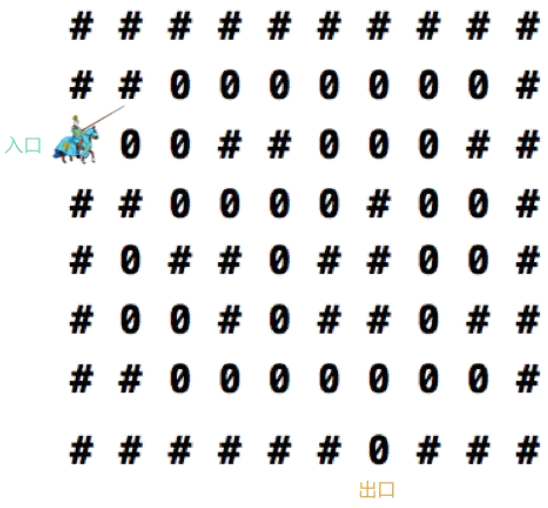
4

入口到出口的最短路径为(逆序):

(2,2) (1,2) (1,1) (0,1) (0,0)

广度优先搜索解决最短路径问题

除了以上两种可以称得上是直接求最短路径的方法，还可以应用本章的广度优先搜索算法查找最短路径，该算法的实现可以直接在二维数组中完成，没有必要转化为图的形式。



例如拿上图中的迷宫举例，骑士一开始只能选择向右走，当走到坐标为（2,2）的位置，骑士有两个选择：向上走或者向下走。

对于广度优先搜索来说，其实现使用的数据存储结构为队列，在搜索的过程中，将每种可选情况都入队，然后一轮一轮的对队列中的可选情况进行尝试，知道尝试出想要的结果为止。

对于此时的骑士来说，结合对广度优先算法的理解，就相当于骑士会分身术，一分为二，一个往上，一个往下，每个人每次只能走一步（你走一步然后我走一步）。

例如假设骑士走下，分身去上，当骑士走到坐标为（3，4）的位置时，又需要选择，要么往右，要么往下，此时骑士又分身，各走各的。但是无论怎么分，所有的骑士都是每次只走一步。

在这种情况下，当只要有一个骑士找到出口时，他所走的路径就绝对是最短路径。

对于广度优先搜索来说，使用的是队列的数据结构，等同于在遍历一棵二叉树时，一层一层的遍历（从上往下，从左往右），可以看做是，对于每种情况，轮流去试探，每次只走一步。

在实际编程实现时，使用广度优先搜索查找最短路径时，只能求得最短路径的长度，如果想要获取最短路径的具体路线，还需要结合其他算法。

在本节，给大家的一个解决思路是：在存储迷宫时，对于每个顶点都分配一个整形变量，在进行广度优先搜索时，骑士和其分身每走一步，该顶点所携带的整形变量的值都是骑士之前所处位置的整形变量+1。

例如，对于下图的迷宫来说，骑士在最终找到出口时的整形变量为：

提示：从入口开始，初始值假设为 0 ，其右侧通路和下方通路的整形变量的值是 0+1=1，最终其出口自身所携带的整形变量值就是最短路径的长度。

通过对“骑士们”所走路线中整形变量的设置，此时我们可以结合回溯法，从入口开始寻找骑士所可能走的所有的最短路径（此时找到的可能不只有一条）。

在使用回溯法时，从入口出发，每次同当前顶点周围查找比自身整形变量值大 1 的顶点，就是骑士所走的路线。如果找不到，回退再找，直到将所有的情况都试探完。

广度优先搜索+回溯法解决移动迷宫问题的完整代码为：

```
#include <stdio.h>
typedef enum{false,true} bool;
typedef struct {
    int x;
    int y;
```

```
    char mess;
    int value;
}check;
bool canUsed(int x,int y,char data,int n,int m){
    if (x>=0 && x<n && y>=0 && y<m && data!='#') {
        return true;
    }
    return false;
}
void createMaze(int n,int m,check a[][110],int *entryx,int *entryy,int *exitx,int *exity){
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            scanf("%c",&a[i][j].mess);
            a[i][j].x=i;
            a[i][j].y=j;
            if (a[i][j].mess=='S') {
                *entryx=i;
                *entryy=j;
            }else if(a[i][j].mess=='E'){
                *exitx=i;
                *exity=j;
            }
        }
        getchar();
    }
}
//使用的广度优先搜索的思想，采用队列的数据结构实现
void findRoad(check a[][110],int top,int rear,check queue[],int *value,int entryx,int entryy,int n,int m){
    //首先将入口顶点入队
    check data;
    data.x=entryx;
    data.y=entryy;
    a[entryx][entryy].mess='#';
    data.mess=a[entryx][entryy].mess;
    data.value=0;
    queue[rear]=data;
    bool success=false;
```

```
    rear++;
    //队列不满
    while (top!=rear) {
        //逐个出队
        check temp=queue[top];
        a[temp.x][temp.y].value=temp.value;
        top++;
        //对于出队的顶点判断是否是出口，首个判断为出口的顶点，其 value 值就是最短路径的长度
        if (temp.mess=='E') {
            *value=temp.value;
            printf("%d\n",temp.value);
            success=true;
            break;
        }
        //每次入队，判断其上、下、左、右的顶点是否符合条件，若符合，则入队，同时对其 value 值赋值为前一个顶点 value+1，
为了避免重复判断此顶点，对每个入队的顶点，设定其字符为'#'
        if(canUsed(temp.x-1,temp.y,a[temp.x-1][temp.y].mess,n,m)){
            data.x=temp.x-1;
            data.y=temp.y;
            data.mess=a[temp.x-1][temp.y].mess;
            data.value=temp.value+1;
            queue[rear]=data;
            a[temp.x-1][temp.y].mess='#';
            rear++;
        }
        //右边
        if(canUsed(temp.x,temp.y+1,a[temp.x][temp.y+1].mess,n,m)){
            data.x=temp.x;
            data.y=temp.y+1;
            data.mess=a[temp.x][temp.y+1].mess;
            data.value=temp.value+1;
            queue[rear]=data;
            a[temp.x][temp.y+1].mess='#';
            rear++;
        }
        //下边
        if(canUsed(temp.x+1,temp.y,a[temp.x+1][temp.y].mess,n,m)){
```

```
                data.x=temp.x+1;
                data.y=temp.y;
                data.mess=a[temp.x+1][temp.y].mess;
                data.value=temp.value+1;
                queue[rear]=data;
                a[temp.x+1][temp.y].mess='#';
                rear++;
            }
            //左边
            if(canUsed(temp.x,temp.y-1,a[temp.x][temp.y-1].mess,n,m)){
                data.x=temp.x;
                data.y=temp.y-1;
                data.mess=a[temp.x][temp.y-1].mess;
                data.value=temp.value+1;
                queue[rear]=data;
                a[temp.x][temp.y-1].mess='#';
                rear++;
            }
        }
        //如果不成功，证明出口和入口之间没有通路
        if (success==false) {
            printf("-1\n");
        }
}
//用于输出最短路径时回溯过程中的判断
bool judgeValue(int x,int y,int n,int m){
        if (x>=0 && x<n && y>=0 && y<m ){
            return true;
        }
        return false;
}
//在输出时，由于最短路径中从入口开始，一直到出口，所经过的顶点的 value 值逐渐 +1，所以采用回溯法查找所有可能的最短路径
void displayRoad(check a[][110],int entryx,int entryy,int n,int m,int value){
        //设置静态数组，实现栈的作用
        static check stack[1000];
        static int top=-1;//栈的栈顶
```

```
    //对于每个当前的顶点，首先需要判断其是否符合最基本的要求，由于在前期二维数组中的通路都变成了'#'，这里采用另一个关键字 ，value 的值为主关键字进行搜索
    if (judgeValue(entryx, entryy, n, m)) {
        //回溯思想的实现用的是递归，所以需要设置一个出口，出口就是当查找到顶点的 value 值为最短路径的顶点数时，表明此时已经搜索在出口位置，此时就可以依次输出栈内存储的各个经过的顶点的坐标
        if (a[entryx][entryy].value==value) {
            for (int i=0; i<top; i++) {
                printf("(%d,%d) ",stack[i].x,stack[i].y);
            }
            printf("\n");
            return;
        }
        //从入口出发，判断当前点的上、下、左、右位置上的顶点是否符合要求：1、该顶点的坐标没有超出范围；2 该顶点的 value 值是前一个顶点的 value 值+1，如果都符合，说明之前判断最短路径时就途径此顶点，将其入栈进行保存
        if (judgeValue(entryx+1, entryy, n, m) && a[entryx+1][entryy].value==a[entryx][entryy].value+1) {
            top++;
            stack[top]=a[entryx+1][entryy];
            displayRoad(a, entryx+1, entryy, n, m,value);
            //当运行至此，又两种情况：途径此顶点最终找到出口，并将最终结果输出，此时应将该顶点弹栈；该顶点的路径不是正确的，应弹栈。两种情况都应弹栈。
            top--;
        }
        if (judgeValue(entryx-1, entryy, n, m) && a[entryx-1][entryy].value==a[entryx][entryy].value+1) {
            top++;
            stack[top]=a[entryx-1][entryy];
            displayRoad(a, entryx-1, entryy, n, m,value);
            top--;
        }
        if (judgeValue(entryx, entryy+1, n, m) && a[entryx][entryy+1].value==a[entryx][entryy].value+1) {
            top++;
            stack[top]=a[entryx][entryy+1];
            displayRoad(a, entryx, entryy+1, n, m,value);
            top--;
        }
        if (judgeValue(entryx, entryy-1, n, m) && a[entryx][entryy-1].value==a[entryx][entryy].value+1) {
            top++;
            stack[top]=a[entryx][entryy-1];
            displayRoad(a, entryx, entryy-1, n, m,value);
```

```
                top--;
            }
        }
}
int main(int argc, const char * argv[]) {
    check a[110][110];
    check queue[100000];
    int top=0,rear=0;
    int n,m;
    int entryx = 0,entryy=0,exitx=0,exity=0;
    scanf("%d %d",&n,&m);
    getchar();
    //创建迷宫，并找到入口和出口的位置坐标
    createMaze(n,m,a,&entryx,&entryy,&exitx,&exity);
    //在迷宫中查找从入口到出口的最短路径，若有，输出最短路径的长度；反之，输出-1
    int value;
    findRoad(a,top,rear,queue,&value,entryx,entryy,n,m);
    //输出所有的最短路径
    displayRoad(a, entryx, entryy, n, m, value);
    return 0;
}
```

运行结果:

输入部分:

```
3 3
S-#
---
#-E
4
```

程序输出部分:

```
(1,0) (1,1) (2,1)
(1,0) (1,1) (1,2)
(0,1) (1,1) (2,1)
(0,1) (1,1) (1,2)
```

提示：通过对通路进行整体的回溯，可以找到所有可能的结果，这个例子中，有四条长度相同的最短路径。

总结

本节主要解决的是求有关点到点的最短路径的问题，其问题的解决既可以使用“专业工具” -- 迪杰斯特拉算法和弗洛伊德算法，也有“组合套装” -- 广度优先搜索 + 回溯法。通过这个项目，大家要学会将所学的知识融会贯通，综合运用。

分块查找算法（索引顺序查找）及 C 语言实现

本节介绍一种在顺序查找的基础上对其进行改进的算法——分块查找算法。

分块查找，也叫索引顺序查找，算法实现除了需要查找表本身之外，还需要根据查找表建立一个索引表。例如图 1，给定一个查找表，其对应的索引表如图所示：

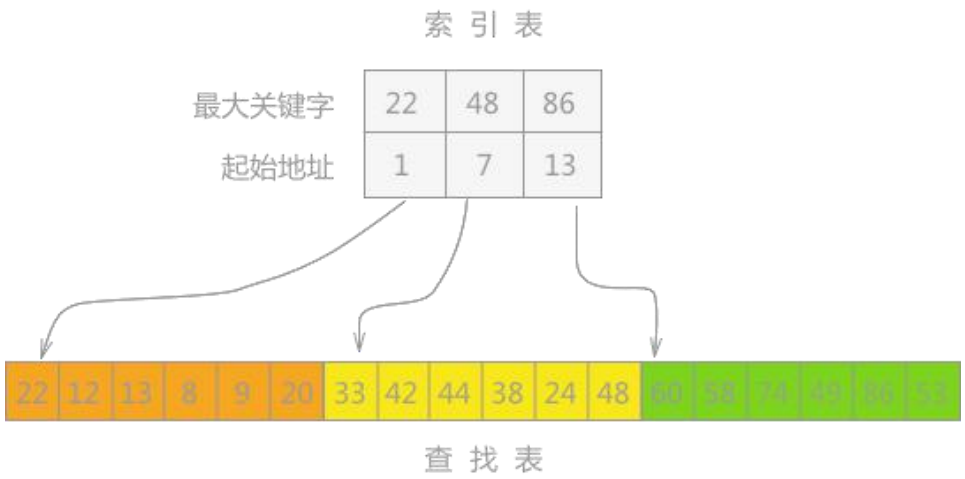


图 1 查找表及其对应的索引表

图 1 中，查找表中共 18 个查找关键字，将其平均分为 3 个子表，对每个子表建立一个索引，索引中包含中两部分内容：该子表部分中最大的关键字以及第一个关键字在总表中的位置，即该子表的起始位置。

建立的索引表要求按照关键字进行升序排序，查找表要么整体有序，要么分块有序。

分块有序指的是第二个子表中所有关键字都要大于第一个子表中的最大关键字，第三个子表的所有关键字都要大于第二个子表中的最大关键字，依次类推。

块（子表）中各关键字的具体顺序，根据各自可能会被查找到的概率而定。如果各关键字被查找到的概率是相等的，那么可以随机存放；否则可按照被查找概率进行降序排序，以提高算法运行效率。

分块查找的具体实现

所有前期准备工作完成后，开始在此基础上进行分块查找。分块查找的过程分为两步进行：

1. 确定要查找的关键字可能存在的具体块（子表）；
2. 在具体的块中进行顺序查找。

以图 1 中的查找表为例，假设要查找关键字 38 的具体位置。首先将 38 依次和索引表中各最大关键字进行比较，因为 22 < 38 < 48，所以可以确定 38 如果存在，肯定在第二个子表中。

由于索引表中显示第二子表的起始位置在查找表的第 7 的位置上，所以从该位置开始进行顺序查找，一直查找到该子表最后一个关键字（一般将查找表进行等分，具体子表个数根据实际情况而定）。结果在第 10 的位置上确定该关键字即为所找。

提示：在第一步确定块（子表）时，由于索引表中按照关键字有序，所有可以采用折半查找算法。而在第二步中，由于各子表中关键字没有严格要求有序，所以只能采用顺序查找的方式。

具体实现代码:

```
#include <stdio.h>
#include <stdlib.h>
struct index {    //定义块的结构
    int key;
    int start;
} newIndex[3];     //定义结构体数组

int search(int key, int a[]);

int cmp(const void *a,const void* b){
    return (*(struct index*)a).key>(*(struct index*)b).key?1:-1;
}
int main(){
    int i, j=-1, k, key;
    int a[] = {33,42,44,38,24,48, 22,12,13,8,9,20,    60,58,74,49,86,53};
    //确认模块的起始值和最大值
    for (i=0; i<3; i++) {
        newIndex[i].start = j+1;   //确定每个块范围的起始值
        j += 6;
        for (int k=newIndex[i].start; k<=j; k++) {
            if (newIndex[i].key<a[k]) {
                newIndex[i].key=a[k];
            }
        }
    }
    //对结构体按照 key 值进行排序
    qsort(newIndex,3, sizeof(newIndex[0]), cmp);

    //输入要查询的数，并调用函数进行查找
    printf("请输入您想要查找的数：\n");
    scanf("%d", &key);
    k = search(key, a);
    //输出查找的结果
    if (k>0) {
```

```
            printf("查找成功！您要找的数在数组中的位置是：%d\n",k+1);
    }else{
            printf("查找失败！您要找的数不在数组中。\n");
    }
    return 0;
}
int search(int key, int a[]){
    int i, startValue;
    i = 0;
    while (i<3 && key>newIndex[i].key) { //确定在哪个块中，遍历每个块，确定 key 在哪个块中
            i++;
    }
    if (i>=3) {   //大于分得的块数，则返回 0
            return -1;
    }
    startValue = newIndex[i].start;   //startValue 等于块范围的起始值
    while (startValue <= startValue+5 && a[startValue]!=key)
    {
            startValue++;
    }
    if (startValue>startValue+5) {   //如果大于块范围的结束值，则说明没有要查找的数
            return -1;
    }
    return startValue;
}
```

运行结果：

```
请输入您想要查找的数：
22
查找成功！您要找的数在数组中的位置是：7
```

分块查找的性能分析

分块查找算法的运行效率受两部分影响：查找块的操作和块内查找的操作。查找块的操作可以采用顺序查找，也可以采用折半查找（更优）；块内查找的操作采用顺序查找的方式。相比于折半查找，分块查找时间效率上更低一些；相比于顺序查找，由于在子表中进行，比较的子表个数会不同程度的减少，所有分块查找算法会更优。

总体来说，分块查找算法的效率介于顺序查找和折半查找之间。

静态树表查找算法及 C 语言实现

前面章节所介绍的有关在静态查找表中对特定关键字进行顺序查找、折半查找或者分块查找，都是在查找表中各关键字被查找概率相同的前提下进行的。

例如查找表中有 n 个关键字，表中每个关键字被查找的概率都是 1/n。在等概率的情况，使用折半查找算法的性能最优。

而在某些情况下，查找表中各关键字被查找的概率是不同的。例如水果商店中有很多种水果，对于不同的顾客来说，由于口味不同，各种水果可能被选择的概率是不同的。假设该顾客喜吃酸，那么相对于苹果和橘子，选择橘子的概率肯定要更高一些。

在查找表中各关键字查找概率不相同的情况下，对于使用折半查找算法，按照之前的方式进行，其查找的效率并不一定是最优的。例如，某查找表中有 5 个关键字，各关键字被查找到的概率分别为：0.1，0.2，0.1，0.4，0.2（全部关键字被查找概率和为 1 ），则根据之前介绍的折半查找算法，建立相应的判定树为（树中各关键字用概率表示）：

