

一、字符串

1. C 语言标准库函数

标准库 提供很多功能各异的函数，下面字符串函数均在 `string.h` 中定义；

`strcpy(s1, s2);`

复制字符串 `s2` 到字符串 `s1`。

`strcat(s1, s2);`

连接字符串 `s2` 到字符串 `s1` 的末尾。

`strlen(s1);`

返回字符串 `s1` 的长度。

`strcmp(s1, s2);`

如果 `s1` 和 `s2` 是相同的，则返回 0；如果 `s1<s2` 则返回小于 0；如果 `s1>s2` 则返回大于 0。

`strchr(s1, ch);`

返回一个指针，指向字符串 `s1` 中字符 `ch` 的第一次出现的位置。

`strstr(s1, s2);`

返回一个指针，指向字符串 `s1` 中字符串 `s2` 的第一次出现的位置。

```
int main()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len;

    /* 复制 str1 到 str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3);

    /* 连接 str1 和 str2 */
    strcat(str1, str2);
```

```

printf("strcat( str1, str2):  %s\n", str1);

/* 连接后, str1 的总长度 */
len = strlen(str1);
printf("strlen(str1) :  %d\n", len);

/* str1 str2比较 */
int i = strcmp(str1, str2);
printf("strcmp = %d\n", i);

/* 找到W第一次出现的位置, 并返回其后面的字符串 */
char* ch = strchr(str1, 'W');
printf("%s\n", ch);

/* 找到Wor第一次出现的位置, 并返回其后面的字符串 */
char *p = strstr(str1, "Wor");
printf("%s\n", p);
return 0;
}

```

2. 字符串匹配

1. 简单模式匹配算法

模式串跟主串左端对齐, 先比较第一个字符, 如果不一样就, 模式串后移, 直到第一个字符相等, 时间复杂度为 $O(n*m)$, n m 分别为主串和模式串的长度, 匹配过程如下:

第一步

主串 : baddef (匹配不上后移一位)

模式串: abc

第二步

主串 : baddef (匹配不上后移一位)

模式串: abc

第三步

主串 : baddef (匹配不上后移一位)

模式串: abc

第四步

主串 : baddef (匹配不上结束)

模式串: abc

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

```

int index(char s1[] ,int m, char s2[],int n) {
    int i = 1;
    int j = 1;

    while (i<= m && j<= n) {
        if (s1[i] == s2[j]){
            i++;
            j++;
        } else {
            i = i - j + 2;
            j = 1;
        }
    }

    if (j > n) {
        return i - n;
    } else {
        return 0;
    }
}

int main()
{
    char s[] = { ' ', 'a', 'b', 'c', 'a', 'b', 'a' }; //从序号1开始存
    char p[] = { ' ', 'a', 'b', 'a' };
    int sLen = sizeof(s) / sizeof(char) - 1;
    int pLen = sizeof(p) / sizeof(char) - 1;

    int idx = index(s, sLen, p, pLen);

    if (idx > 0) {
        printf("The matched idx is %d\n", idx);
    }
    else {
        printf("not matched!\n");
    }

    return 0;
}

```

II. KMP 算法

KMP 算法可以在 $O(m+n)$ 的时间复杂度上完成串的模式匹配，其改进在于：每当一趟匹配过程中出现字符比较不相等时，不需要回溯 **i** 指针，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较；

- 首先通过计算最长串前缀，获取一个 **next** 数组；
- 然后通过 **next** 数组，进行字符串匹配；

比如这里的模式串是： ababacd

- a 的前缀和后缀都是空，共有元素为 0
- ab 的前缀是[a]后缀是[b]，共有元素为 0
- aba 的前缀是[a, ab]后缀是[ba, a]，共有元素 a 的长度是 1
- abab 的前缀是[a, ab, aba]后缀是[bab, ab, b],共有元素是[ab]长度为 2
- ababa 的前缀是[a, ab, aba, abab] 后缀是[baba, aba, ba, a],最长共有元素是 [aba]长度是 3
- ababac 的前缀是[a,ab,aba,abab,ababa]后缀是[babac,abac,bac,ac,c]共有元素为 0
- ababacd 的前缀是[a,ab, aba, abab, ababa, ababac]后缀是[babacd, abacd, bacd, acd, cd, d]共有元素是 0

编号	1	2	3	4	5	6	7
Next	0	1	1	2	3	4	0

```
#include <stdio.h>

void get_next(char T[], int len, int next[]) {
    int i = 1;
    next[1] = 0;
    int j = 0;

    while (i <= len) {
        if (j == 0 || T[i] == T[j]) {
            i++; j++;
            next[i] = j;
        }
        else {
            j = next[j];
        }
    }
}

int KMP(char S[], int sLen, char T[], int tLen, int next[], int pos) {
    int i = 1;
    int j = 1;
    while (i <= sLen && j <= tLen) {
        if (j == 0 || S[i] == T[j]) {
            i++; j++;
        }
        else {
            j = next[j];
        }
    }
}
```

```

        j = next[j];
    }
}

if (j > tLen) {
    return i - tLen;
}

else {
    return 0;
}
}

int main()
{
    char s[] = { ' ', 'a', 'b', 'c', 'a', 'b', 'a' }; //从序号1开始存
    char t[] = { ' ', 'a', 'b', 'a' };

    int sLen = sizeof(s) / sizeof(char) - 1;
    int tLen = sizeof(t) / sizeof(char) - 1;
    int next[10] = {0};

    //获取next数组
    get_next(t, tLen, next);
    int idx = KMP(s, sLen, t, tLen, next, 0);

    if (idx > 0) {
        printf("The matched idx is %d\n", idx);
    }
    else {
        printf("not matched!\n");
    }
    return 0;
}

```

输出结果为：

```

$ ./result.exe
The matched idx is 4

```

二、线性表

1. 线性表定义

线性表是 n 个数据特性相同的元素的组成有限序列, 是最基本且常用的一种线性结构(线性表, 栈, 队列, 串和数组都是线性结构), 同时也是其他数据结构的基础。

对于非空的线性表或者线性结构的特点：

- 表中元素个数有限；
- 存在唯一的一个被称作“第一个”的数据元素；
- 存在唯一的一个被称作“最后一个”的数据元素；
- 除第一个外，结构中的每个数据元素均只有一个前驱；
- 除最后一个外，结构中的每个数据元素均只有一个后继；

2. 线性表的两种实现方式

按照存储方式可以分为如下两种：

顺序存储

- 顺序表

链式存储

- 单链表
- 双链表
- 循环链表
- 静态链表（借助数组实现）

2.1 顺序表

概念

用一组地址连续的存储单元依次存储线性表的数据元素，这种存储结构的线性表称为顺序表。

特点

逻辑上相邻的数据元素，物理次序也是相邻的。

- 只要确定好了存储线性表的起始位置，线性表中任一数据元素都可以随机存取，所以线性表的顺序存储结构是一种随机存取的存储结构；
- 因为高级语言中的数组类型也是有随机存取的特性，所以通常我们都使用数组来描述数据结构中的顺序储存结构，用动态分配的一维数组表示线性表。

顺序表初始化

一维数组可以静态分配，也可以动态分配。在静态分配时，由于数组的大小和空间事先已经固定，一旦空间占满，再加入新的数据将产生溢出，就会导致程序崩溃。

动态分配时，存储数组的空间是在程序执行过程中通过动态存储分配语句分配的，一旦数据空间占满，可以另外开辟一块更大的存储空间，从而达到扩充存储数组空间的目的。

//静态分配

```
typedef struct {
    ElemType data[maxsize];
    int length;
}SqList;
```

//动态分配

```
typedef struct {
    ElemType *data; // 指示动态分配数组的指针;
    int MaxSize, length; //数组最大容量和当前个数;
}SeqList;
```

//线性表初始化

```
void InitList(SeqList* L, int length) {
    L->data = (ElemType*)malloc(sizeof(ElemType)*length);

    if (L->data == NULL) {
        printf("内存分配失败! \n");
    }
    else {
        L->MaxSize = length;
        L->length = 0;
    }
}
```

顺序表元素插入

//数据插入

```
int ListInsert(SeqList &L, int i, ElemType e) {
    if (i<1 || i>L.length + 1)
        return ERROR;

    if (L.length > maxsize)
        return ERROR;

    for (int j = L.length; j >= i; j--) {
        L.data[j] = L.data[j - 1];
    }
    L.data[i - 1] = e;
    L.length++;
    return OK;
}
```

```
}
```

顺序表元素删除

//数据删除

```
int ListDelete(SeqList &L, int i, ElemType &e) {  
    if (i<1 || i>L.length) {  
        return ERROR;  
    }  
    e = L.data[i - 1];  
    for (int j = i; j < L.length; j++) {  
        L.data[j - 1] = L.data[j];  
    }  
    L.length--;  
    return OK;  
}
```

测试用例 (demo_2_1_sqliist)

```
int main()  
{  
    SeqList list;  
    InitList(&list, 20);  
    printf("insert...\n");  
    ListInsert(list, 1, 12);  
    ListInsert(list, 2, 13);  
    ListInsert(list, 3, 14);  
    ListInsert(list, 4, 15);  
    ListInsert(list, 5, 16);  
    traverse(list);  
  
    printf("delete elem idx = 3\n");  
    ElemType val;  
    ListDelete(list, 3, val);  
    printf("have deleted, val = %d\n", val);  
    traverse(list);  
    return 0;  
}
```

运行结果


```

$ ./result.exe
insert...
idx 1 = 12
idx 2 = 13
idx 3 = 14
idx 4 = 15
idx 5 = 16
delete elem idx = 3
have deleted, val = 14
idx 1 = 12
idx 2 = 13
idx 3 = 15
idx 4 = 16

```

2.2 链表

概念

用一组任意的存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的），包括数据域和指针域，数据域存数据，指针域指示其后继的信息。

//单链表存储结构

```

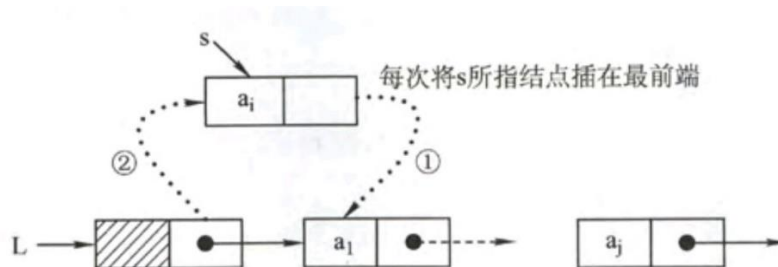
typedef struct LNode
{
    ElemType data;    //数据域
    struct LNode *next; //指针域
}LNode,*LinkList;

```

链表的插入

可以分为头插法和尾插法，使用尾插法时需要增加一个尾指针；

如下为头插法示意图：



```

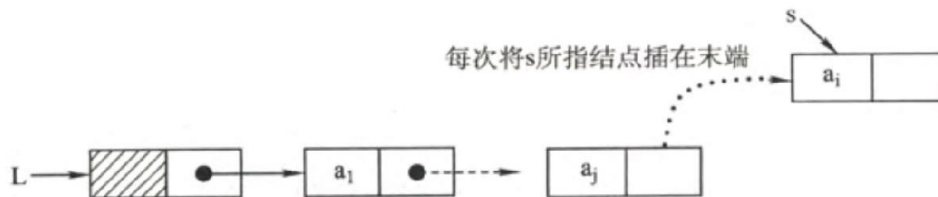
//头插法
LinkedList CreateListHead(ElemType data[], int n)
{
    LNode * L;
    InitList(L);

    LNode *s;

    for (int i = 0; i < n; i++) {
        s = (LNode *)malloc(sizeof(LinkedList));
        s->data = data[i];
        s->next = L->next;
        L->next = s;
    }
    return L;
}