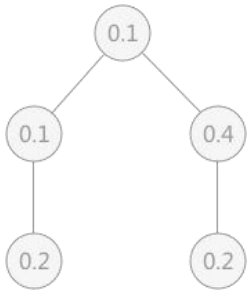


图 1 折半查找对应的判定树

折半查找查找成功时的平均查找长度的计算方式为：

```
ASL = 判定树中各结点的查找概率*所在层次
```

所以该平均查找长度为：

```
ASL=0.1*1 + 0.1*2 + 0.4*2 + 0.2*3 + 0.2*3 = 2.3
```

由于各关键字被查找的概率是不相同的，所以若在查找时遵循被查找关键字先和查找概率大的关键字进行比对，建立的判定树为：

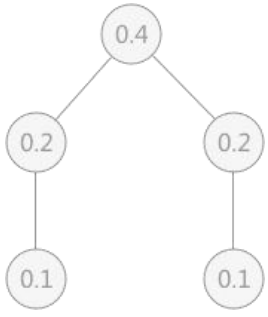


图 2 折半查找对应的新判定树

相应的平均查找长度为：

```
ASL=0.4*1 + 0.2*2 + 0.2*2 + 0.1*3 + 0.1*3=1.8
```

后者折半查找的效率要比前者高，所以在查找表中各关键字查找概率不同时，要考虑建立一棵查找性能最佳的判定树。若在只考虑查找成功的情况下，描述查找过程的判定树其带权路径长度之和（用 PH 表示）最小时，查找性能最优，称该二叉树为静态最优查找树。

带权路径之和的计算公式为：PH = 所有结点所在的层次数 * 每个结点对应的概率值。

但是由于构造最优查找树花费的时间代价较高，而且有一种构造方式创建的判定树的查找性能同最优查找树仅差 1% - 2%，称这种极度接近于最优查找树的二叉树为次优查找树。

次优查找树的构建方法

首先取出查找表中每个关键字及其对应的权值，采用如下公式计算出每个关键字对应的一个值：

$$\Delta P_i = \left| \sum_{j=i+1}^{h} w_j - \sum_{j=1}^{i-1} w_j \right|$$

其中 w_j 表示每个关键字的权值（被查找到的概率），h 表示关键字的个数。

表中有多少关键字，就会有多少个 $\triangle P_i$，取其中最小的做为次优查找树的根结点，然后将表中关键字从第 i 个关键字的位置分成两部分，分别作为该根结点的左子树和右子树。同理，左子树和右子树也这么处理，直到最后构成次优查找树完成。

代码实现为：

```
typedef int KeyType;//定义关键字类型
typedef struct{
    KeyType key;
}ElemType;//定义元素类型
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;

//定义变量
int i;
int min;
int dw;
//创建次优查找树，R 数组为查找表，sw 数组为存储的各关键字的概率（权值），low 和 high 表示的 sw 数组中的权值的范围
void SecondOptimal(BiTree T, ElemType R[], float sw[], int low, int high){
    //由有序表 R[low...high]及其累计权值表 sw（其中 sw[0]==0）递归构造次优查找树
    i = low;
    min = abs(sw[high] - sw[low]);
    dw = sw[high] + sw[low - 1];
```

```
    //选择最小的△Pi 值
    for (int j = low+1; j <=high; j++){
        if (abs(dw-sw[j]-sw[j-1])<min){
            i = j;
            min = abs(dw - sw[j] - sw[j - 1]);
        }
    }

    T = (BiTree)malloc(sizeof(BiTNode));
    T->data = R[i];//生成结点（第一次生成根）
    if (i == low) T->lchild = NULL;//左子树空
    else SecondOptimal(T->lchild, R, sw, low, i - 1);//构造左子树
    if (i == high) T->rchild = NULL;//右子树空
    else SecondOptimal(T->rchild, R, sw, i + 1, high);//构造右子树
}
```

完整事例演示

例如，一含有 9 个关键字的查找表及其相应权值如下表所示：

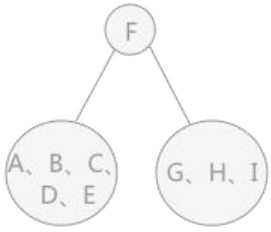
关键字：	A	B	C	D	E	F	G	H	I
权值：	1	1	2	5	3	4	4	3	5

则构建次优查找树的过程如下：

首先求出查找表中所有的 ▵P 的值，找出整棵查找表的根结点：

关键字：	A	B	C	D	E	F	G	H	I
△P：	27	25	22	15	7	0	8	15	23

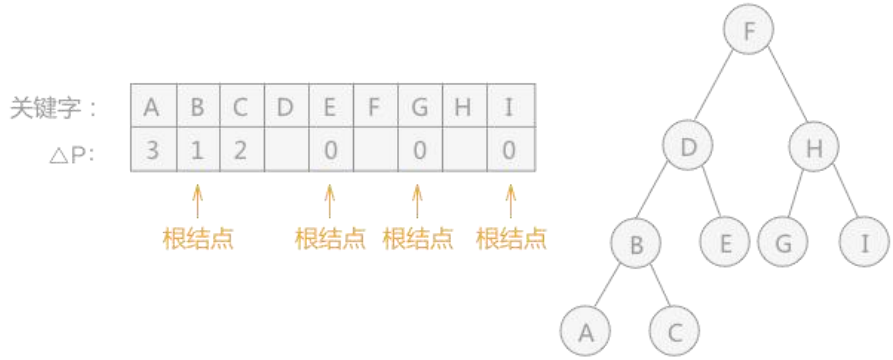
根结点



例如，关键字 F 的 △P 的计算方式为：从 G 到 I 的权值和 - 从 A 到 E 的权值和 = 4+3+5-1-1-2-5-3 = 0。

通过上图左侧表格得知，根结点为 F，以 F 为分界线，左侧子表为 F 结点的左子树，右侧子表为 F 结点的右子树（如上图右侧所示），继续查找左右子树的根结点：

通过重新分别计算左右两查找子表的 ▵P 的值，得知左子树的根结点为 D，右子树的根结点为 H （如上图右侧所示），以两结点为分界线，继续判断两根结点的左右子树：



通过计算，构建的次优查找树如上图右侧二叉树所示。

后边还有一步，判断关键字 A 和 C 在树中的位置，最后一步两个关键字的权值为 0 ，分别作为结点 B 的左孩子和右孩子，这里不再用图表示。

注意：在建立次优查找树的过程中，由于只根据的各关键字的 P 的值进行构建，没有考虑单个关键字的相应权值的大小，有时会出现根结点的权值比孩子结点的权值还小，此时就需要适当调整两者的位置。

总结

由于使用次优查找树和最优查找树的性能差距很小，构造次优查找树的算法的时间复杂度为 O(nlogn)，因此可以使用次优查找树表示概率不等的查找表对应的静态查找表（又称为静态树表）。

红黑树算法和应用(更高级的二叉查找树)

红黑树（R-B TREE，全称：Red-Black Tree），本身是一棵二叉查找树，在其基础上附加了两个要求：

1. 树中的每个结点增加了一个用于存储颜色的标志域；
2. 树中没有一条路径比其他任何路径长出两倍，整棵树要接近于“平衡”的状态。

这里所指的路径，指的是从任何一个结点开始，一直到其子孙的叶子结点的长度；接近于平衡：红黑树并不是平衡二叉树，只是由于对各路径的长度之差有限制，所以近似于平衡的状态。

红黑树对于结点的颜色设置不是任意的，需满足以下性质的二叉查找树才是红黑树：

- 树中的每个结点颜色不是红的，就是黑的；
- 根结点的颜色是黑的；

- 所有为 nil 的叶子结点的颜色是黑的；（注意：叶子结点说的只是为空（nil 或 NULL）的叶子结点！）
- 如果此结点是红的，那么它的两个孩子结点全部都是黑的；
- 对于每个结点，从该结点到到该结点的所有子孙结点的所有路径上包含有相同数目的黑结点；

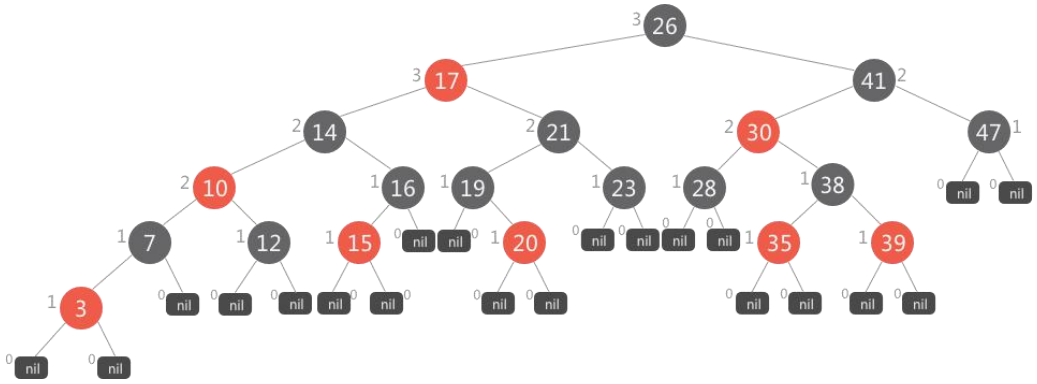


图 1 红黑树

注意：图中每个结点附带一个整形数值，表示的是此结点的黑高度（从该结点到其子孙结点中包含的黑结点数，用 bh(x) 表示（x 表示此结点）），nil 的黑高度为 0，颜色为黑色（在编程时为节省空间，所有的 nil 共用一个存储空间）。在计算黑高度时，也看做是一个黑结点。

红黑树中每个结点都有各自的黑高度，整棵树也有自己的黑高度，即为根结点的黑高度，例如图 1 中的红黑树的黑高度为 3。

对于一棵具有 n 个结点的红黑树，树的高度至多为：2lg(n+1)。

由此可推出红黑树进行查找操作时的时间复杂度为 O(lgn)，因为对于高度为 h 的二叉查找树的运行时间为 O(h)，而包含有 n 个结点的红黑树本身就是最高为 lgn（简化之后）的查找树（h=lgn），所以红黑树的时间复杂度为 O(lgn)。

红黑树本身作为一棵二叉查找树，所以其任务就是用于动态表中数据的插入和删除的操作。在进行该操作时，避免不了会破坏红黑树的结构，此时就需要进行适当的调整，使其重新成为一棵红黑树，可以从两个方面着手对树进行调整：

- 调整树中某些结点的指针结构；
- 调整树中某些结点的颜色；

红黑树的旋转

当使用红黑树进行插入或者删除结点的操作时，可能会破坏红黑树的 5 条性质，从而变成了一棵普通树，此时就可以通过对树中的某些子树进行旋转，从而使整棵树重新变为一棵红黑树。

旋转操作分为左旋和右旋，同二叉排序树转平衡二叉树的旋转原理完全相同。例如图 2 表示的是对一棵二叉查找树中局部子树进行左旋和右旋操作：

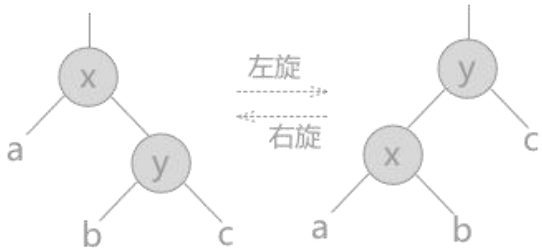


图 2 旋转操作

左旋：如图 2 所示，左旋时 y 结点变为该部分子树的根结点，同时 x 结点（连同其左子树 a）移动至 y 结点的左孩子。若 y 结点有左孩子 b，由于 x 结点需占用其位置，所以调整至 x 结点的右孩子处。

左旋操作的具体实现函数：

```
//T 表示为树根，x 表示需要进行左旋的子树的根结点
void rbTree_left_rotate( RBT_Root* T, RB_TREE* x){
    RB_TREE* y = x->right;//找到根结点的右子树

    x->right = y->left;//将右子树的左孩子移动至结点 x 的右孩子处
    if(x->right != T->nil){//如果 x 的右子树不是 nil，需重新连接 右子树的双亲结点为 x
        x->right->p = x;
    }
    y->p = x->p;//设置 y 的双亲结点为 x 的双亲结点
    //重新设置 y 的双亲结点同 y 的连接，分为 2 种情况：1、原 x 结点本身就是整棵树的数根结点，此时只需要将 T 指针指向 y；2、根据 y 中关键字同其父结点关键字的值的大小，判断 y 是父结点的左孩子还是右孩子
    if(y->p == T->nil){
        T->root = y;
    }else if(y->key < y->p->key){
        y->p->left = y;
    }else{
        y->p->right = y;
    }
    y->left = x;//将 x 连接给 y 结点的左孩子处
    x->p = y;//设置 x 的双亲结点为 y。
}
```

右旋：如图 2 所示，同左旋是同样的道理，x 结点变为根结点，同时 y 结点连同其右子树 c 作为 x 结点的右子树，原 x 结点的右子树 b 变为 y 结点的左子树。

右旋的具体代码实现：

```
void rbTree_right_rotate( RBT_Root* T, RB_TREE* x){
```

```
    RB_TREE * y = x->left;
    x->left = y->right;
    if(T->nil != x->left){
        x->left->p = x;
    }
    y->p = x->p;
    if(y->p == T->nil){
        T->root = y;
    }else if(y->key < y->p->key){
        y->p->left= y;
    }else{
        y->p->right = y;
    }
    y->right = x;
    x->p = y;
}
```

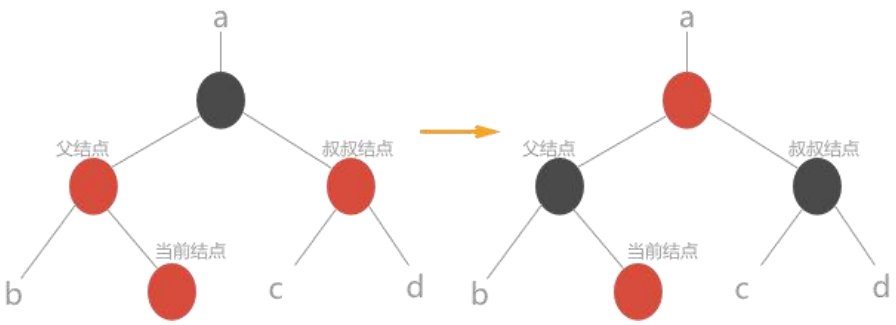
红黑树中插入新结点

当创建一个红黑树或者向已有红黑树中插入新的数据时，只需要按部就班地执行以下 3 步：

- 由于红黑树本身是一棵二叉查找树，所以在插入新的结点时，完全按照二叉查找树插入结点的方法，找到新结点插入的位置；
- 将新插入的结点结点初始化，颜色设置为红色后插入到指定位置；（将新结点初始化为红色插入后，不会破坏红黑树第 5 条的性质）
- 由于插入新的结点，可能会破坏红黑树第 4 条的性质（若其父结点颜色为红色，就破坏了红黑树的性质），此时需要调整二叉查找树，想办法通过旋转以及修改树中结点的颜色，使其重新成为红黑树！

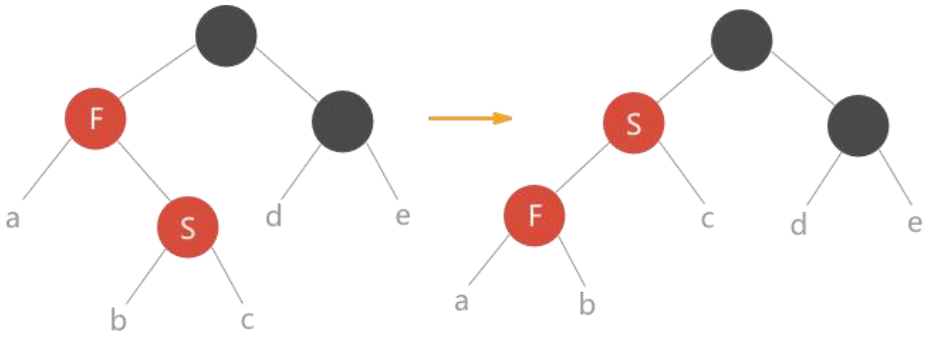
插入结点的第 1 步和第 2 步都非常简单，关键在于最后一步对树的调整！在红黑树中插入结点时，根据插入位置的不同可分为以下 3 种情况：

1. 插入位置为整棵树的树根。处理办法：只需要将插入结点的颜色改为黑色即可。
2. 插入位置的双亲结点的颜色为黑色。处理方法：此种情况不需要做任何工作，新插入的颜色为红色的结点不会破坏红黑树的性质。
3. 插入位置的双亲结点的颜色为红色。处理方法：由于插入结点颜色为红色，其双亲结点也为红色，破坏了红黑树第 4 条性质，此时需要结合其祖父结点和祖父结点的另一个孩子结点（父结点的兄弟结点，此处称为“叔叔结点”）的状态，分为 3 种情况讨论:
    - 当前结点的父节点是红色，且“叔叔结点”也是红色：破坏了红黑树的第 4 条性质，解决方案为：将父结点颜色改为黑色；将叔叔结点颜色改为黑色；将祖父结点颜色改为红色；下一步将祖父结点认做当前结点，继续判断，处理结果如下图所示：



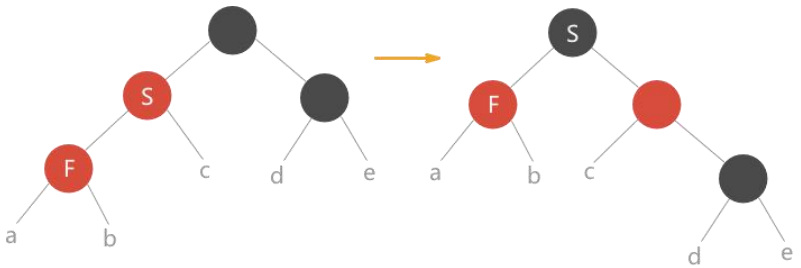
分析：此种情况下，由于父结点和当前结点颜色都是红色，所以为了不产生冲突，将父结点的颜色改为黑色。但是虽避免了破坏第 4 条，但是却导致该条路径上的黑高度增加了 1 ，破坏了第 5 条性质。但是在将祖父结点颜色改为红色、叔叔结点颜色改为黑色后，该部分子树没有破坏第 5 条性质。但是由于将祖父结点的颜色改变，还需判断是否破坏了上层树的结构，所以需要将祖父结点看做当前结点，继续判断。