```

如下为尾插法示意图



```

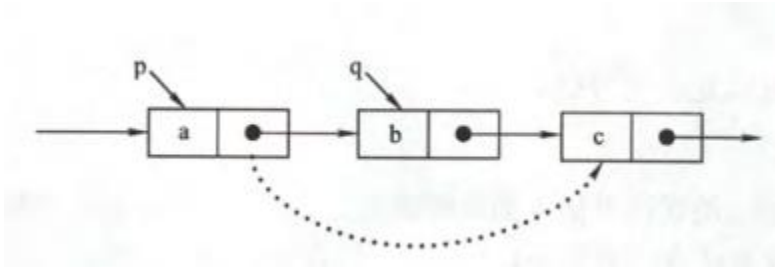
//尾插法
LinkedList CreateListTail(ElemType data[], int n)
{
    LNode * L;
    InitList(L);

    LNode *s, *r = L;

    for (int i = 0; i < n; i++) {
        s = (LNode *)malloc(sizeof(LinkedList));
        s->data = data[i];
        r->next = s;
        r = s;
        r->next = NULL;
    }
    return L;
}

```

链表结点删除



首先找到被删除结点的前驱结点*p，然后修改*p的指针域，将*p的指针域 next 指向*q 的下一个结点；

该算法的主要时间消耗在查找操作上，时间复杂度为 $O(n)$ 。

//删除结点

```
int ListDelete(LinkList &L, int i, ElemType &e) {
    LNode * p = L;
    LNode * q;
    int j = 0;

    while (p->next && j < i - 1) {
        p = p->next;
        j++;
    }

    if (!(p->next) || j > i - 1) {
        return ERROR;
    }

    q = p->next;
    p->next = q->next;
    e = q->data;
    free(q);
    return OK;
}
```

测试用例 (demo_2_2_linklist)

```
int main() {
    ElemType arr[] = {12, 13, 14, 15, 16, 17};
    int arrLen = sizeof(arr) / sizeof(ElemType);

    printf("insert order\n");
    for (int i = 0; i < arrLen; i++) {
        printf("%d ", arr[i]);
    }

    LinkList L1 = CreateListHead(arr, arrLen);
}
```

```

printf("\nhead insert result:\n");
traverse(L1);

LinkedList L2 = CreateListTail(arr, arrLen);
printf("\ntail insert result:\n");
traverse(L2);

printf("\ndelete elem idx = 2\n");
ElemType val;
ListDelete(L2, 2, val);
printf("have deleted, val = %d\n", val);
printf("current seq\n");
traverse(L2);
return 0;
}

```

```

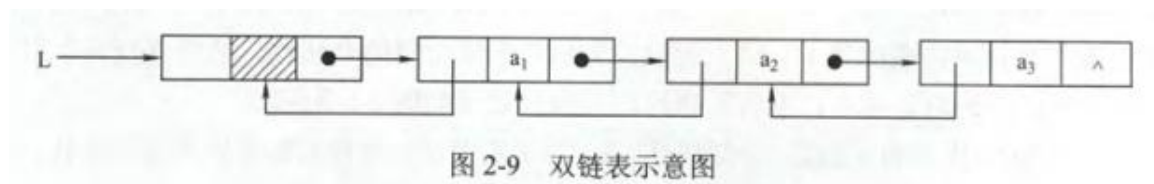
$ ./result.exe
insert order
12 13 14 15 16 17
head insert result:
17 16 15 14 13 12
tail insert result:
12 13 14 15 16 17
delete elem idx = 2
have deleted, val = 13
current seq
12 14 15 16 17

```

2.3 双向链表、循环链表、静态链表

双向链表

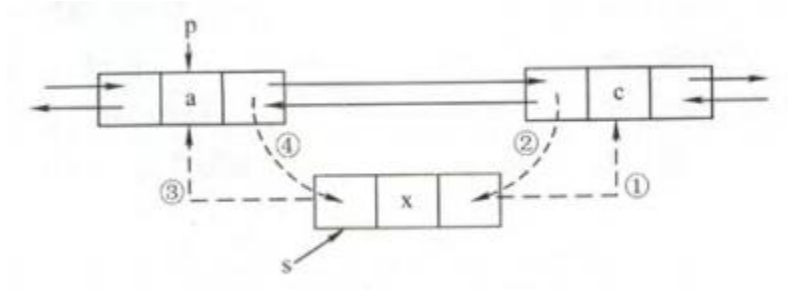
- 单链表的一个优点是结构简单，但是它也有一个缺点，即在单链表中只能通过一个结点的引用访问其后续结点，而无法直接访问其前驱结点，
- 要在单链表中找到某个结点的前驱结点，必须从链表的首结点出发依次向后寻找，为此我们可以扩展单链表的结点结构，使得通过一个结点的引用，不但能够访问其后续结点，也可以方便的访问其前驱结点。
- 扩展单链表结点结构的方法是，在单链表结点结构中新增加一个域，该域用于指向结点的直接前驱结点。



其结点类型描述如下：

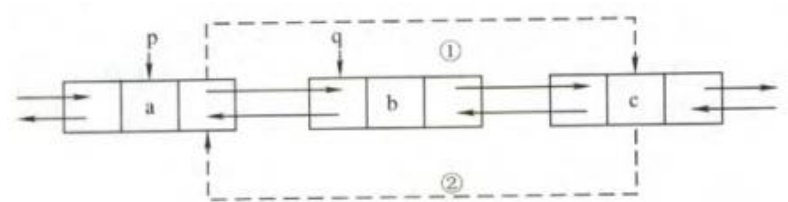
```
typedef struct DNode{
    ElemType data;
    struct DNode *prior,*next;
}DNode, *DLinklist;
```