- 当前结点的父结点颜色为红色，叔叔结点颜色为黑色，且当前结点是父结点的右孩子。解决方案：将父结点作为当前结点做左旋操作。



提示：在进行以父结点为当前结点的左旋操作后，此种情况就转变成了第 3 种情况，处理过程跟第 3 种情况同步进行。

- 当前结点的父结点颜色为红色，叔叔结点颜色为黑色，且当前结点是父结点的左孩子。解决方案：将父结点颜色改为黑色，祖父结点颜色改为红色，从祖父结点处进行右旋处理。如下图所示：



分析：在此种情况下，由于当前结点 F 和父结点 S 颜色都为红色，违背了红黑树的性质 4，此时可以将 S 颜色改为黑色，有违反了性质 5，因为所有通过 S 的路径其黑高度都增加了 1 ，所以需要将其祖父结点颜色设为红色后紧接一个右旋，这样这部分子树有成为了红黑树。（上图中的有图虽看似不是红黑树，但是只是整棵树的一部分，以 S 为根结点的子树一定是一棵红黑树）

红黑树中插入结点的具体实现代码：

```
void RB_Insert_Fixup(RBT_Root* T, RB_TREE* x){
```

```
    //首先判断其父结点颜色为红色时才需要调整；为黑色时直接插入即可，不需要调整
    while (x->p->color == RED) {
        //由于还涉及到其叔叔结点，所以此处需分开讨论，确定父结点是祖父结点的左孩子还是右孩子
        if (x->p == x->p->p->left) {
            RB_TREE * y = x->p->p->right;//找到其叔叔结点
            //如果叔叔结点颜色为红色，此为第 1 种情况，处理方法为：父结点颜色改为黑色；叔叔结点颜色改为黑色；祖父结点颜色改为红色，将祖父结点赋值为当前结点，继续判断；
            if (y->color == RED) {
                x->p->color = BLACK;
                y->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            }else{
                //反之，如果叔叔结点颜色为黑色，此处需分为两种情况：1、当前结点时父结点的右孩子；2、当前结点是父结点的左孩子
                if (x == x->p->right) {
                    //第 2 种情况：当前结点时父结点的右孩子。解决方案：将父结点作为当前结点做左旋操作。
                    x = x->p;
                    rbTree_left_rotate(T, x);
                }else{
                    //第 3 种情况：当前结点是父结点的左孩子。解决方案：将父结点颜色改为黑色，祖父结点颜色改为红色，从祖父结点处进行右旋处理。
                    x->p->color = BLACK;
                    x->p->p->color = RED;
                    rbTree_right_rotate(T, x->p->p);
                }
            }
        }else{//如果父结点时祖父结点的右孩子，换汤不换药，只需将以上代码部分中的 left 改为 right 即可，道理是一样的。
            RB_TREE * y = x->p->p->left;
            if (y->color == RED) {
                x->p->color = BLACK;
                y->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            }else{
                if (x == x->p->left) {
                    x = x->p;
                    rbTree_right_rotate(T, x);
```

```
                }else{
                    x->p->color = BLACK;
                    x->p->p->color = RED;
                    rbTree_left_rotate(T, x->p->p);
                }
            }
        }
    }
    T->root->color = BLACK;
}
//插入操作分为 3 步：1、将红黑树当二叉查找树，找到其插入位置；2、初始化插入结点，将新结点的颜色设为红色；3、通过调用调整函数，将二叉查找树重新改为红黑树
void rbTree_insert(RBT_Root**T, int k){
    //1、找到其要插入的位置。解决思路为：从树的根结点开始，通过不断的同新结点的值进行比较，最终找到插入位置
    RB_TREE * x, *p;
    x = (*T)->root;
    p = x;

    while(x != (*T)->nil){
        p = x;
        if(k<x->key){
            x = x->left;
        }else if(k>x->key){
            x = x->right;
        }else{
            printf("\n%d 已存在\n",k);
            return;
        }
    }
    //初始化结点，将新结点的颜色设为红色
    x = (RB_TREE *)malloc(sizeof(RB_TREE));
    x->key = k;
    x->color = RED;
    x->left = x->right =(*T)->nil;
    x->p = p;

    //对新插入的结点，建立与其父结点之间的联系
```

```
    if((*T)->root == (*T)->nil){
        (*T)->root = x;
    }else if(k < p->key){
        p->left = x;
    }else{
        p->right = x;
    }
    //3、对二叉查找树进行调整
    RB_Insert_Fixup((*T),x);
}
```

红黑树中删除结点

在红黑树中删除结点，思路更简单，只需要完成 2 步操作：

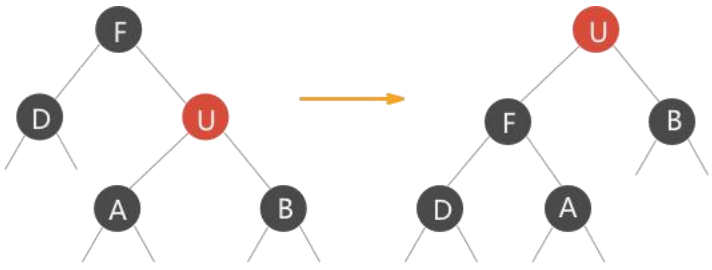
1. 将红黑树按照二叉查找树删除结点的方法删除指定结点；
2. 重新调整删除结点后的树，使之重新成为红黑树；（还是通过旋转和重新着色的方式进行调整）

在二叉查找树删除结点时，分为 3 种情况：

- 若该删除结点本身是叶子结点，则可以直接删除；
- 若只有一个孩子结点（左孩子或者右孩子），则直接让其孩子结点顶替该删除结点；
- 若有两个孩子结点，则找到该结点的右子树中值最小的叶子结点来顶替该结点，然后删除这个值最小的叶子结点。

以上三种情况最终都需要删除某个结点，此时需要判断删除该结点是否会破坏红黑树的性质。判断的依据是：

1. 如果删除结点的颜色为红色，则不会破坏；
2. 如果删除结点的颜色为黑色，则肯定会破坏红黑树的第 5 条性质，此时就需要对树进行调整，调整方案分 4 种情况讨论：
    - 删除结点的兄弟结点颜色是红色，调整措施为：将兄弟结点颜色改为黑色，父亲结点改为红色，以父亲结点来进行左旋操作，同时更新删除结点的兄弟结点（左旋后兄弟结点发生了变化），如下图所示：



    - 删除结点的兄弟结点及其孩子全部都是黑色的，调整措施为：将删除结点的兄弟结点设为红色，同时设置删除结点的父结点标记为新的结点，继续判断；
    - 删除结点的兄弟结点是黑色，其左孩子是红色，右孩子是黑色。调整措施为：将兄弟结点设为红色，兄弟结点的左孩子结点设为黑色，以兄弟结点为准进行右旋操作，最终更新删除结点的兄弟结点；

- 删除结点的兄弟结点是黑色，其右孩子是红色（左孩子不管是什么颜色），调整措施为：将删除结点的父结点的颜色赋值给其兄弟结点，然后再设置父结点颜色为黑色，兄弟结点的右孩子结点为黑色，根据其父结点做左旋操作，最后设置替换删除结点的结点为根结点；

红黑树删除结点具体实现代码为：

```
void rbTree_transplant(RBT_Root* T, RB_TREE* u, RB_TREE* v){
    if(u->p == T->nil){
        T->root = v;
    }else if(u == u->p->left){
        u->p->left=v;
    }else{
        u->p->right=v;
    }
    v->p = u->p;
}
void RB_Delete_Fixup(RBT_Root**T,RB_TREE*x){
    while(x != (*T)->root && x->color == BLACK){
        if(x == x->p->left){
            RB_TREE* w = x->p->right;
            //第 1 种情况：兄弟结点是红色的
            if(RED == w->color){
                w->color = BLACK;
                w->p->color = RED;
                rbTree_left_rotate((*T),x->p);
                w = x->p->right;
            }
            //第 2 种情况：兄弟是黑色的，并且兄弟的两个儿子都是黑色的。
            if(w->left->color == BLACK && w->right->color == BLACK){
                w->color = RED;
                x = x->p;
            }
            //第 3 种情况
            if(w->left->color == RED && w->right->color == BLACK){
                w->left->color = BLACK;
                w->color = RED;
                rbTree_right_rotate((*T),w);
                w = x->p->right;
```

```
            }
            //第 4 种情况
            if (w->right->color == RED) {
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color = BLACK;
                rbTree_left_rotate((*T),x->p);
                x = (*T)->root;
            }
        }else{
            RB_TREE* w = x->p->left;
            //第 1 种情况
            if(w->color == RED){
                w->color = BLACK;
                x->p->color = RED;
                rbTree_right_rotate((*T),x->p);
                w = x->p->left;
            }
            //第 2 种情况
            if(w->left->color == BLACK && w->right->color == BLACK){
                w->color = RED;
                x = x->p;
            }
            //第 3 种情况
            if(w->left->color == BLACK && w->right->color == RED){
                w->color = RED;
                w->right->color = BLACK;
                w = x->p->left;
            }
            //第 4 种情况
            if (w->right->color == BLACK){
                w->color=w->p->color;
                x->p->color = BLACK;
                w->left->color = BLACK;
                rbTree_right_rotate((*T),x->p);
                x = (*T)->root;
            }
```

```
            }
        }
        x->color = BLACK;//最终将根结点的颜色设为黑色
}
void rbTree_delete(RBT_Root* *T, int k){
        if(NULL == (*T)->root){
                return ;
        }
        //找到要被删除的结点
        RB_TREE * toDelete = (*T)->root;
        RB_TREE * x = NULL;
        //找到值为 k 的结点
        while(toDelete != (*T)->nil && toDelete->key != k){
                if(k<toDelete->key){
                        toDelete = toDelete->left;
                }else if(k>toDelete->key){
                        toDelete = toDelete->right;
                }
        }
        if(toDelete == (*T)->nil){
                printf("\n%d  不存在\n",k);
                return;
        }
        //如果两个孩子，就找到右子树中最小的结点，将之代替，然后直接删除该结点即可
        if(toDelete->left != (*T)->nil && toDelete->right != (*T)->nil){
                RB_TREE* alternative = rbt_findMin((*T), toDelete->right);
                k = toDelete->key = alternative->key;//这里只对值进行复制，并不复制颜色，以免破坏红黑树的性质
                toDelete = alternative;
        }
        //如果只有一个孩子结点（只有左孩子或只有右孩子），直接用孩子结点顶替该结点位置即可（没有孩子结点的也走此判断
语句）。
        if(toDelete->left == (*T)->nil){
                x = toDelete->right;
                rbTree_transplant((*T),toDelete,toDelete->right);
        }else if(toDelete->right == (*T)->nil){
                x = toDelete->left;
                rbTree_transplant((*T),toDelete,toDelete->left);
```

```
    }
    //在删除该结点之前，需判断此结点的颜色：如果是红色，直接删除，不会破坏红黑树；若是黑色，删除后会破坏红黑树的第 5 条性质，需要对树做调整。
    if(toDelete->color == BLACK){
        RB_Delete_Fixup(T,x);
    }
    //最终可以彻底删除要删除的结点，释放其占用的空间
    free(toDelete);
}
```

本节完整实现代码

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

typedef enum {RED, BLACK} ColorType;
typedef struct RB_TREE{
    int key;
    struct  RB_TREE * left;
    struct  RB_TREE * right;
    struct  RB_TREE * p;
    ColorType color;
}RB_TREE;

typedef struct RBT_Root{
    RB_TREE* root;
    RB_TREE* nil;
}RBT_Root;

RBT_Root* rbTree_init(void);
void rbTree_insert(RBT_Root* *T, int k);
void rbTree_delete(RBT_Root* *T, int k);

void rbTree_transplant(RBT_Root* T, RB_TREE* u, RB_TREE* v);

void rbTree_left_rotate( RBT_Root* T, RB_TREE* x);
void rbTree_right_rotate( RBT_Root* T, RB_TREE* x);
```

```
void rbTree_inPrint(RBT_Root* T, RB_TREE* t);
void rbTree_prePrint(RBT_Root * T, RB_TREE* t);
void rbTree_print(RBT_Root* T);

RB_TREE* rbt_findMin(RBT_Root * T, RB_TREE* t);
RB_TREE* rbt_findMin(RBT_Root * T, RB_TREE* t){
    if(t == T->nil){
        return T->nil;
    }
    while(t->left != T->nil){
        t = t->left;
    }
    return t;
}
RBT_Root* rbTree_init(void){
    RBT_Root* T;
    T = (RBT_Root*)malloc(sizeof(RBT_Root));
    T->nil = (RB_TREE*)malloc(sizeof(RB_TREE));
    T->nil->color = BLACK;
    T->nil->left = T->nil->right = NULL;
    T->nil->p = NULL;
    T->root = T->nil;
    return T;
}

void RB_Insert_Fixup(RBT_Root* T, RB_TREE* x){
    //首先判断其父结点颜色为红色时才需要调整；为黑色时直接插入即可，不需要调整
    while (x->p->color == RED) {
        //由于还涉及到其叔叔结点，所以此处需分开讨论，确定父结点是祖父结点的左孩子还是右孩子
        if (x->p == x->p->p->left) {
            RB_TREE * y = x->p->p->right;//找到其叔叔结点
            //如果叔叔结点颜色为红色，此为第 1 种情况，处理方法为：父结点颜色改为黑色；叔叔结点颜色改为黑色；祖父结点颜色改为红色，将祖父结点赋值为当前结点，继续判断；
            if (y->color == RED) {
                x->p->color = BLACK;
                y->color = BLACK;
                x->p->p->color = RED;
```

```
                x = x->p->p;
            }else{
                //反之，如果叔叔结点颜色为黑色，此处需分为两种情况：1、当前结点时父结点的右孩子；2、当前结点是父结点的左孩子
                if (x == x->p->right) {
                    //第 2 种情况：当前结点时父结点的右孩子。解决方案：将父结点作为当前结点做左旋操作。
                    x = x->p;
                    rbTree_left_rotate(T, x);
                }else{
                    //第 3 种情况：当前结点是父结点的左孩子。解决方案：将父结点颜色改为黑色，祖父结点颜色改为红色，从祖父结点处进行右旋处理。
                    x->p->color = BLACK;
                    x->p->p->color = RED;
                    rbTree_right_rotate(T, x->p->p);
                }
            }
        }else{//如果父结点时祖父结点的右孩子，换汤不换药，只需将以上代码部分中的 left 改为 right 即可，道理是一样的。
            RB_TREE * y = x->p->p->left;
            if (y->color == RED) {
                x->p->color = BLACK;
                y->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            }else{
                if (x == x->p->left) {
                    x = x->p;
                    rbTree_right_rotate(T, x);
                }else{
                    x->p->color = BLACK;
                    x->p->p->color = RED;
                    rbTree_left_rotate(T, x->p->p);
                }
            }
        }
    }
    T->root->color = BLACK;
}
```

```
//插入操作分为 3 步：1、将红黑树当二叉查找树，找到其插入位置；2、初始化插入结点，将新结点的颜色设为红色；3、通过调用调整函数，将二叉查找树重新改为红黑树
void rbTree_insert(RBT_Root**T, int k){
    //1、找到其要插入的位置。解决思路为：从树的根结点开始，通过不断的同新结点的值进行比较，最终找到插入位置
    RB_TREE * x, *p;
    x = (*T)->root;
    p = x;

    while(x != (*T)->nil){
        p = x;
        if(k<x->key){
            x = x->left;
        }else if(k>x->key){
            x = x->right;
        }else{
            printf("\n%d 已存在\n",k);
            return;
        }
    }
    //初始化结点，将新结点的颜色设为红色
    x = (RB_TREE *)malloc(sizeof(RB_TREE));
    x->key = k;
    x->color = RED;
    x->left = x->right =(*T)->nil;
    x->p = p;
    //对新插入的结点，建立与其父结点之间的联系
    if((*T)->root == (*T)->nil){
        (*T)->root = x;
    }else if(k < p->key){
        p->left = x;
    }else{
        p->right = x;
    }
    //3、对二叉查找树进行调整
    RB_Insert_Fixup((*T),x);
}
void rbTree_transplant(RBT_Root* T, RB_TREE* u, RB_TREE* v){
```

```
    if(u->p == T->nil){
        T->root = v;
    }else if(u == u->p->left){
        u->p->left=v;
    }else{
        u->p->right=v;
    }
    v->p = u->p;
}
void RB_Delete_Fixup(RBT_Root**T,RB_TREE*x){
    while(x != (*T)->root && x->color == BLACK){
        if(x == x->p->left){
            RB_TREE* w = x->p->right;
            //第 1 种情况：兄弟结点是红色的
            if(RED == w->color){
                w->color = BLACK;
                w->p->color = RED;
                rbTree_left_rotate((*T),x->p);
                w = x->p->right;
            }
            //第 2 种情况：兄弟是黑色的，并且兄弟的两个儿子都是黑色的。
            if(w->left->color == BLACK && w->right->color == BLACK){
                w->color = RED;
                x = x->p;
            }
            //第 3 种情况
            if(w->left->color == RED && w->right->color == BLACK){
                w->left->color = BLACK;
                w->color = RED;
                rbTree_right_rotate((*T),w);
                w = x->p->right;
            }
            //第 4 种情况
            if (w->right->color == RED) {
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color = BLACK;
```

```
                rbTree_left_rotate((*T),x->p);
                x = (*T)->root;
            }
        }else{
            RB_TREE* w = x->p->left;
            //第 1 种情况
            if(w->color == RED){
                w->color = BLACK;
                x->p->color = RED;
                rbTree_right_rotate((*T),x->p);
                w = x->p->left;
            }
            //第 2 种情况
            if(w->left->color == BLACK && w->right->color == BLACK){
                w->color = RED;
                x = x->p;
            }
            //第 3 种情况
            if(w->left->color == BLACK && w->right->color == RED){
                w->color = RED;
                w->right->color = BLACK;
                w = x->p->left;
            }
            //第 4 种情况
            if (w->right->color == BLACK){
                w->color=w->p->color;
                x->p->color = BLACK;
                w->left->color = BLACK;
                rbTree_right_rotate((*T),x->p);
                x = (*T)->root;
            }
        }
    }
    x->color = BLACK;//最终将根结点的颜色设为黑色
}
void rbTree_delete(RBT_Root* *T, int k){
    if(NULL == (*T)->root){
```

```
        return ;
    }
    //找到要被删除的结点
    RB_TREE * toDelete = (*T)->root;
    RB_TREE * x = NULL;
    //找到值为 k 的结点
    while(toDelete != (*T)->nil && toDelete->key != k){
        if(k<toDelete->key){
            toDelete = toDelete->left;
        }else if(k>toDelete->key){
            toDelete = toDelete->right;
        }
    }
    if(toDelete == (*T)->nil){
        printf("\n%d  不存在\n",k);
        return;
    }
    //如果两个孩子，就找到右子树中最小的结点，将之代替，然后直接删除该结点即可
    if(toDelete->left != (*T)->nil && toDelete->right != (*T)->nil){
        RB_TREE* alternative = rbt_findMin((*T), toDelete->right);
        k = toDelete->key = alternative->key;//这里只对值进行复制，并不复制颜色，以免破坏红黑树的性质
        toDelete = alternative;
    }
    //如果只有一个孩子结点（只有左孩子或只有右孩子），直接用孩子结点顶替该结点位置即可（没有孩子结点的也走此判断语句）。
    if(toDelete->left == (*T)->nil){
        x = toDelete->right;
        rbTree_transplant((*T),toDelete,toDelete->right);
    }else if(toDelete->right == (*T)->nil){
        x = toDelete->left;
        rbTree_transplant((*T),toDelete,toDelete->left);
    }
    //在删除该结点之前，需判断此结点的颜色：如果是红色，直接删除，不会破坏红黑树；若是黑色，删除后会破坏红黑树的第 5 条性质，需要对树做调整。
    if(toDelete->color == BLACK){
        RB_Delete_Fixup(T,x);
    }
```

```
    //最终可以彻底删除要删除的结点，释放其占用的空间
    free(toDelete);
}
//T 表示为树根，x 表示需要进行左旋的子树的根结点
void rbTree_left_rotate( RBT_Root* T, RB_TREE* x){
    RB_TREE* y = x->right;//找到根结点的右子树

    x->right = y->left;//将右子树的左孩子移动至结点 x 的右孩子处
    if(x->right != T->nil){//如果 x 的右子树不是 nil，需重新连接 右子树的双亲结点为 x
        x->right->p = x;
    }
    y->p = x->p;//设置 y 的双亲结点为 x 的双亲结点
    //重新设置 y 的双亲结点同 y 的连接，分为 2 种情况：1、原 x 结点本身就是整棵树的数根结点，此时只需要将 T 指针指向 y；2、根据 y 中关键字同其父结点关键字的值的大小，判断 y 是父结点的左孩子还是右孩子
    if(y->p == T->nil){
        T->root = y;
    }else if(y->key < y->p->key){
        y->p->left = y;
    }else{
        y->p->right = y;
    }
    y->left = x;//将 x 连接给 y 结点的左孩子处
    x->p = y;//设置 x 的双亲结点为 y。
}
void rbTree_right_rotate( RBT_Root* T, RB_TREE* x){
    RB_TREE * y = x->left;
    x->left = y->right;
    if(T->nil != x->left){
        x->left->p = x;
    }
    y->p = x->p;
    if(y->p == T->nil){
        T->root = y;
    }else if(y->key < y->p->key){
        y->p->left= y;
    }else{
        y->p->right = y;
```

```
    }
    y->right = x;
    x->p = y;
}
void rbTree_prePrint(RBT_Root* T, RB_TREE* t){
    if(T->nil == t){
        return;
    }
    if(t->color == RED){
        printf("%dR ",t->key);
    }else{
        printf("%dB ",t->key);
    }
    rbTree_prePrint(T,t->left);
    rbTree_prePrint(T,t->right);
}
void rbTree_inPrint(RBT_Root* T, RB_TREE* t){
    if(T->nil == t){
        return ;
    }
    rbTree_inPrint(T,t->left);
    if(t->color == RED){
        printf("%dR ",t->key);
    }else{
        printf("%dB ",t->key);
    }
    rbTree_inPrint(T,t->right);
}

//输出红黑树的前序遍历和中序遍历的结果
void rbTree_print(RBT_Root* T){
    printf("前序遍历 ： ");
    rbTree_prePrint(T,T->root);
    printf("\n");
    printf("中序遍历 ： ");
    rbTree_inPrint(T,T->root);
    printf("\n");
```

```
}

int main(){
    RBT_Root* T = rbTree_init();

    rbTree_insert(&T,3);
    rbTree_insert(&T,5);
    rbTree_insert(&T,1);
    rbTree_insert(&T,2);
    rbTree_insert(&T,4);
    rbTree_print(T);
    printf("\n");
    rbTree_delete(&T,3);
    rbTree_print(T);