双链表插入



```
s->next=p->next;
p->next->prior=s;
s->prior=p;
p->next=s;
```

双链表删除



```
p->next=q->next;
q->next->prior=p;
free(q);
```

循环链表

循环单链表

循环单链表和单链表的区别在于，表中最后一个结点的指针不是 NULL，而是改为相应的头结点，使整个链表形成一个环。

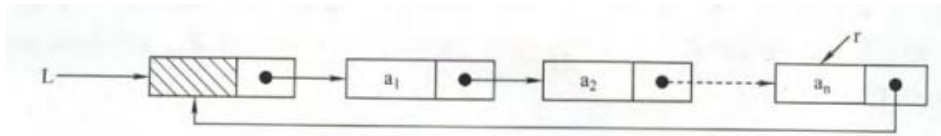


图 2-12 循环单链表

循环双链表

与循环单链表不同的是，循环双链表的头结点的 **prior** 指针还要指向表尾结点；

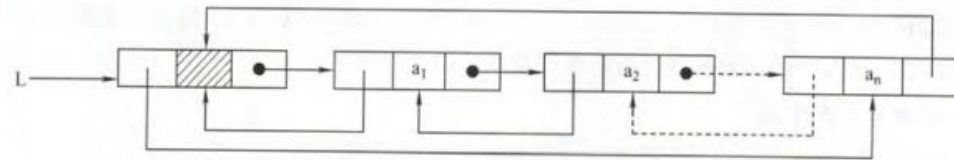


图 2-13 循环双链表

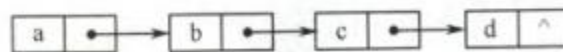
静态链表

静态链表是借助数组来描述线性表的链式存储结构，结点也有数据域 **data** 和指针域 **next**，与前面链表不同的是，这里的指针是结点的相对地址（数组下标）；

与顺序表一样，静态链表也要预先分配一块连续的内存空间。

0		2
1	b	6
2	a	1
3	d	-1
4		
5		
6	c	3

(a)静态链表示例



(b)静态链表对应的单链表

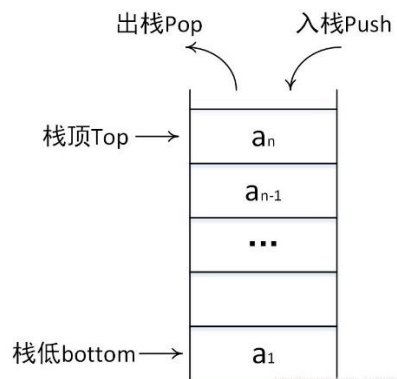
三、栈

1. 栈的逻辑结构

定义

- 栈 (Stack) 是一种特殊的线性表，它所有的插入和删除都限制在表的同一端进行。
- 栈中允许进行插入、删除操作的一端叫做栈顶 (Top)，另一端则叫做栈底 (Bottom)。当栈中没有元素时，称之为空栈。栈的插入运算通常称为压栈、进栈或入栈 (Push)，栈的删除运算通常称为弹栈或出栈 (Pop)。

示意图如下：



栈与线性表的区别与联系

- 栈是特殊的线性表；
- 栈的插入与删除运算只能在栈顶进行，而线性表的插入和删除运算可在线性表中的任意位置进行；

栈的相关运算

InitStack(&S):

栈的初始化操作：建立一个空栈 S。

StackEmpty(S)

判栈空：判断栈是否为空。

GetTop(S,&x)

取栈顶元素：取得栈顶元素的值。

Push(&S, &x)

入栈：在栈 S 中插入元素 e，使其成为新的栈顶元素。

Pop(&S,&x)

出栈：删除栈 S 的栈顶元素。

ClearStack(&S)

销毁栈

2. 存储结构

根据存储结构的不同，可以分为顺序栈和链式栈；

2.1 顺序栈

栈的顺序存储结构需要使用一个数组和一个整形变量来实现。利用数组来顺序存储栈中的所有元素，利用整形变量来存储栈顶元素的下标位置。

```
typedef struct{
    ElemType data[Max_Size];
    int top;
}SqStack;
```

初始化

```
//初始化
void InitStack(SqStack *s)    //要更改栈内数据，需要传址调用，或者采取引用方式
{
    s->top = -1;
}
```

判断栈空

```
//判断栈空
int StackEmpty(SqStack s)
{
    if (s.top == -1) {
        return TRUE;
    }
    else {
        return FALSE;
    }
}
```


入栈

```
//入栈
int Push(SqStack &S, ElemType x)
{
    if (S.top==Max_Size-1)
        return FALSE;
    else
    {
        S.top++;
        S.data[S.top] = x;
    }
    return TRUE;
}
```

出栈

```
//出栈
int Pop(SqStack &S, ElemType &x)
{
    if (StackEmpty(S))
        return FALSE;
    else
    {
        x = S.data[S.top];
        S.top--;
    }
    return TRUE;
}
```

获取栈顶元素

```
//获取栈顶元素
int GetTop(SqStack S, ElemType &x) {
    if (StackEmpty(S))
        return FALSE;

    x = S.data[S.top];
    return TRUE;
}
```

测试用例 (demo_3_1_stack)

```
int main() {
    SqStack S;

    ElemType arr[] = {11, 12, 13, 14, 15};
    int sLen = sizeof(arr) / sizeof(ElemType);

    InitStack(&S);
```

```

    for (int i = 0; i < sLen; i++) {
        printf("push element is %d\n", arr[i]);
        Push(S, arr[i]);
    }

    ElemType val;

    printf("---\n", val);

    while (!StackEmpty(S))
    {
        GetTop(S, val);
        printf("top element is %d\n", val);
        printf("exe pop...\n");
        Pop(S, val);
    }

    printf("stack is null!\n");
}

```

结果

```

$ ./result.exe
push element is 11
push element is 12
push element is 13
push element is 14
push element is 15
---
top element is 15
exe pop...
top element is 14
exe pop...
top element is 13
exe pop...
top element is 12
exe pop...
top element is 11
exe pop...
stack is null!