    return 0;
}
```

运行结果：

```
前序遍历 ： 3B 1B 2R 5B 4R
中序遍历 ： 1B 2R 3B 4R 5B

前序遍历 ： 4B 1B 2R 5B
中序遍历 ： 1B 2R 4B 5B
```

总结

本节介绍的红黑树，虽隶属于二叉查找树，但是二叉查找树的时间复杂度会受到其树深度的影响，而红黑树可以保证在最坏情况下的时间复杂度仍为 O(lgn)。当数据量多到一定程度时，使用红黑树比二叉查找树的效率要高。

同平衡二叉树相比较，红黑树没有像平衡二叉树对平衡性要求的那么苛刻，虽然两者的时间复杂度相同，但是红黑树在实际测算中的速度要更胜一筹！

提示：平衡二叉树的时间复杂度是 O(logn)，红黑树的时间复杂度为 O(lgn)，两者都表示的都是时间复杂度为对数关系（lg 函数为底是 10 的对数，用于表示时间复杂度时可以忽略）。

B-树及其基本操作（插入和删除）详解

前面介绍了二叉排序树和平衡二叉树，本节开始介绍两种用于查找功能的树数据结构——B-树和 B+树。

什么是 B-树？

B-树，有时又写为 B_树（其中的“-”或者“_”只是连字符，并不读作“B 减树”），一颗 m 阶的 B-树，或者本身是空树，否则必须满足以下特性：

- 树中每个结点至多有 m 棵子树；
- 若根结点不是叶子结点，则至少有两棵子树；
- 除根之外的所有非终端结点至少有棵子树；
- 所有的非终端结点中包含下列信息数据：（n，A_0，K_1，A_1，K_2，A_2，…，K_n，A_n）；

  n 表示结点中包含的关键字的个数，取值范围是：$\lceil m/2 \rceil -1 \le n \le m-1$。$K_i$（i 从 1 到 n）为关键字，且 $K_i < K_{i+1}$；A_i 代表指向子树根结点的指针，且指针 A_{i-1} 所指的子树中所有结点的关键字都小于 K_i，A_n 所指子树中所有的结点的关键字都大于 K_n。

图 1 结点结构

如图 1 所示，当前结点中有 4 个关键字，之间的关系为：$K_1<K_2<k_3<K_4$。同时对于 A_0 指针指向的子树中的所有关键字来说，其值都要比 K_1 小；而 A_1 指向的子树中的所有的关键字的值，都比 K_1 大，但是都要比 K_2 小。

- 所有的叶子结点都出现在同一层次，实际上这些结点都不存在，指向这些结点的指针都为 NULL；

  例如图 2 所示就是一棵 4 阶的 B-树，这棵树的深度为 4 ：

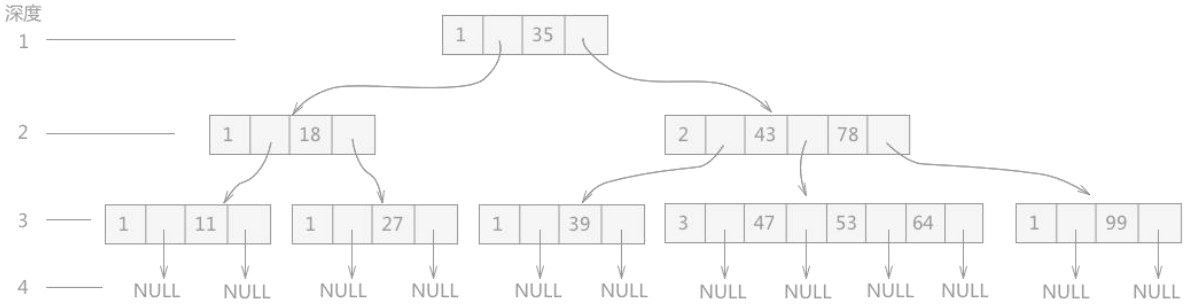


图 2 深度为 4 的 B-树

在使用 B-树进行查找操作时，例如在如图 2 所示的 B-树中查找关键字 47 的过程为：

1. 从整棵树的根结点开始，由于根结点只有一个关键字 35，且 35 < 47 ，所以如果 47 存在于这棵树中，肯定位于 A_1 指针指向的右子树中；
2. 然后顺着指针找到存有关键字 43 和 78 的结点，由于 43 < 47 < 78，所以如果 47 存在，肯定位于 A_1 所指的子树中；
3. 然后找到存有 47、53 和 64 三个关键字的结点，最终找到 47 ，查找操作结束；

以图 2 中的 B-树为例，若查找到深度为 3 的结点还没结束，则会进入叶子结点，但是由于叶子结点本身不存储任何信息，全部为 NULL，所以查找失败。

B-树中插入关键字（构建 B-树）

B-树也是从空树开始，通过不断地插入新的数据元素构建的。但是 B-树构建的过程同前面章节的二叉排序树和平衡二叉树不同，B-树在插入新的数据元素时并不是每次都向树中插入新的结点。

因为对于 m 阶的 B-树来说，在定义中规定所有的非终端结点（终端结点即叶子结点，其关键字个数为 0）中包含关键字的个数的范围是[⌈ m/2⌉ -1,m-1]，所以在插入新的数据元素时，首先向最底层的某个非终端结点中添加，如果该结点中的关键字个数没有超过 m-1，则直接插入成功，否则还需要继续对该结点进行处理。

假设现在图 3 的基础上插入 4 个关键字 30、26、85 和 7：

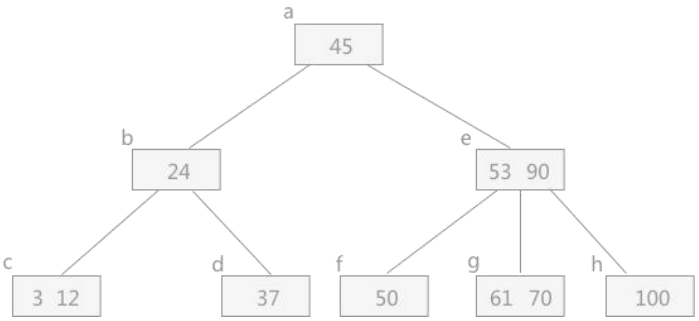


图 3 3 阶 B-树（深度为 4，省略了叶子结点）

插入关键字 30 ：从根结点开始，由于 30 < 45，所以要插入到以 b 结点为根结点的子树中，再由于 24 < 30，插入到以 d 结点为根结点的子树中，由于 d 结点中的关键字个数小于 m-1=2，所以可以将关键字 30 直接插入到 d 结点中。结果如下图所示：

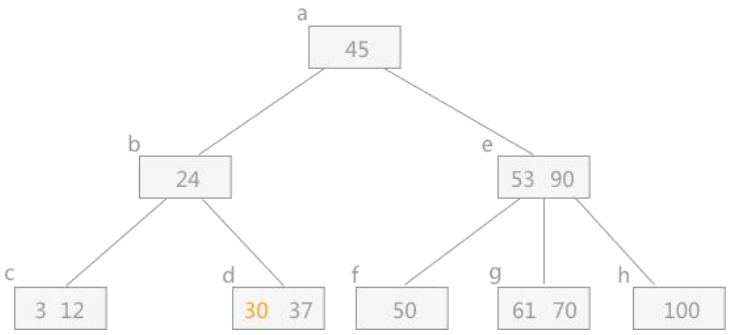


图 4 插入关键字 30 后的 B-树

插入关键字 26：从根结点开始，经过逐个比较，最终判定 26 还是插入到 d 结点中，但是由于 d 结点中关键字的个数超过了 2，所以需要做如下操作：

- 关键字 37 及其左右两个指针存储到新的结点中，假设为 d’ 结点；
- 关键字 30 存储到其双亲结点 b 中，同时设置关键字 30 右侧的指针指向 d’ ；

经过以上操作后，插入 26 后的 B-树为：

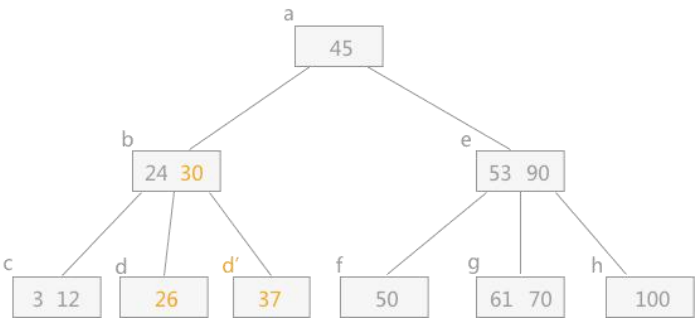


图 5 插入关键字 26 后的 B-树

插入关键字 85：从根结点开始，经过逐个比较，最终判定插入到 g 结点中，同样需要对 g 做分裂操作：

- 关键字 85 及其左右两个指针存储到新的结点中，假设为 g’ 结点；
- 关键字 70 存储到其双亲结点 e 中，同时设置 70 的右侧指针指向 g’ ；

经过以上操作后，插入 85 后的结果图为：

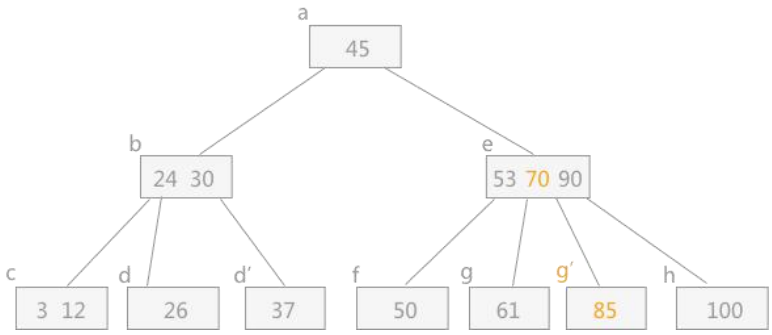


图 6 插入 85 的效果图

图 6 中，由于关键字 70 调整到其双亲结点中，使得其 e 结点中的关键字个数超过了 2，所以还需进一步调整：

- 将 90 及其左右指针存储到一个新的结点中，假设为 e’ 结点；
- 关键字 70 存储到其双亲结点 a 中，同时其右侧指针指向 e’ ；

最终插入关键字 85 后的 B-树为：

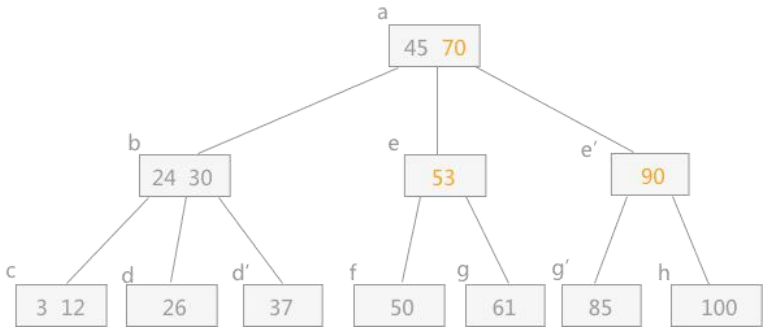


图 7 插关键字 85 后的 B-树

插入关键字 7：从根结点开始依次做判断，最终判定在 c 结点中添加，添加后发现 c 结点需要分裂，分裂规则同上面的方式一样，结果导致关键字 7 存储到其双亲结点 b 中；后 b 结点分裂，关键字 24 存储到结点 a 中；结点 a 同样需要做分裂操作，最终 B-树为：

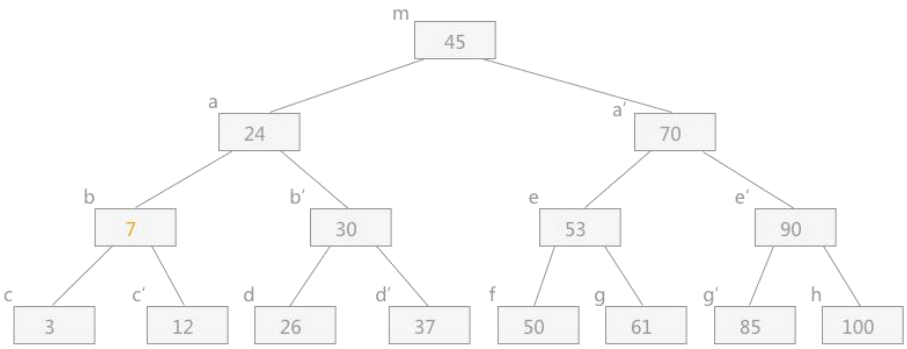


图 8 插入关键字 7 后的 B-树

通过上边的例子，可以总结出一下结论：在构建 B-树的过程中，假设 p 结点中已经有 m-1 个关键字，当再插入一个关键字之后，此结点分裂为两个结点，如下图所示：

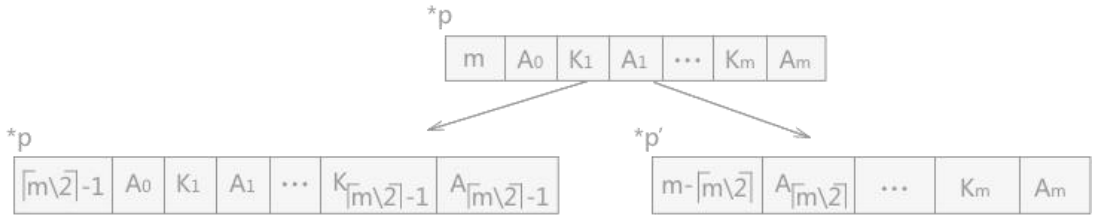


图 9 B-树构成过程中的“分裂”

提示：如图 9 所示，结点分裂为两个结点的同时，还分裂出来一个关键字 $K_{\lceil m/2 \rceil}$，存储到其双亲结点中。

B-树中删除关键字

在 B-树种删除关键字时，首先前提是找到该关键字所在结点，在做删除操作的时候分为两种情况，一种情况是删除结点为 B-树的非终端结点（不处在最后一层）；另一种情况是删除结点为 B-树最后一层的非终端结点。

> 例如图 3 来说，关键字 24、45、53、90 属于不处在最后一层的非终端结点，关键字 3、12、37 等同属于最后一层的非终端结点。

如果该结点为非终端结点且不处在最后一层，假设用 K_i 表示，则只需要找到指针 A_i 所指子树中最小的一个关键字代替 K_i，同时将该最小的关键字删除即可。

> 例如图 3 中，如果要删除关键字 45 ，只需要使用关键字 50 代替 45 ，同时删除 f 结点中的 50 即可。

如果该结点为最后一层的非终端结点，有下列 3 种可能：

- 被删关键字所在结点中的关键字数目不小于$\lceil m/2 \rceil$，则只需从该结点删除该关键字 K_i 以及相应的指针 A_i 。

  > 例如，在图 3 中，删除关键字 12 ，只需要删除该关键字 12 以及右侧指向 NULL 指针即可。

- 被删关键字所在结点中的关键字数目等于$\lceil m/2 \rceil -1$，而与该结点相邻的右兄弟结点（或者左兄弟）结点中的关键字数目大于$\lceil m/2 \rceil -1$，只需将该兄弟结点中的最小（或者最大）的关键字上移到双亲结点中，然后将双亲结点中小于（或者大于）且紧靠该上移关键字的关键字移动到被删关键字所在的结点中。

例如在图 3 中删除关键字 50，其右兄弟结点 g 中的关键字大于 2，所以需要将结点 g 中最小的关键字 61 上移到其双亲结点 e 中（由此 e 中结点有：53，61，90），然后将小于 61 且紧靠 61 的关键字 53 下移到结点 f 中，最终删除后的 B-树如图 10 所示。

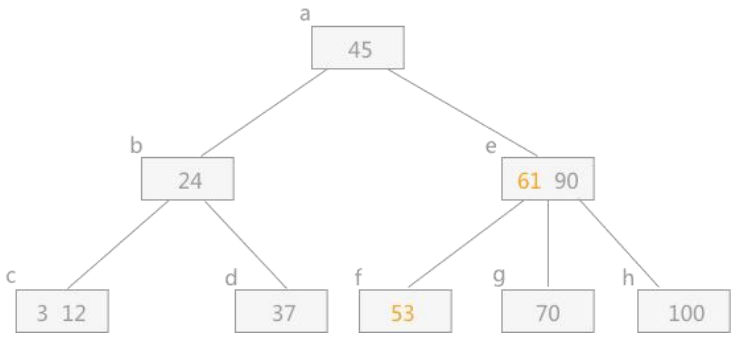


图 10 删除结点 50 后的 B-树

- 被删除关键字所在的结点如果和其相邻的兄弟结点中的关键字数目都正好等于⌈ m/2⌉ -1，假设其有右兄弟结点，且其右兄弟结点是由双亲结点中的指针 A_i 所指，则需要在删除该关键字的同时，将剩余的关键字和指针连同双亲结点中的 K_i 一起合并到右兄弟结点中。

  例如，在图 10 中 B-树中删除关键字 53，由于其有右兄弟，且右兄弟结点中只有 1 个关键字。在删除关键字 53 后，结点 f 中只剩指向叶子结点的空指针，连同双亲结点中的 61（因为 61 右侧指针指向的兄弟结点 g）一同合并到结点 g 中，最终删除 53 后的 B-树为：

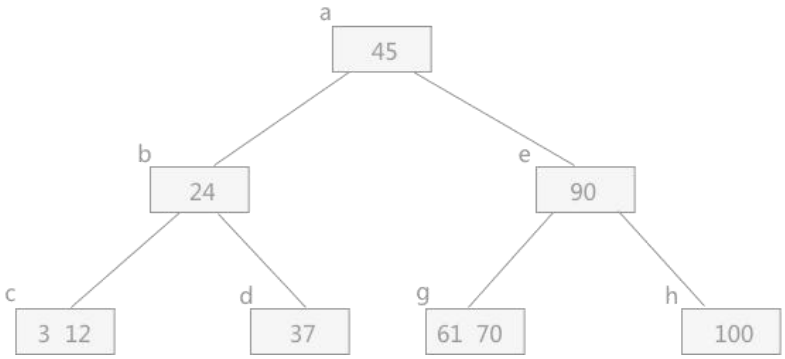


图 11 删除结点 53 后的 B-树

在合并的同时，由于从双亲结点中删除一个关键字，若导致双亲结点中关键字数目小于⌈ m/2⌉ -1，则继续按照该规律进行合并。例如在图 11 中 B-树的情况下删除关键字 12 时，结点 c 中只有一个关键字，然后做删除关键字 37 的操作。此时在删除关键字 37 的同时，结点 d 中的剩余信息（空指针）同双亲结点中的关键字 24 一同合并到结点 c 中，效果图为：

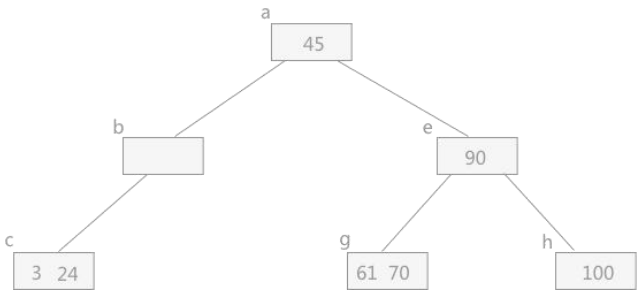


图 12 删除结点 37 后的效果图

由于结点 b 中一个关键字也没有，所以破坏了 B-树的结构，继续整合。在删除结点 b 的同时，由于 b 中仅剩指向结点 c 的指针，所以连同其双亲结点中的 45 一同合并到其兄弟结点 e 中，最终的 B-树为：

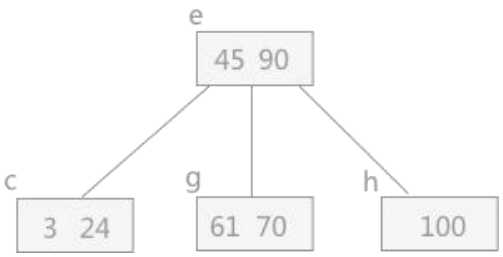


图 13 删除 37 后的 B-树

总结

由于 B-树具有分支多层数少的特点，使得它更多的是应用在数据库系统中。除了 B-树，还有专门为文件系统而生的 B+树，在本章的下一节会详细介绍。

B+树及插入和删除操作详解

本节介绍一种应文件系统所需而生的一种 B-树的变型树——B+树。前面介绍了 B-树，B+树其实同 B-树有许多相同之处，本节将用 B-树同 B+树通过对比两者的差异来介绍 B+树。

什么是 B+树？

一颗 m 阶的 B+树和 m 阶的 B-树的差异在于：

- 有 n 棵子树的结点中含有 n 个关键字；

在上一节中，在 B-树中的每个结点关键字个数 n 的取值范围为⌈m/2⌉ -1≤n≤m-1，而在 B+树中每个结点中关键字个数 n 的取值范围为：⌈m/2⌉ ≤n≤m。

- 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
- 所有的非终端结点（非叶子结点）可以看成是索引部分，结点中仅含有其子树（根结点）中的最大（或最小）关键字。

例如，图 1 中所示的就是一棵深度为 4 的 3 阶 B+树：

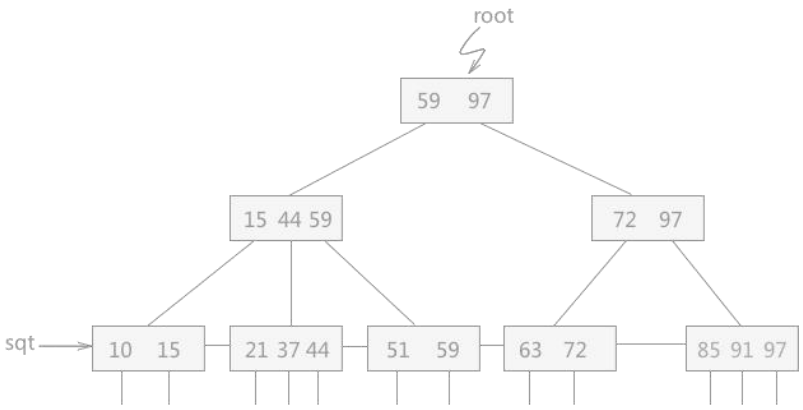


图 1 3 阶 B+树

如图 1 所示，B+树中含有两个头指针，一个指向整棵树的根结点，另一个指向关键字最小的叶子结点。同时所有的叶子结点依据其关键字的大小自小而大顺序链接，所有的叶子结点构成了一个 sqt 指针为头指针的链表。

所有，B+树可以进行两种查找运算：一种是利用 sqt 链表做顺序查找，另一种是从树的根结点开始，进行类似于二分查找的查找方式。

在 B+树中，所有非终端结点都相当于是终端结点的索引，而所有的关键字都存放在终端结点中，所有在从根结点出发做查找操作时，如果非终端结点上的关键字恰好等于给定值，此时并不算查找完成，而是要继续向下直到叶子结点。

B+树的查找操作，无论查找成功与否，每次查找操作都是走了一条从根结点到叶子结点的路径。

B+树中插入关键字

在 B+树中插入关键字时，需要注意以下几点：

- 插入的操作全部都在叶子结点上进行，且不能破坏关键字自小而大的顺序；
- 由于 B+树中各结点中存储的关键字的个数有明确的范围，做插入操作可能会出现结点中关键字个数超过阶数的情况，此时需要将该结点进行“分裂”；

B+树中做插入关键字的操作，有以下 3 种情况：

1、 若被插入关键字所在的结点，其含有关键字数目小于阶数 M，则直接插入结束；

例如，在图 1 中插入关键字 13，其结果如图 2 所示：

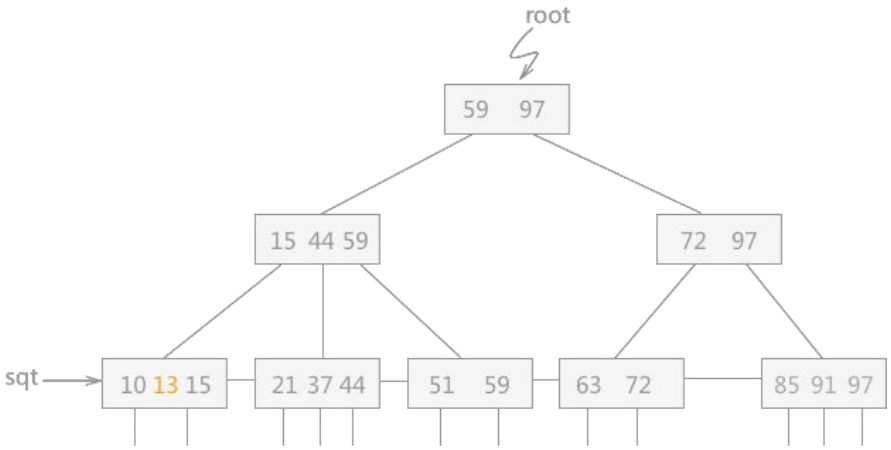


图 2 插入 13 后的 B+树

2、 若被插入关键字所在的结点，其含有关键字数目等于阶数 M，则需要将该结点分裂为两个结点，一个结点包含⌊M/2⌋，另一个结点包含⌈M/2⌉。同时，将⌈M/2⌉的关键字上移至其双亲结点。假设其双亲结点中包含的关键字个数小于 M，则插入操作完成。

例如，在图 1 的基础上插入关键字 95，其插入后的 B+树如图 3 所示：

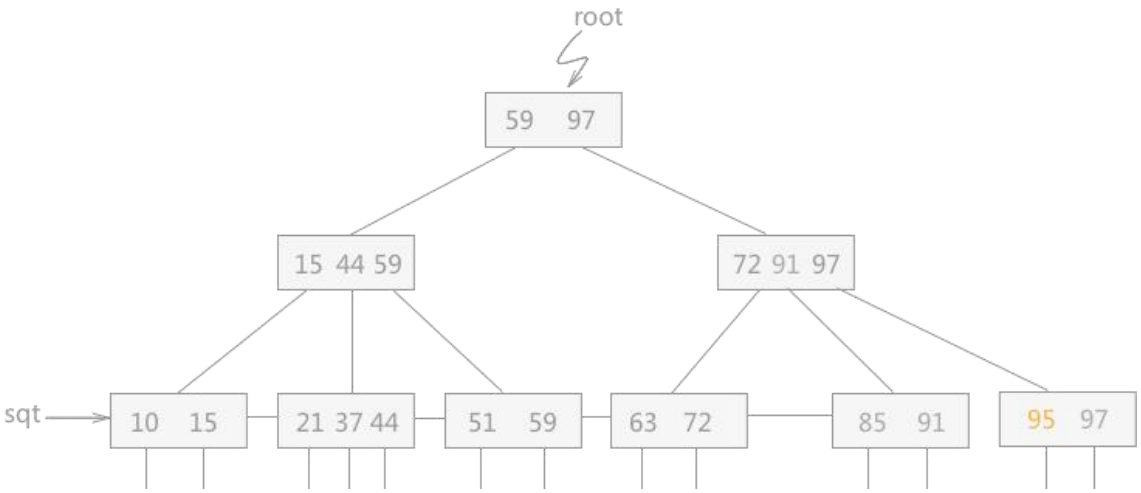


图 3 插入 95 后的 B+树

3、在第 2 情况中，如果上移操作导致其双亲结点中关键字个数大于 M，则应继续分裂其双亲结点。

例如，在图 1 的 B+树中插入关键字 40，则插入后的 B+树如图 4 所示：

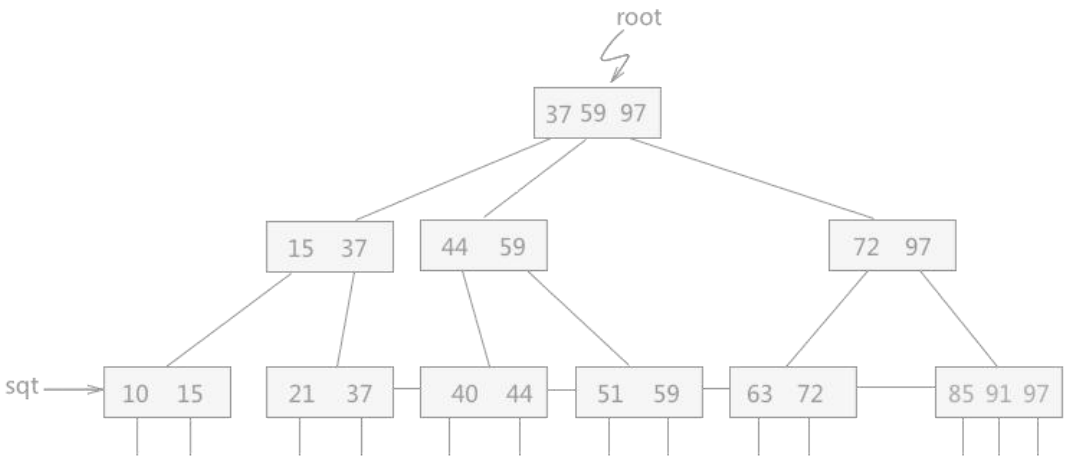


图 4 插入 40 后的 B+树

注意：如果插入的关键字比当前结点中的最大值还大，破坏了 B+树中从根结点到当前结点的所有索引值，此时需要及时修正后，再做其他操作。例如，在图 1 的 B+树种插入关键字 100，由于其值比 97 还大，插入之后，从根结点到该结点经过的所有结点中的所有值都要由 97 改为 100。改完之后再做分裂操作。

B+树中删除关键字

在 B+树中删除关键字时，有以下几种情况：

1、 找到存储有该关键字所在的结点时，由于该结点中关键字个数大于⌈ M/2⌉，做删除操作不会破坏 B+树，则可以直接删除。

例如，在图 1 所示的 B+树中删除关键字 91，删除后的 B+树如图 5 所示：

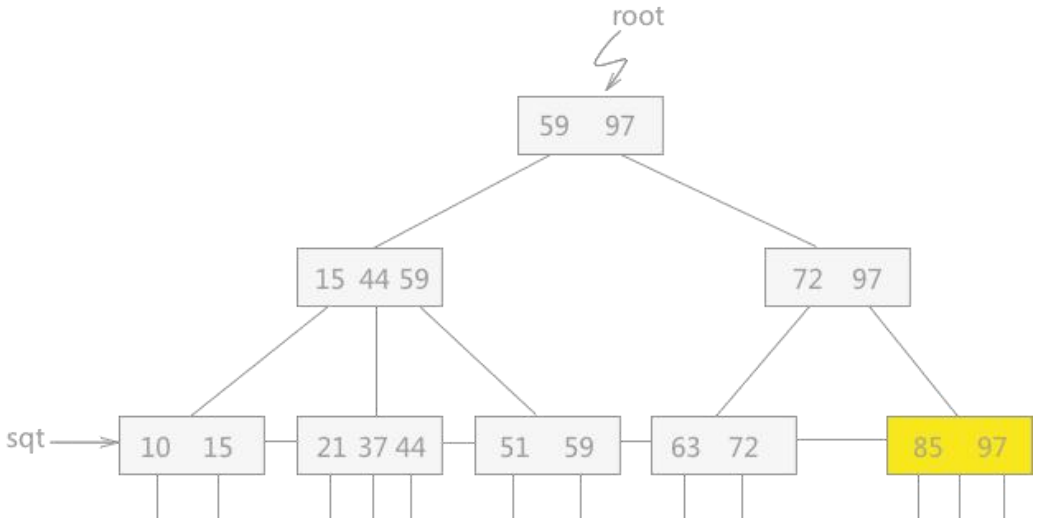


图 5 删除 91 的 B+树

2、 当删除某结点中最大或者最小的关键字，就会涉及到更改其双亲结点一直到根结点中所有索引值的更改。

例如，在图 1 的 B+树中删除关键字 97，删除后的 B+树如图 6 所示：

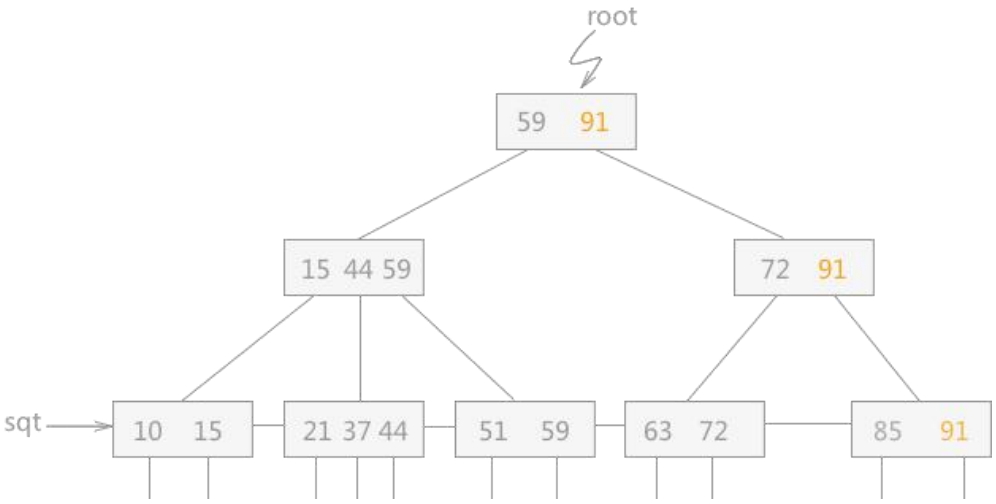


图 6 删除 97 后的 B+树

3、 当删除该关键字，导致当前结点中关键字个数小于⌈ M/2⌉，若其兄弟结点中含有多余的关键字，可以从兄弟结点中借关键字完成删除操作。

例如，在图 1 的 B+树中删除关键字 51，由于其兄弟结点中含有 3 个关键字，所以可以选择借一个关键字，同时修改双亲结点中的索引值，删除之后的 B+树如图 7 所示：

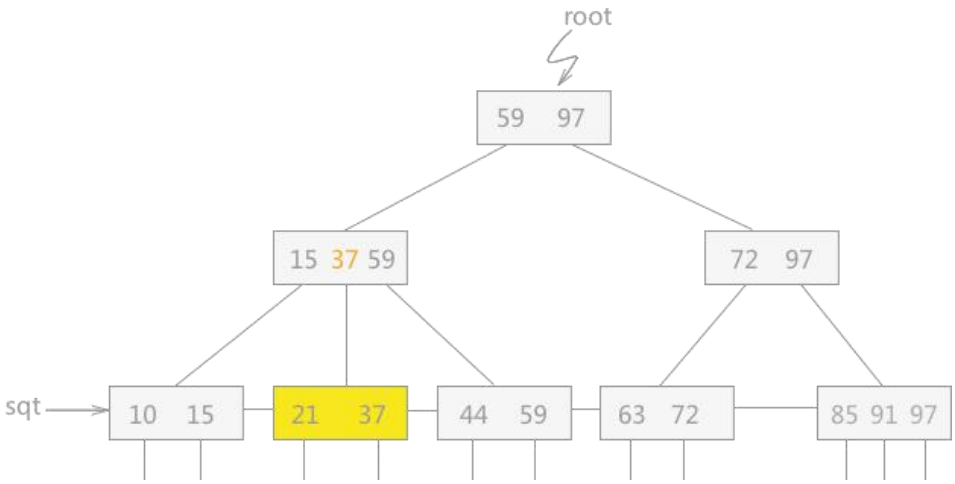


图 7 删除关键字 51 后的 B+树

4、 第 3 种情况中，如果其兄弟结点没有多余的关键字，则需要同其兄弟结点进行合并。

例如，在图 7 的 B+树种删除关键字 59，删除后的 B+树为:

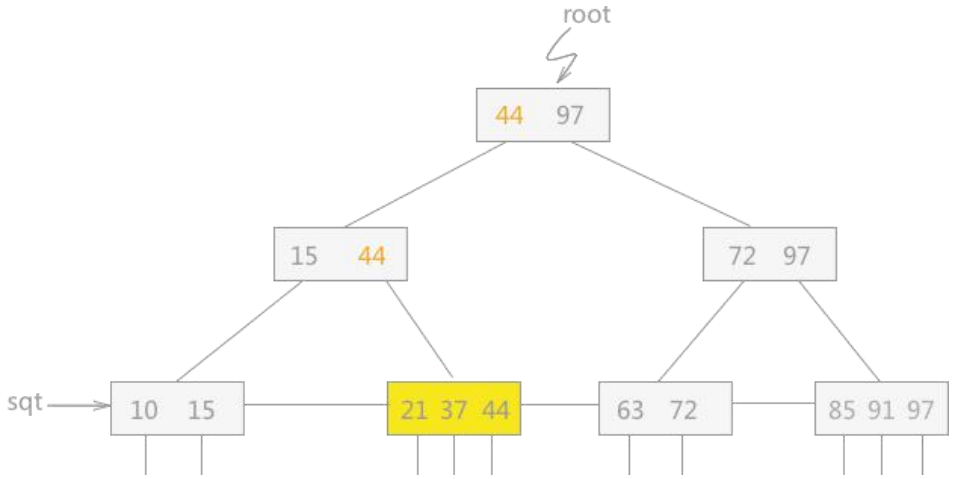


图 8 删除关键字 59 后的 B+树

5、 当进行合并时，可能会产生因合并使其双亲结点破坏 B+树的结构，需要依照以上规律处理其双亲结点。

例如，在图 6 的 B+树中删除关键字 63，当删除后该结点中只剩关键字 72，且其兄弟结点中只有 2 个关键字，无法实现借的操作，只能进行合并。但是合并后，合并后的效果图如图 9 所示:

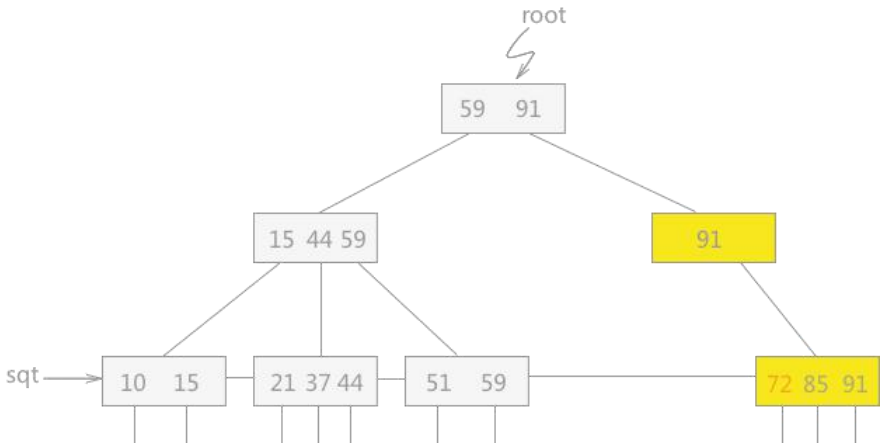


图 9 合并操作后的效果图

如图 9 所示，其双亲结点中只有一个关键字，而其兄弟结点中有 3 个关键字，所以可以通过借的操作，来满足 B+树的性质，最终的 B+树如图 10 所示：

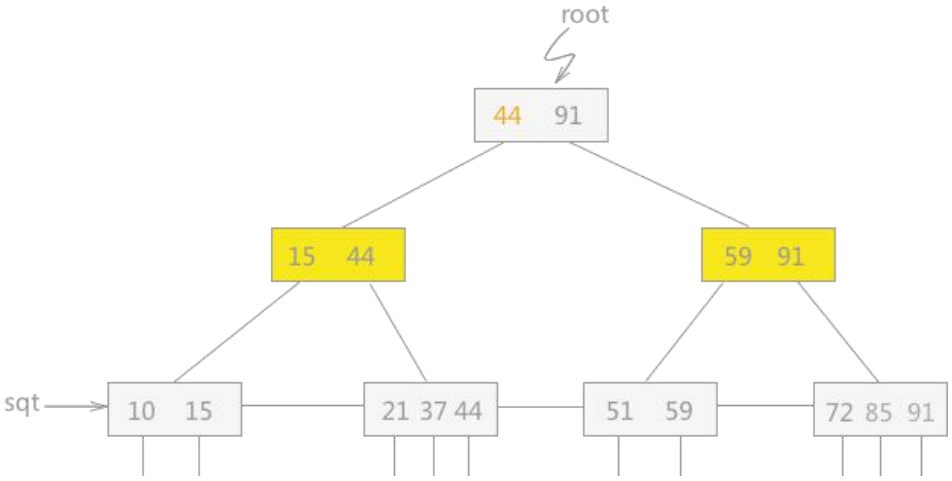


图 10 删除 63 后的 B+树

总之，在 B+树中做删除关键字的操作，采取如下的步骤：

1. 删除该关键字，如果不破坏 B+树本身的性质，直接完成操作；
2. 如果删除操作导致其该结点中最大（或最小）值改变，则应相应改动其父结点中的索引值；
3. 在删除关键字后，如果导致其结点中关键字个数不足，有两种方法：一种是向兄弟结点去借，另外一种是同兄弟结点合并。（注意这两种方式有时需要更改其父结点中的索引值。）

总结

本节介绍了有关 B+树的查找、插入和删除操作，由于其更多的是用于文件索引系统，所以没有介绍具体地代码实现，只需要了解实现过程即可。

键树查找法（双链树和字典树）及 C 语言实现

键树，又称为数字查找树（根结点的子树个数 >= 2），同以往所学习的树不同的是，键树的结点中存储的不是某个关键字，而是只含有组成关键字的单个符号。

如果关键字本身是字符串，则键树中的一个结点只包含有一个字符；如果关键字本身是数字，则键树中的一个结点只包含一个数位。每个关键字都是从键树的根结点到叶子结点中经过的所有结点中存储的组合。

例如，当使用键树表示查找表{CAI，CAO，CHEN，LI，LAN，ZHAO}时，为了查找方便，首先对该查找表中关键字按照首字符进行分类（相同的在一起）：

```
{{CAI,CAO,CHEN},{LI,LAN},{ZHAO}}
```

然后继续分割，按照第二个字符、第三个字符、...，最终得到的查找表为：

```
{{CAI,CAO},{CHEN},{LI,LAN},{ZHAO}}
```

然后使用键树结构表示该查找表，如图 1 所示：

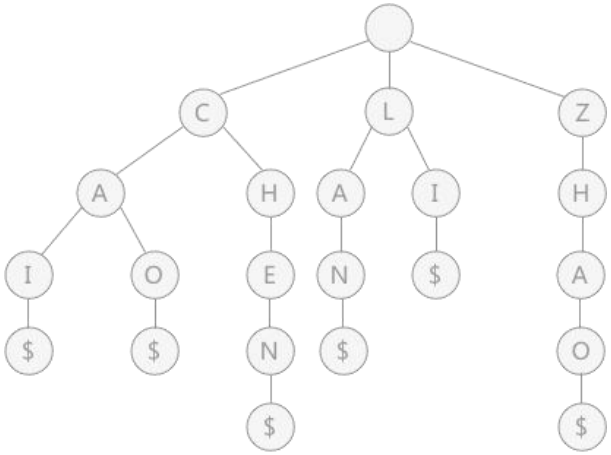


图 1 键树表示多个关键字

注意：键树中叶子结点的特殊符号 $ 为结束符，表示字符串的结束。使用键树表示查找表时，为了方便后期的查找和插入操作，约定键树是有序树（兄弟结点之间自左至右有序），同时约定结束符 ‘$’ 小于任何字符。

键树的存储结构

键树的存储结构有两种：一种是通过使用树的孩子兄弟表示法来表示键树，即双链树；另一种是以树的多重链表表示键树，即 Trie 树，又称字典树。

双链树

当使用孩子兄弟表示法表示键树时，树的结点构成分为 3 部分：

- symbol 域：存储关键字的一个字符；
- first 域：存储指向孩子结点的指针；
- next 域：存储指向兄弟结点的指针；

注意：对于叶子结点来说，由于其没有孩子结点，在构建叶子结点时，将 first 指针换成 infoptr 指针，用于指向该关键字。当叶子结点（结束符 ‘$’ 所在的结点）中使用 infoptr 域指向该自身的关键字时，此时的键树被称为双链树。

如图 1 中的键树用孩子兄弟表示法表示为双链树时，如图 2 所示：

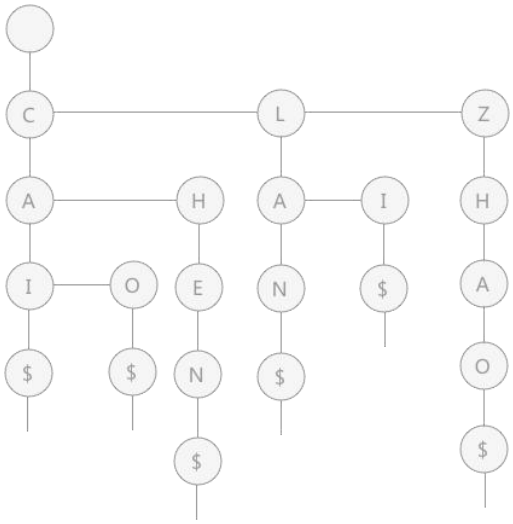


图 2 双链树

提示：每个关键字的叶子结点 $ 的 infoptr 指针指向的是各自的关键字，通过该指针就可以找到各自的关键字的首地址。

双链树查找功能的具体实现

在使用孩子兄弟表示法表示的键树中做查找操作，从树的根结点出发，依次同被查找的关键字进行比对，如果比对成功，进行下一字符的比对；反之，如果比对失败，则跳转至该结点的兄弟结点中去继续比对，直至比对成功或者为找到该关键字。

具体实现的代码：

```
#include <stdio.h>
typedef enum{LEFT,BRANCH}NodeKind;//定义结点的类型，是叶子结点还是其他类型的结点
typedef   struct {
    char a[20];//存储关键字的数组
    int num;//关键字长度
}KeysType;
//定时结点结构
typedef struct DLTNode{
    char symbol;//结点中存储的数据
    struct DLTNode *next;//指向兄弟结点的指针
    NodeKind *kind;//结点类型
    union{//其中两种指针类型每个结点二选一
        struct DLTNode* first;//孩子结点
        struct DLTNode* infoptr;//叶子结点特有的指针
    };
}*DLTree;
//查找函数，如果查找成功，返回该关键字的首地址，反则返回 NULL。T 为用孩子兄弟表示法表示的键树，K 为被查找的关键字。
DLTree SearchChar(DLTree T, KeysType k){
```

```
    int i = 0;
    DLTree p = T->first;//首先令指针 P 指向根结点下的含有数据的孩子结点
    //如果 p 指针存在，且关键字中比对的位数小于总位数时，就继续比对
    while (p && i < k.num){
        //如果比对成功，开始下一位的比对
        if (k.a[i] == p->symbol){
            i++;
            p = p->first;
        }
        //如果该位比对失败，则找该结点的兄弟结点继续比对
        else{
            p = p->next;
        }
    }
    //比对完成后，如果比对成功，最终 p 指针会指向该关键字的叶子结点 $，通过其自有的 infoptr 指针找到该关键字。
    if ( i == k.num){
        return p->infoptr;
    }
    else{
        return NULL;
    }
}
```

Trie 树（字典树）

若以树的多重链表表示键树，则树中如同双链树一样，会含有两种结点：

1. 叶子结点：叶子结点中含有关键字域和指向该关键字的指针域；
2. 除叶子结点之外的结点（分支结点）：含有 d 个指针域和一个整数域（记录该结点中指针域的个数）；

> d 表示每个结点中存储的关键字的所有可能情况，如果存储的关键字为数字，则 d= 11（0—9，以及 $），同理，如果存储的关键字为字母，则 d=27（26 个字母加上结束符 $）。

图 1 中的键树，采用字典树表示如图 3 所示：

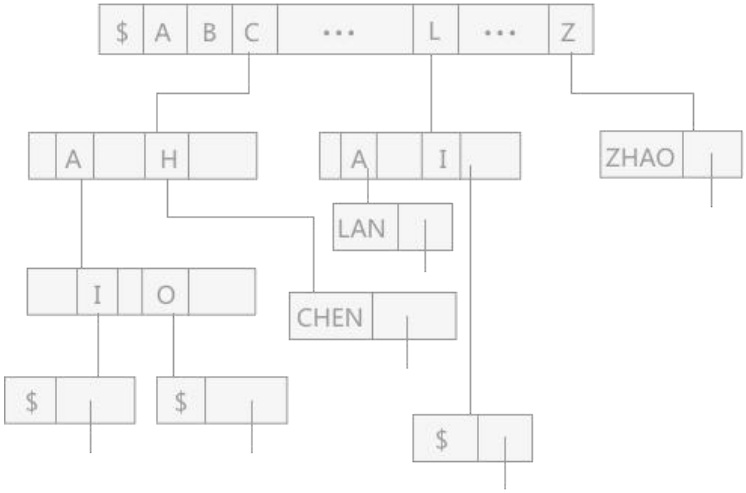


图 3 Trie 树

注意：在 Trie 树中，如果从某个结点一直到叶子结点都只有一个孩子，这些结点可以用一个叶子结点来代替，例如 ZHAO 就可以直接作为叶子结点。

字典树查找功能的具体实现

使用 Trie 树进行查找时，从根结点出发，沿和对应关键字中的值相对应的指针逐层向下走，一直到叶子结点，如果全部对应相等，则查找成功；反之，则查找失败。

具体实现代码为：