```

2.2 链栈

顺序栈存在栈满以后就不能在进栈的问题，这是因为用了定长的数组存储栈的元素。解决的方法可以是使用链式存储结构，让栈空间可以动态扩充。

栈的链式存储结构成为链栈，它是运算受限的单链表，其插入和删除操作仅限制在表头位置上进行。由于只能在链表头部进行操作，故链栈没有必要像链表那样附加头结点。栈顶指针就是链表的头指针。

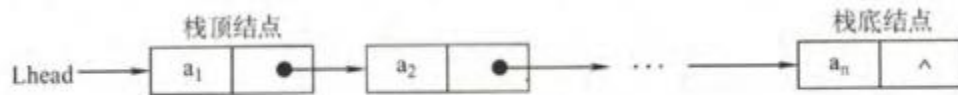


图 3-4 栈的链式存储

链式存储类型可描述为：

```
typedef struct Linknode{
    ElemType data;
    struct Linknode *next;
} *LiStack;
```

采用链式存储，便于结点的插入和删除。

四、队列

1.定义

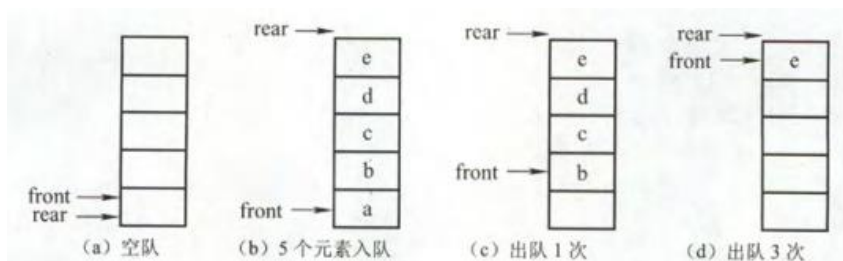
- 队列(queue)是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。
- 队列是一种先进先出(first in first out)的线性表，简称 FIFO。允许插入的一端称为队尾(rear)，允许删除的一端称为队头(Front)。
- 队列是一种运算受限的线性表，它的运算限制与栈不同，是两头都有限制，插入只能在表的一端进行(只进不出)，而删除只能在表的另一端进行(只出不进)。

2.队列的存储结构

队列也是线性表，所以有两种存储方式，顺序存储和链式存储。

2.1 顺序队列

存储空间不够用的时候需要开发人员通过编程手段来扩展数组容量



2.2 循环队列

在上述场景 d，在 $Q.rear = \text{MaxSize}$ 时并不能作为队列满的条件，这是入队出现“上溢出”并不是真正的溢出，在 `data` 数组中依然存在可以存放元素的空位置，是一种“假溢出”。

引入循环队列可以避免顺序队列的缺点。把存储队列元素的表从逻辑上看成一个环，称为循环队列。当队首指针 $Q.front = \text{MaxSize} - 1$ 后，再前进一个位置自动到 0。

初始时: $Q.front = Q.rear = 0$

队首指针进 1: $Q.front = (Q.front + 1) \% \text{MaxSize}$

队尾指针进 1: $Q.rear = (Q.rear + 1) \% \text{MaxSize}$

队列长度: $(Q.rear + \text{MaxSize} - Q.front) \% \text{MaxSize}$

出队入队时: 指针都按顺时针方向进 1 (如图 3-7 所示)。

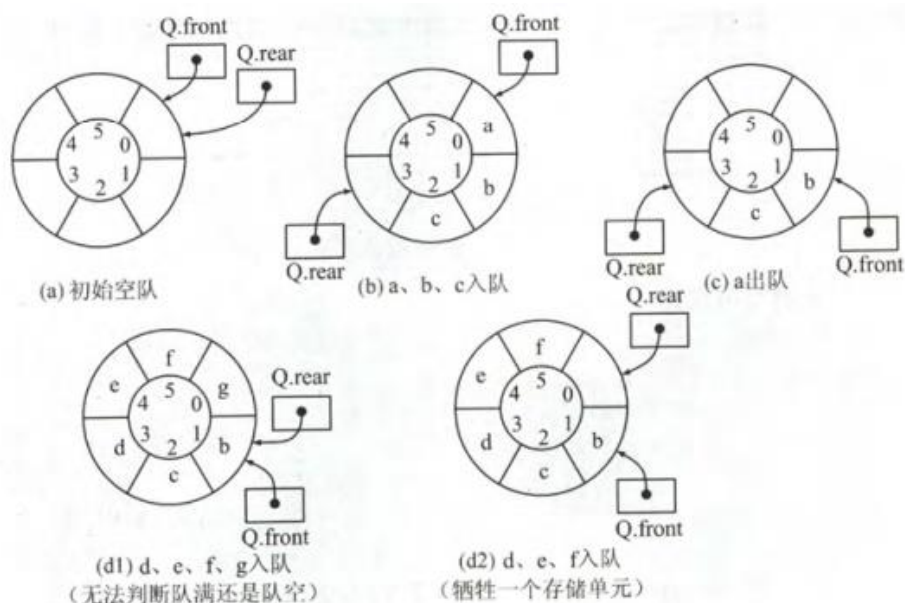


图 3-7 循环队列出入队示意图

为了区分队满、队空，有下面三种处理方式

牺牲一个单元来区分队满和队空。

队满条件为： $(Q.rear+1)\%MaxSize==Q.front$ 。

队空条件仍为： $Q.front==Q.rear$ 。

队列中元素的个数： $(Q.rear-Q.front+MaxSize)\%MaxSize$

类型中增设表示元素个数的成员变量。

类型中增设 tag 数据成员；Tag=0 表示队空，tag=1 表示队满。

循环队列基本操作

(1) 初始化

```
void InitQueue(&Q){
    Q.rear=Q.front=0;           //初始化队首、队尾指针
}
```

(2) 判队空

```
bool isEmpty(Q){
    if(Q.rear==Q.front) return true;   //队空条件
    else return false;
}
```

(3) 入队

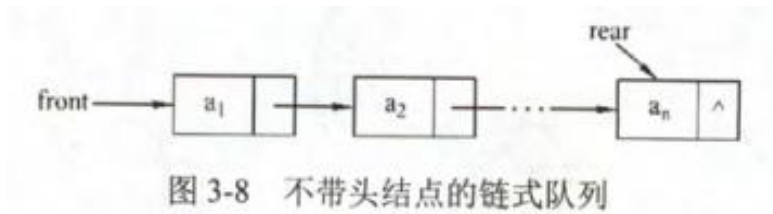
```
bool EnQueue(SqQueue &Q, ElemType x){
    if((Q.rear+1)%MaxSize==Q.front) return false; //队满
    Q.data[Q.rear]=x;
    Q.rear=(Q.rear+1)%MaxSize;           //队尾指针加1取模
    return true;
}
```

(4) 出队

```
bool DeQueue(SqQueue &Q, ElemType &x){
    if(Q.rear==Q.front) return false; //队空，报错
    x=Q.data[Q.front];
    Q.front=(Q.front+1)%MaxSize;        //队头指针加1取模
    return true;
}
```

2.3 链式存储

队列的链式表示称为链式队列，他实际上也是个同时带有队头指针和队尾指针的单链表。头指针指向队头结点，尾指针指向队尾结点。



队列的存储类型可描述为：

```
//链式队列结点
typedef struct LinkNode
{
    ElemType data;
    struct LinkNode *next;
}LinkNode;

//链式队列
typedef struct
{
    LinkNode *front, *rear;    //队头和队尾指针
}LinkQueue;
```

- 当 $Q.front = NULL$ 且 $Q.rear = NULL$ 时，链式队列为空；
- 用单链表表示的链式队列特别适合于数据元素变动比较大的情形，而且不存在队列溢出的问题。

链式队列的基本操作

初始化

```
//初始化
void InitQueue(LinkQueue &Q) {
    Q.front = Q.rear = (LinkNode*)malloc(sizeof(LinkNode));
    Q.front->next = NULL;
}
```

判断队空

```
//判断队空
int IsEmpty(LinkQueue Q) {
    if (Q.front == Q.rear)
        return TRUE;
    else
        return FALSE;
}
```

入队

```
void EnQueue(LinkQueue &Q, ElemType x) {
    LinkNode* s;
    s = (LinkNode *)malloc(sizeof(LinkNode));
    s->data = x;
    s->next = NULL;
    Q.rear->next = s;
    Q.rear = s;
}
```

出队

```
int Dequeue(LinkQueue &Q, ElemType &x) {
    if (IsEmpty(Q)) {
        return FALSE;
    }
    LinkNode *p = Q.front->next;
    x = p->data;
    Q.front->next = p->next;

    if (Q.rear == p) {
        Q.rear = Q.front; // 如果队列只有一个结点，删除后队列为空;
    }
    free(p);
    return TRUE;
}
```

测试用例 (demo_4_1_queue)

```
int main() {

    LinkQueue Q;
    InitQueue(Q);

    ElemType arr[] = { 11, 12, 13, 14, 15 };
    int arrLen = sizeof(arr) / sizeof(ElemType);

    printf("enqueue...\n");
    for (int i = 0; i < arrLen; i++) {
        EnQueue(Q, arr[i]);
        printf("%d ", arr[i]);
    }
    printf("\ncurrent elem in queue:\n");
    traverse(Q);

    ElemType val;
    printf("\ndequeue...\n", val);

    while (!IsEmpty(Q))
    {
        Dequeue(Q, val);
        printf("dequeue element is %d\n", val);
    }
    printf("queue is null!\n");
}
```

```
    return 0;  
}
```

```
$ ./result.exe  
enqueue...  
11 12 13 14 15  
current elem in queue:  
11 12 13 14 15  
dequeue...  
dequeue element is 11  
dequeue element is 12  
dequeue element is 13  
dequeue element is 14  
dequeue element is 15  
queue is null!
```

五、哈希表

1. 什么是哈希表

哈希表是唯一的专用于集合的数据结构。可以以常量的平均时间实现插入、删除和查找。

用一个与集合规模差不多大的数组来存储这个集合，将数据元素的关键字映射到数组的下标，这个映射称为“**散列函数**”，数组称为“**散列表**”。查找时，根据被查找的关键字找到存储数据元素的地址，从而获取数据元素。

2. 散列函数

在散列表中。插入、删除和查找都会用到散列函数。散列函数的计算速度直接影响散列表的性能。好的散列函数的一个标准就是：**计算速度快**。另一点就是：结点的**散列地址尽可能均匀**。使得冲突的机会尽可能少。

常用的散列函数包括直接定址法、保留余数法、数字分析法、平方取中法和折叠法等。

1) 直接定址法

哈希函数为关键字的线性函数如 $H(key) = a * key + b$

2) 保留余数法

$H(key) = key \% p$ ($p \leq m$ m 为表长)

- 3) 数字分析法
- 4) 平方取中法
- 5) 折叠法

3. 哈希冲突

即不同 key 值产生相同的地址, $H(\text{key1}) = H(\text{key2})$

比如我们上面说的存储 3 6 9, p 取 3 是

$3 \text{ MOD } 3 == 6 \text{ MOD } 3 == 9 \text{ MOD } 3$

此时 3 6 9 都发生了 hash 冲突

4. 哈希冲突 的解决方案

不管 hash 函数设计的如何巧妙, 总会有特殊的 key 导致 hash 冲突, 特别是对动态查找表来说。

hash 函数解决冲突的方法有以下几个常用的方法:

1. 开放定址法

- 1) 线性探测 再散列
- 2) 平方探测 再散列
- 3) 随机探测 再散列 (双探测 再散列)

2. 链地址法

下面以开放定址为例进行一次数据存储:

使用开放地址方法解决冲突的时候, 数据仍然保存在 hash 表的存储单元中, 但是当冲突发生的时候, 要再次计算新的地址。

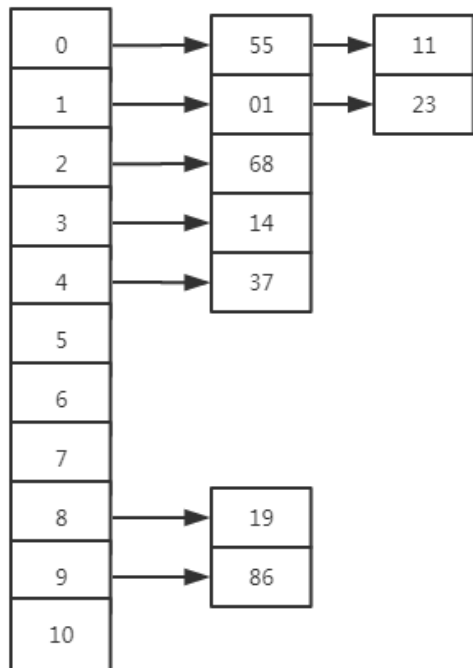
常用的开放地址法是线性探查, 就是当对一个数据进行插入删除或者查找的时候, 通过 hash 函数计算, 发现这个位置不是要找的数据, 这时候就检查下一个存储单元, 一直找到要操作的数据为止。

19 01 23 14 55 68 11 86 37 要存储在表长 11 的数组中, 其中 $H(\text{key}) = \text{key} \text{ MOD } 11$ 举例, (从前向后插入数据, 如果插入位置已经占用, 发生冲突, 冲突的另起一行, 计算地址, 直到地址可用, 后面冲突的继续向下另起一行。最终结果取最上面的数据 (因为是最“占座”的数据))

index	0	1	2	3	4	5	6	7	8	9	10
key	55	1		14					19	86	
23 冲突	23										
68 冲突	68 冲突	68									
11 冲突	11 冲突	11 冲突	11 冲突	11 冲突	11						
37 冲突	37 冲突	37									
最终结果	55	1	23	14	68	11	37		19	86	

下面以[链地址](#)为例进行一次数据存储；

产生 hash 冲突后在存储数据后面加一个指针，指向后面冲突的数据
上面的例子，用链地址法则是下面这样：



5. 哈希表的删除

首先链地址法是可以直接删除元素的，但是开放定址法是不行的，拿前面的双探测再散列来说，假如我们删除了元素 1，将其位置置空，那 23 就永远找不到了。正确做法应该是删除之后置入一个原来不存在的数据，比如 -1

六、树

1. 二叉树

定义

树是一种数据结构，它是由 n ($n \geq 1$) 个有限节点组成一个具有层次关系的集合。

特点

- 每个结点最多有两颗子树，所以二叉树中不存在度大于 2 的结点。
- 左子树和右子树是有顺序的，次序不能任意颠倒。
- 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。

二叉树遍历

- 先序遍历
- 中序遍历
- 后序遍历
- 层序遍历

// 先序遍历二叉树

```
void PreOrder(BiTree T)
{
    if (T) {
        printf("%d ", T->data);
        PreOrder(T->lchild);
        PreOrder(T->rchild);
    }
}
```

// 中序遍历二叉树

```
void InOrder(BiTree T)
{
    if (T) {
```

```

        InOrder(T->lchild);
        printf("%d ", T->data);
        InOrder(T->rchild);
    }
}

// 后序遍历二叉树
void PostOrder(BiTree T)
{
    if (T) {
        PostOrder(T->lchild);
        PostOrder(T->rchild);
        printf("%d ", T->data);
    }
}

```

2. 二叉查找树

2.1 定义

二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树：

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉排序树。

2.2 二叉排序树的查找

- 1) 若根结点的关键字值等于查找的关键字，成功。
- 2) 否则，若小于根结点的关键字值，递归查左子树。
- 3) 若大于根结点的关键字值，递归查右子树。

2.3 二叉排序树的插入

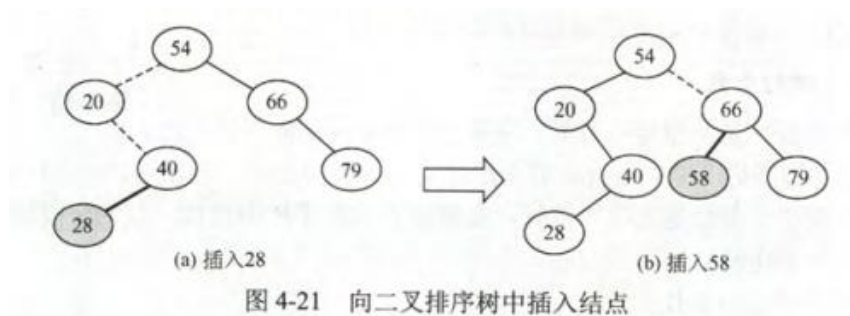
二叉树作为一种动态集合，其特点是树的结构不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。

- 1) 首先执行查找算法，找出被插结点的父亲结点。
- 2) 判断被插结点是其父亲结点的左、右儿子。将被插结点作为叶子结点插入。
- 3) 若二叉树为空。则首先单独生成根结点。

注意：新插入的结点总是叶子结点。

//二叉排序树插入

```
int BST_Insert(BiTree &T, ElemType k) {
    if (T == NULL) {
        T = (BiTree)malloc(sizeof(BiTreeNode));
        T->data = k;
        T->lchild = T->rchild = NULL;
        return TRUE;
    }
    else if (T->data == k) {
        return FALSE; //存在相同的关键字
    }
    else if (T->data > k) //左子树中插入
        return BST_Insert(T->lchild, k);
    else if (T->data < k) //右子树中插入
        return BST_Insert(T->rchild, k);
}
```



2.4 二叉排序树的构造

构造一颗二叉排序树就是依次输入数据元素，并将它们插入到二叉排序树中的适当位置上的过程。

具体过程是，每读入一个元素，就建立一个结点，若二叉排序树非空，则将新节点的值与根结点的比较，如果小于根结点，插入到左子树中，否则插入到右子树中；

若二叉排序树为空，则新节点作为二叉排序树的根结点

//二叉排序树创建

```
void Create_BST(BiTree &T, ElemType arr[], int n) {
    T = NULL;
    for (int i = 0; i < n; i++) {
        int flag = BST_Insert(T, arr[i]);
        printf("%d %d %d\n", i, arr[i], flag);
    }
}
```