```
typedef enum{LEFT,BRANCH}NodeKind;//定义结点类型
typedef struct {//定义存储关键字的数组
    char a[20];
    int num;
}KeysType;
//定义结点结构
typedef struct TrieNode{
    NodeKind kind;//结点类型
    union{
        struct { KeysType k; struct TrieNode *infoptr; }lf;//叶子结点
        struct{ struct TrieNode *ptr[27]; int num; }bh;//分支结点
    };
}*TrieTree;
//求字符 a 在字母表中的位置
int ord(char  a){
    int b = a - 'A'+1;
    return b;
}
```

```
//查找函数
TrieTree SearchTrie(TrieTree T, KeysType K){
    int i=0;
    TrieTree p = T;
    while (i < K.num){
        if (p && p->kind==BRANCH && p->bh.ptr[ord(K.a[i])]){
            i++;
            p = p->bh.ptr[ord(K.a[i])];
        }
        else{
            break;
        }
    }
    if (p){
        return p->lf.infoptr;
    }
    return p;
}
```

使用 Trie 树进行查找的过程实际上是走了一条从根结点到叶子结点的路径，所以使用 Trie 进行的查找效率取决于该树的深度。

总结

双链树和字典树是键树的两种表示方法，各有各的特点，具体使用哪种方式表示键树，需要根据实际情况而定。例如，若键树中结点的孩子结点较多，则使用字典树较双链树更为合适。

哈希查找算法及 C 语言实现

上一节介绍了有关哈希表及其构造过程的相关知识，本节将介绍如何利用哈希表实现查找操作。

在哈希表中进行查找的操作同哈希表的构建过程类似，其具体实现思路为：对于给定的关键字 K，将其带入哈希函数中，求得与该关键字对应的数据的哈希地址，如果该地址中没有数据，则证明该查找表中没有存储该数据，查找失败：如果哈希地址中有数据，就需要做进一步的证明（排除冲突的影响），找到该数据对应的关键字同 K 进行比对，如果相等，则查找成功；反之，如果不相等，说明在构造哈希表时发生了冲突，需要根据构造表时设定的处理冲突的方法找到下一个地址，同地址中的数据进行比对，直至遇到地址中数据为 NULL（说明查找失败），或者比对成功。

回顾：哈希表在构造过程中，处理冲突的方法有：开放定址法、再哈希法、链地址法、建立公共溢出区法。

假设哈希表在构造过程采用的开放定址法处理的冲突，则哈希表的查找过程用代码实现为：

```
#include "stdio.h"
#include "stdlib.h"
#define HASHSIZE 7 //定义散列表长为数组的长度
```

```
#define NULLKEY -1
typedef struct{
    int *elem;//数据元素存储地址，动态分配数组
    int count; //当前数据元素个数
}HashTable;
//对哈希表进行初始化
void Init(HashTable *hashTable){
    int i;
    hashTable->elem= (int *)malloc(HASHSIZE*sizeof(int));
    hashTable->count=HASHSIZE;
    for (i=0;i<HASHSIZE;i++){
        hashTable->elem[i]=NULLKEY;
    }
}
//哈希函数(除留余数法)
int Hash(int data){
    return data%HASHSIZE;
}
//哈希表的插入函数，可用于构造哈希表
void Insert(HashTable *hashTable,int data){
    int hashAddress=Hash(data); //求哈希地址
    //发生冲突
    while(hashTable->elem[hashAddress]!=NULLKEY){
        //利用开放定址法解决冲突
        hashAddress=(++hashAddress)%HASHSIZE;
    }
    hashTable->elem[hashAddress]=data;
}

//哈希表的查找算法
int Search(HashTable *hashTable,int data){
    int hashAddress=Hash(data); //求哈希地址
    while(hashTable->elem[hashAddress]!=data){//发生冲突
        //利用开放定址法解决冲突
        hashAddress=(++hashAddress)%HASHSIZE;
        //如果查找到的地址中数据为 NULL，或者经过一圈的遍历回到原位置，则查找失败
        if (hashTable->elem[hashAddress]==NULLKEY||hashAddress==Hash(data)){
```

```
                return -1;
            }
        }
        return hashAddress;
}
int main(){
    int i,result;
    HashTable hashTable;
    int arr[HASHSIZE]={13,29,27,28,26,30,38};
    //初始化哈希表
    Init(&hashTable);
    //利用插入函数构造哈希表
    for (i=0;i<HASHSIZE;i++){
        Insert(&hashTable,arr[i]);
    }
    //调用查找算法
    result= Search(&hashTable,29);
    if (result==-1) printf("查找失败");
    else printf("29 在哈希表中的位置是:%d",result+1);
    return  0;
}
```

运行结果为：

```
29 在哈希表中的位置是:2
```

查找算法的效率分析

在构造哈希表的过程中，由于冲突的产生，使得哈希表的查找算法仍然会涉及到比较的过程，因此对于哈希表的查找效率仍需以平均查找长度来衡量。

在哈希表的查找过程中需和给定值进行比较的关键字的个数取决于以下 3 个因素：

- 哈希函数：哈希函数的“好坏”取决于影响出现冲突的频繁程度。但是一般情况下，哈希函数相比于后两种的影响，可以忽略不计。
- 处理冲突的方式：对于同一组关键字，设定相同的哈希函数，使用不同的处理冲突的方式得到的哈希表是不同的，表的平均查找长度也不同。
- 哈希表的装填因子：在一般情况下，当处理冲突的方式相同的情况下，其平均查找长度取决于哈希表的装满程度：装的越满，插入数据时越有可能发生冲突；反之则越小。

装填因子=哈希表中数据的个数/哈希表的长度，用字符 α 表示（是数学符号，而不是字符 a）。装填因子越小，表示哈希表中空闲的位置就越多。

经过计算，在假设查找表中的所有数据的查找概率相等的情况下，对于表长为 m，数据个数为 n 的哈希表：

- 其查找成功的平均查找长度约为：-1/α * ln(1-α)
- 其查找不成功的平均查找长度约为：1/(1-α)

通过公式可以看到，哈希表的查找效率只同装填因子有关，而同哈希表中的数据的个数无关，所以在选用哈希表做查找操作时，选择一个合适的装填因子是非常有必要的。

希尔排序算法（缩小增量排序）及 C 语言实现

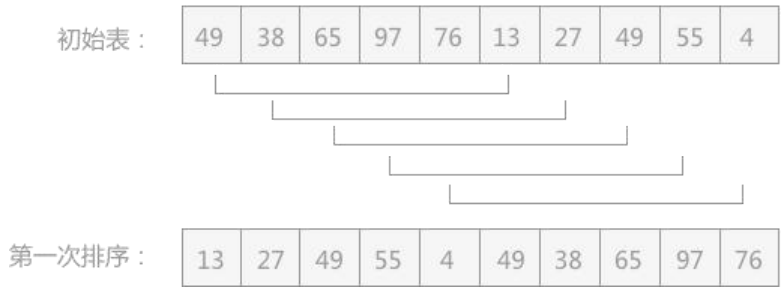
希尔排序，又称“缩小增量排序”，也是插入排序的一种，但是同前面几种排序算法比较来看，希尔排序在时间效率上有很大的改进。

在使用直接插入排序算法时，如果表中的记录只有个别的是无序的，多数保持有序，这种情况下算法的效率也会比较高；除此之外，如果需要排序的记录总量很少，该算法的效率同样会很高。希尔排序就是从这两点出发对算法进行改进得到的排序算法。

希尔排序的具体实现思路是：先将整个记录表分割成若干部分，分别进行直接插入排序，然后再对整个记录表进行一次直接插入排序。

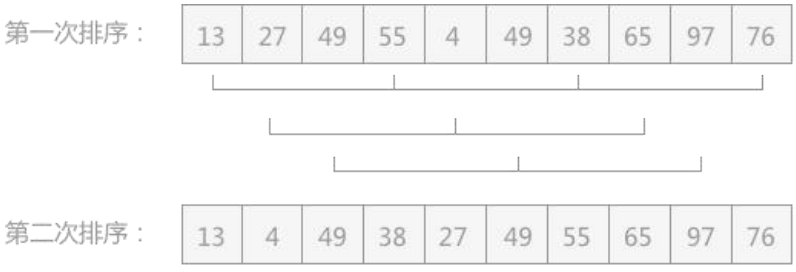
例如无序表{49，38，65，97，76，13，27，49，55，4}进行希尔排序的过程为：

- 首先对 {49，13}，{38，27}，{65，49}，{97，55}，{76，4} 分别进行直接插入排序（如果需要调换位置也只是互换存储位置），如下图所示：

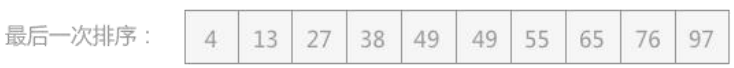


上图中两两进行比较，例如 49 和 13 进行比较，13<49，所以交换存储位置。

- 通过一次排序，无序表中的记录已基本有序，此时还可以再进行一次分割，如下图所示：



- 经过两次分割，无序表中已基本有序，此时对整张表进行一次直接插入排序（只需要做少量的比较和插入操作即可），最终希尔排序的结果为：



希尔排序的过程中，对于分割的每个子表，其各自包含的记录在原表中并不是相互挨着的，而是相互之间相隔着某个固定的常数。例如上例中第一次排序时子表中的记录分割的常量为 5，第二次排序时为 3。

通过此种方式，对于关键字的值较小的记录，其前移的过程不是一步一步的，而是跳跃性的前移，并且在最后一次对整表进行插入排序时减少了比较和排序的次数。

一般在记录的数量多的情况下，希尔排序的排序效率较直接插入排序高。

希尔排序的具体代码实现：

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 15
typedef struct {
    int key;    //关键字的值
}SLNode;
typedef struct {
    SLNode r[SIZE];//存储记录的数组
    int length;//记录数组中记录的数量
}SqList;
//希尔排序的实现函数，其中 dk 表示增值量
void ShellInsert(SqList * L,int dk){
    //从 dk+1 个记录起，每间隔 dk 个记录就调取一个进行直接插入排序算法
    for (int i=dk+1; i<=L->length; i++) {
        //如果新调取的关键字的值，比子表中最后一个记录的关键字小，说明需要将该值调换位置
        if (L->r[i].key<L->r[i-dk].key) {
            int j;
            //记录表中，使用位置下标为 0 的空间存储需要调换位置的记录的值
            L->r[0]=L->r[i];
            //做直接插入排序，如果下标为 0 的关键字比下标为 j 的关键字小，则向后一行下标为 j 的值，为新插入的记录腾出空间。
            for (j=i-dk; j>0 && (L->r[0].key<L->r[j].key); j-=dk){
                L->r[j+dk]=L->r[j];
            }
            //跳出循环后，将新的记录值插入到腾出的空间中，即完成了一次直接插入排序
            L->r[j+dk]=L->r[0];
        }
    }
}
//希尔排序，通过调用不同的增量值（记录），实现对多个子表分别进行直接插入排序
void ShellSort(SqList * L,int dlta[],int t){
```

```
    for (int k=0; k<t; k++) {
        ShellInsert(L, dlta[k]);
    }
}
int main(int argc, const char * argv[]) {
    int dlta[3]={5,3,1};//用数组来存储增量值，例如 5 代表每间隔 5 个记录组成一个子表，1 表示每间隔一个，也就是最后一次对整张表的直接插入排序
    SqList *L=(SqList*)malloc(sizeof(SqList));
    L->r[1].key=49;
    L->r[2].key=38;
    L->r[3].key=64;
    L->r[4].key=97;
    L->r[5].key=76;
    L->r[6].key=13;
    L->r[7].key=27;
    L->r[8].key=49;
    L->r[9].key=55;
    L->r[10].key=4;
    L->length=10;
    //调用希尔排序算法
    ShellSort(L, dlta, 3);
    //输出语句
    for (int i=1; i<=10; i++) {
        printf("%d ",L->r[i].key);
    }
    return 0;
}
```

运行结果：

```
4 13 27 38 49 49 55 64 76 97
```

提示：经过大量的研究表明，所选取的增量值最好是没有除 1 之外的公因子，同时整个增量数组中最后一个增量值必须等于 1 ，因为最后必须对整张表做一次直接插入排序算法。

快速排序算法（QSort，快排）及 C 语言实现

上节介绍了如何使用起泡排序的思想对无序表中的记录按照一定的规则进行排序，本节再介绍一种排序算法——快速排序算法（Quick Sort）。

C 语言中自带函数库中就有快速排序——qsort 函数 ，包含在 <stdlib.h> 头文件中。

快速排序算法是在起泡排序的基础上进行改进的一种算法，其实现的基本思想是：通过一次排序将整个无序表分成相互独立的两部分，其中一部分中的数据都比另一部分中包含的数据的值小，然后继续沿用此方法分别对两部分进行同样的操作，直到每一个小部分不可再分，所得到的整个序列就成为了有序序列。

例如，对无序表{49，38，65，97，76，13，27，49}进行快速排序，大致过程为：

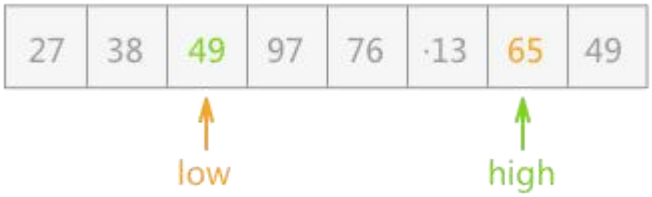
1. 首先从表中选取一个记录的关键字作为分割点（称为“枢轴”或者支点，一般选择第一个关键字），例如选取 49；
2. 将表格中大于 49 个放置于 49 的右侧，小于 49 的放置于 49 的左侧，假设完成后的无序表为：{27，38，13，49，65，97，76，49}；
3. 以 49 为支点，将整个无序表分割成了两个部分，分别为{27，38，13}和{65，97，76，49}，继续采用此种方法分别对两个子表进行排序；
4. 前部分子表以 27 为支点，排序后的子表为{13，27，38}，此部分已经有序；后部分子表以 65 为支点，排序后的子表为{49，65，97，76}；
5. 此时前半部分子表中的数据已完成排序；后部分子表继续以 65 为支点，将其分割为{49}和{97，76}，前者不需排序，后者排序后的结果为{76，97}；
6. 通过以上几步的排序，最后由子表{13，27，38}、{49}、{49}、{65}、{76，97}构成有序表：{13，27，38，49，49，65，76，97}；

整个过程中最重要的是实现第 2 步的分割操作，具体实现过程为：

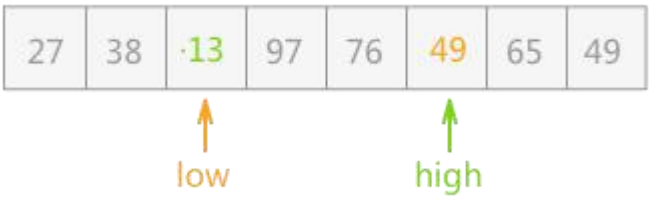
- 设置两个指针 low 和 high，分别指向无序表的表头和表尾，如下图所示：

- 先由 high 指针从右往左依次遍历，直到找到一个比 49 小的关键字，所以 high 指针走到 27 的地方停止。找到之后将该关键字同 low 指向的关键字进行互换：

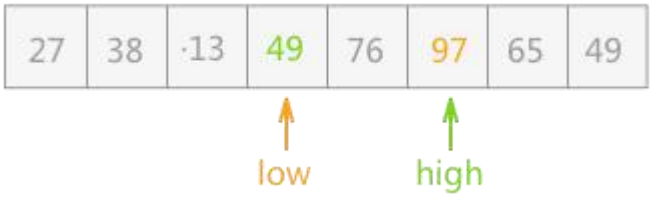
- 然后指针 low 从左往右依次遍历，直到找到一个比 49 大的关键字为止，所以 low 指针走到 65 的地方停止。同样找到后同 high 指向的关键字进行互换：



- 指针 high 继续左移，到 13 所在的位置停止（13<49），然后同 low 指向的关键字进行互换：



- 指针 low 继续右移，到 97 所在的位置停止（97>49），然后同 high 指向的关键字互换位置：



- 指针 high 继续左移，此时两指针相遇，整个过程结束；

该操作过程的具体实现代码为：

```
#define MAX 8
typedef struct {
    int key;
}SqNote;

typedef struct {
    SqNote r[MAX];
    int length;
}SqList;
//交换两个记录的位置
void swap(SqNote *a,SqNote *b){
    int key=a->key;
    a->key=b->key;
    b->key=key;
}
//快速排序，分割的过程
int Partition(SqList *L,int low,int high){
    int pivotkey=L->r[low].key;
    //直到两指针相遇，程序结束
    while (low<high) {
        //high 指针左移，直至遇到比 pivotkey 值小的记录，指针停止移动
        while (low<high && L->r[high].key>=pivotkey) {
            high--;
        }
```

```
        //交换两指针指向的记录
        swap(&(L->r[low]), &(L->r[high]));
        //low  指针右移，直至遇到比 pivotkey 值大的记录，指针停止移动
        while (low<high && L->r[low].key<=pivotkey) {
            low++;
        }
        //交换两指针指向的记录
        swap(&(L->r[low]), &(L->r[high]));
    }
    return low;
}
```

该方法其实还有可以改进的地方：在上边实现分割的过程中，每次交换都将支点记录的值进行移动，而实际上只需在整个过程结束后（low==high），两指针指向的位置就是支点记录的准确位置，所以无需每次都移动支点的位置，最后移动至正确的位置即可。

所以上边的算法还可以改写为：

```
//此方法中，存储记录的数组中，下标为 0 的位置时空着的，不放任何记录，记录从下标为 1 处开始依次存放
int Partition(SqList *L,int low,int high){
    L->r[0]=L->r[low];
    int pivotkey=L->r[low].key;
    //直到两指针相遇，程序结束
    while (low<high) {
        //high 指针左移，直至遇到比 pivotkey 值小的记录，指针停止移动
        while (low<high && L->r[high].key>=pivotkey) {
            high--;
        }
        //直接将 high 指向的小于支点的记录移动到 low 指针的位置。
        L->r[low]=L->r[high];
        //low  指针右移，直至遇到比 pivotkey 值大的记录，指针停止移动
        while (low<high && L->r[low].key<=pivotkey) {
            low++;
        }
        //直接将 low 指向的大于支点的记录移动到 high 指针的位置
        L->r[high]=L->r[low];
    }
    //将支点添加到准确的位置
    L->r[low]=L->r[0];
```

```
    return low;
}
```

快速排序的完整实现代码（C 语言）

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 9
//单个记录的结构体
typedef struct {
    int key;
}SqNote;
//记录表的结构体
typedef struct {
    SqNote r[MAX];
    int length;
}SqList;
//此方法中，存储记录的数组中，下标为 0 的位置时空着的，不放任何记录，记录从下标为 1 处开始依次存放
int Partition(SqList *L,int low,int high){
    L->r[0]=L->r[low];
    int pivotkey=L->r[low].key;
    //直到两指针相遇，程序结束
    while (low<high) {
        //high 指针左移，直至遇到比 pivotkey 值小的记录，指针停止移动
        while (low<high && L->r[high].key>=pivotkey) {
            high--;
        }
        //直接将 high 指向的小于支点的记录移动到 low 指针的位置。
        L->r[low]=L->r[high];
        //low  指针右移，直至遇到比 pivotkey 值大的记录，指针停止移动
        while (low<high && L->r[low].key<=pivotkey) {
            low++;
        }
        //直接将 low 指向的大于支点的记录移动到 high 指针的位置
        L->r[high]=L->r[low];
    }
    //将支点添加到准确的位置
    L->r[low]=L->r[0];
```

```
    return low;
}
void QSort(SqList *L,int low,int high){
    if (low<high) {
        //找到支点的位置
        int pivotloc=Partition(L, low, high);
        //对支点左侧的子表进行排序
        QSort(L, low, pivotloc-1);
        //对支点右侧的子表进行排序
        QSort(L, pivotloc+1, high);
    }
}
void QuickSort(SqList *L){
    QSort(L, 1,L->length);
}
int main() {
    SqList * L=(SqList*)malloc(sizeof(SqList));
    L->length=8;
    L->r[1].key=49;
    L->r[2].key=38;
    L->r[3].key=65;
    L->r[4].key=97;
    L->r[5].key=76;
    L->r[6].key=13;
    L->r[7].key=27;
    L->r[8].key=49;
    QuickSort(L);
    for (int i=1; i<=L->length; i++) {
        printf("%d ",L->r[i].key);
    }
    return 0;
}
```

运行结果：

```
13 27 38 49 49 65 76 97
```

总结

快速排序算法的时间复杂度为 O(nlogn)，是所有时间复杂度相同的排序方法中性能最好的排序算法。

归并排序算法及其 C 语言具体实现

本节介绍一种不同于插入排序和选择排序的排序方法——归并排序，其排序的实现思想是先将所有的记录完全分开，然后两两合并，在合并的过程中将其排好序，最终能够得到一个完整的有序表。

例如对于含有 n 个记录的无序表，首先默认表中每个记录各为一个有序表（只不过表的长度都为 1），然后进行两两合并，使 n 个有序表变为 ⌈n/2⌉ 个长度为 2 或者 1 的有序表（例如 4 个小有序表合并为 2 个大的有序表），通过不断地进行两两合并，直到得到一个长度为 n 的有序表为止。这种归并排序方法称为：2-路归并排序。

例如对无序表{49，38，65，97，76，13，27}进行 2-路归并排序的过程如图 1 所示：

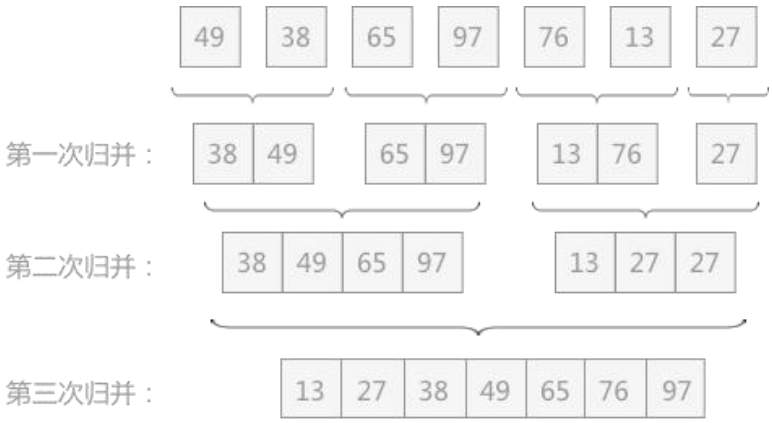


图 1 归并排序过程

归并过程中，每次得到的新的子表本身有序，所以最终得到的为有序表。

2-路归并排序的具体实现代码为（采用了递归的思想）：

```
#include <stdio.h>
#include <stdlib.h>
#define    MAX 8
typedef struct{
    int key;
}SqNode;
typedef struct{
    SqNode r[MAX];
    int length;
}SqList;
//SR 中的记录分成两部分：下标从 i 至 m 有序，从 m+1 至 n 也有序，此函数的功能是合二为一至 TR 数组中，使整个记录表有序
void Merge(SqNode SR[],SqNode TR[],int i,int m,int n){
    int j,k;
    //将 SR 数组中的两部分记录按照从小到大的顺序添加至 TR 数组中
```

```
    for (j=m+1,k=i; i<=m && j<=n; k++) {
        if (SR[i].key<SR[j].key) {
            TR[k]=SR[i++];
        }else{
            TR[k]=SR[j++];
        }
    }
    //将剩余的比目前 TR 数组中都大的记录复制到 TR 数组的最后位置
    while(i<=m) {
        TR[k++]=SR[i++];
    }
    while (j<=n) {
        TR[k++]=SR[j++];
    }
}

void MSort(SqNode SR[],SqNode TR1[],int s,int t){
    SqNode TR2[MAX];
    //递归的出口
    if (s==t) {
        TR1[s]=SR[s];
    }else{
        int m=(s+t)/2;//每次递归将记录表中记录平分，直至每个记录各成一张表
        MSort(SR, TR2, s, m);//将分开的前半部分表中的记录进行排序
        MSort(SR,TR2, m+1, t);//将后半部分表中的记录进行归并排序
        Merge(TR2,TR1,s,m, t);//最后将前半部分和后半部分中的记录统一进行排序
    }
}
//归并排序
void MergeSort(SqList *L){
    MSort(L->r,L->r,1,L->length);
}

int main() {
    SqList * L=(SqList*)malloc(sizeof(SqList));
    L->length=7;
    L->r[1].key=49;
```

```
    L->r[2].key=38;
    L->r[3].key=65;
    L->r[4].key=97;
    L->r[5].key=76;
    L->r[6].key=13;
    L->r[7].key=27;
    MergeSort(L);
    for (int i=1; i<=L->length; i++) {
        printf("%d ",L->r[i].key);
    }
    return 0;
}
```

运行结果为：

```
13 27 38 49 65 76 97
```

提示：归并排序算法在具体实现时，首先需要将整个记录表进行折半分解，直到分解为一个记录作为单独的一张表为止，然后在进行两两合并。整个过程为分而后立的过程。

总结

归并排序算法的时间复杂度为 O(nlogn)。该算法相比于堆排序和快速排序，其主要的优点是：当记录表中含有值相同的记录时，排序前和排序后在表中的相对位置不会改变。

例如，在记录表中记录 a 在记录 b 的前面（记录 a 和 b 的关键字的值相等），使用归并排序之后记录 a 还在记录 b 的前面。这就体现出了该排序算法的稳定性。而堆排序和快速排序都是不稳定的。

基数排序及其 C 语言实现

基数排序不同于之前所介绍的各类排序，前边介绍到的排序方法或多或少的是通过使用比较和移动记录来实现排序，而基数排序的实现不需要进行对关键字的比较，只需要对关键字进行“分配”与“收集”两种操作即可完成。

例如对无序表{50，123，543，187，49，30，0，2，11，100}进行基数排序，由于每个关键字都是整数数值，且其中的最大值由个位、十位和百位构成，每个数位上的数字从 0 到 9，首先将各个关键字按照其个位数字的不同进行分配分配表如下图所示：

个位	
0	50、30、0、100
1	11
2	2
3	123、543
4	
5	
6	
7	187
8	
9	49

通过按照各关键字的个位数进行分配，按照顺序收集得到的序列变为：{50，30，0，100，11，2，123，543，187，49}。在该序列表的基础上，再按照各关键字的十位对各关键字进行分配，得到的分配表如下图所示：

十位	
0	0、100、2
1	11
2	123
3	30
4	543、49
5	50
6	
7	
8	187
9	

由上表顺序收集得到的记录表为：{0、100、2、11、123、30、543、49、50、187}。在该无序表的基础上，依次将表中的记录按照其关键字的百位进行分配，得到的分配如下图所示：

百位	
0	0、2、11、30、49、50
1	100、123、187
2	
3	
4	
5	543
6	
7	
8	
9	

最终通过三次分配与收集，最终得到的就是一个排好序的有序表：{0、2、11、30、49、50、100、123、187、543}。

例子中是按照个位-十位-百位的顺序进行基数排序，此种方式是从最低位开始排序，所以被称为最低位优先法（简称“LSD 法”）。

同样还可以按照百位-十位-各位的顺序进行排序，称为最高位优先法（简称“MSD 法”），使用该方式进行排序同最低位优先法不同的是：当无序表中的关键字进行分配后，相当于进入了多个子序列，后序的排序工作分别在各个子序列中进行（最低位优先法每次分配与收集都是相对于整个序列表而言的）。

例如还是对{50，123，543，187，49，30，0，2，11，100}使用最高位优先法进行排序，首先按照百位的不同进行分配，得到的分配表为：

	百位	
序列1：	0	50、49、30、0、2、11
序列2：	1	123、187、100
	2	
	3	
	4	
序列3：	5	543
	6	
	7	
	8	
	9	

由上图所示，整个无序表被分为了 3 个子序列，序列 1 和序列 2 中含有多个关键字，序列 3 中只包含了一个关键字，最高位优先法完成排序的标准为：直到每个子序列中只有一个关键字为止，所以需要分别对两个子序列进行再分配，各自的分配表如下图所示：

十位	
0	0、2
1	11
2	
3	30
4	49
5	50
6	
7	
8	
9	

（序列 1）

十位	
0	100
1	
2	123
3	
4	
5	
6	
7	
8	187
9	

（序列 2）

上表中，序列 1 中还有含有两个关键字的子序列，所以还需要根据个位进行分配，最终按照各子序列的顺序同样会得到一个有序表。

基数排序的具体实现（采用 LSD 法）

基数排序较宜使用链式存储结构作为存储结构，相比于顺序存储结构更节省排序过程中所需要的存储空间，称之为“链式基数排序”。

其具体的实现代码为：

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_NUM_OF_KEY 8//构成关键字的组成部分的最大个数
#define RADIX 10          //基数，例如关键字是数字，无疑由 0~9 组成，基数就是 10；如果关键字是字符串（字母组成），基数就是 26
#define MAX_SPACE 10000
//静态链表的结点结构
typedef struct{
    int data;//存储的关键字
    int keys[MAX_NUM_OF_KEY];//存储关键字的数组（此时是一位一位的存储在数组中）
    int next;//做指针用，用于是静态链表，所以每个结点中存储着下一个结点所在数组中的位置下标
}SLCell;
//静态链表结构
typedef struct{
    SLCell r[MAX_SPACE];//静态链表的可利用空间，其中 r[0]为头结点
    int keynum;//当前所有的关键字中最大的关键字所包含的位数，例如最大关键字是百，说明所有 keynum=3
    int recnum;//静态链表的当前长度
} SLList;

typedef int    ArrType[RADIX];//指针数组，用于记录各子序列的首尾位置
//排序的分配算法，i 表示按照分配的位次（是个位，十位还是百位），f 表示各子序列中第一个记录和最后一个记录的位置
void Distribute(SLCell *r,int i,ArrType f,ArrType e){
    //初始化指针数组
    for (int j=0; j<RADIX; j++) {
        f[j]=0;
    }
    //遍历各个关键字
    for (int p=r[0].next; p; p=r[p].next) {
        int j=r[p].keys[i];//取出每个关键字的第 i 位，由于采用的是最低位优先法，所以，例如，第 1 位指的就是每个关键字的个位
        if (!f[j]) {//如果只想该位数字的指针不存在，说明这是第一个关键字，直接记录该关键字的位置即可
            f[j]=p;
        }else{//如果存在，说明之前已经有同该关键字相同位的记录，所以需要将其进行连接，将最后一个相同的关键字的 next 指针指向该关键字所在的位置，同时最后移动尾指针的位置。
```

```
                r[e[j]].next=p;
            }
            e[j]=p;//移动尾指针的位置
        }
    }
    //基数排序的收集算法，即重新设置链表中各结点的尾指针
    void Collect(SLCell *r,int i,ArrType f,ArrType e){
        int j;
        //从 0 开始遍历，查找头指针不为空的情况，为空表明该位没有该类型的关键字
        for (j=0;!f[j]; j++);
        r[0].next=f[j];//重新设置头结点
        int t=e[j];//找到尾指针的位置
        while (j<RADIX) {
            for (j++; j<RADIX; j++) {
                if (f[j]) {
                    r[t].next=f[j];//重新连接下一个位次的首个关键字
                    t=e[j];//t 代表下一个位次的尾指针所在的位置
                }
            }
        }
        r[t].next=0;//0 表示链表结束
    }
    void RadixSort(SLList *L){
        ArrType f,e;
        //根据记录中所包含的关键字的最大位数，一位一位的进行分配与收集
        for (int i=0; i<L->keynum; i++) {
            //秉着先分配后收集的顺序
            Distribute(L->r, i, f, e);
            Collect(L->r, i, f, e);
        }
    }
    //创建静态链表
    void creatList(SLList * L){
        int key,i=1,j;
        scanf("%d",&key);
        while (key!=-1) {
            L->r[i].data=key;
```

```
            for (j=0; j<=L->keynum; j++) {
                L->r[i].keys[j]=key%10;
                key/=10;
            }
            L->r[i-1].next=i;
            i++;
            scanf("%d",&key);
        }
        L->recnum=i-1;
        L->r[L->recnum].next=0;
}
//输出静态链表
void print(SLList*L){
        for (int p=L->r[0].next; p; p=L->r[p].next) {
            printf("%d ",L->r[p].data);
        }
        printf("\n");
}
int main(int argc, const char * argv[]) {
        SLList *L=(SLList*)malloc(sizeof(SLList));
        L->keynum=3;
        L->recnum=0;
        creatList(L);//创建静态链表
        printf("排序前：");
        print(L);

        RadixSort(L);//对静态链表中的记录进行基数排序

        printf("排序后：");
        print(L);
        return 0;
}
```

运行结果为：

```
50
123
543
187
```

```
49
30
0
2
11
100
-1
排序前：50 123 543 187 49 30 0 2 11 100
排序后：0 2 11 30 49 50 100 123 187 543
```

内部排序算法的优势分析

本章介绍了以下几种常见的排序算法：

- 插入排序：直接插入排序、折半插入排序、2-路插入排序、表插入排序和希尔排序；
- 起泡排序（冒泡排序）；
- 快速排序（快排）；
- 选择排序：简单选择排序、树形选择排序和堆排序；
- 归并排序；
- 基数排序；

时间性能上的分析

排序方法	平均时间	最坏情况	辅助存储空间
简单排序	$O(n^2)$	$O(n^2)$	O(1)
快速排序	O(nlogn)	$O(n^2)$	O(logn)
堆排序	O(nlogn)	O(nlogn)	O(1)
归并排序	O(nlogn)	O(nlogn)	O(n)
基数排序	O(d*n)	O(d*n)	O(d*n)

上表中的简单排序包含出希尔排序之外的所有插入排序，起泡排序和简单选择排序。同时表格中的 n 表示无序表中记录的数量；基数排序中的 d 表示进行分配和收集的次数。

在上表表示的所有“简单排序算法”中，以直接插入排序算法最为简单，当无序表中的记录数量 n 较小时，选择该算法为最佳排序方法。

所有的排序算法中单就平均时间性能上分析，快速排序算法最佳，其运行所需的时间最短，但其在最坏的情况下的时间性能不如堆排序和归并排序；堆排序和归并排序相比较，当无序表中记录的数量 n 较大时，归并排序所需时间比堆排序短，但是在运行过程中所需的辅助存储空间更多（以空间换时间）。

从基数排序的时间复杂度上分析，该算法最适用于对 n 值很大但是关键字较小的序列进行排序。

在所有基于“比较”实现的排序算法中（以上排序算法中除了基数排序，都是基于“比较”实现），其在最坏情况下能达到的最好的时间复杂度为 O(nlogn)。

算法稳定性

本章所介绍的所有排序算法中，选择排序、快速排序和希尔排序都不是稳定的排序算法；而冒泡排序、插入排序、归并排序和基数排序都是稳定的排序算法。

算法实现的存储结构

本章所介绍的大多数算法都是在顺序存储结构的基础上实现的，基于顺序存储结构的局限性，排序算法在排序过程都需要进行大量记录的移动，影响算法本身的效率。

当无序表中记录的数量很大时，就需要采用静态链表替换顺序存储结构，例如：表插入排序、链式基数排序算法，是以修改指针代替大量移动记录的方式提高算法效率。

小结

通过比较所有的排序算法，没有哪一种是绝对最优的，在使用时需要根据不同的实际情况适当选择合适的排序算法，甚至可以考虑将多种排序算法结合起来使用。