2.5 二叉排序树的删除

删除操作有三种情况：

- 1) 如果被删除结点 z 是叶结点，直接进行删除操作即可
- 2) 如果结点 z 只有一颗左子树或右子树，则让 z 的子树成为 z 父结点的子树
- 3) 如果结点 z 有左右两颗子树，则令 z 的直接后继（或直接前驱）替代 z ，然后从二叉排序树中删除这个直接后继（或直接前驱），转换成第一或第二种情况。（中序遍历的下一个）（不断取左子树的最大或者右子树的最小递归删除）

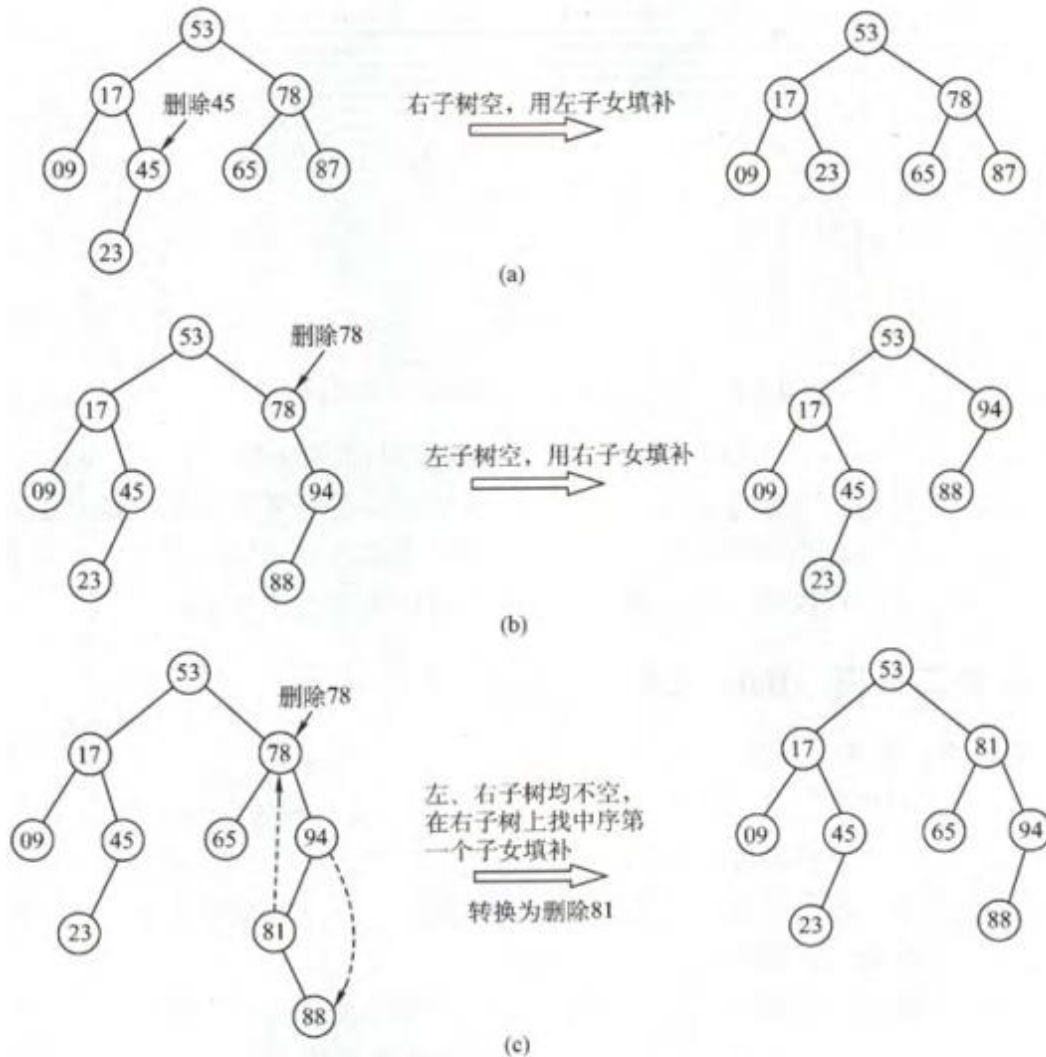


图 4-22 3 种情况下的删除过程

测试用例 (demo_5-1_binary_sort_tree)

```
int main() {  
    ElemType arr[] = { 53, 17, 78, 9, 45, 65, 94, 23, 81, 88 };
```

```

int arrLen = sizeof(arr) / sizeof(ElemType);

BiTree T;
Create_BST(T, arr, arrLen);

printf("insert order: 53,17,78,9,45,65,94,23,81,88");
printf("\npreorder result:\n");
PreOrder(T);
printf("\nin order result:\n");
InOrder(T);
printf("\npost order result:\n");
PostOrder(T);

DeleteBST(T, 78);
printf("\ndelete data 78:\n");
printf("preorder result:\n");
PreOrder(T);
printf("\nin order result:\n");
InOrder(T);
printf("\npost order result:\n");
PostOrder(T);
}

```

```

$ ./result.exe
insert order: 53,17,78,9,45,65,94,23,81,88
preorder result:
53 17 9 45 23 78 65 94 81 88
in order result:
9 17 23 45 53 65 78 81 88 94
post order result:
9 23 45 17 65 88 81 94 78 53
delete data 78:
preorder result:
53 17 9 45 23 81 65 94 88
in order result:
9 17 23 45 53 65 81 88 94
post order result:
9 23 45 17 65 88 94 81 53

```

3. 平衡二叉树

定义

它或者是一颗空树，或者具有以下性质的二叉排序树：它的左子树和右子树的深度之差(平衡因子)的绝对值不超过 1，且它的左子树和右子树都是一颗平衡二叉树。

由于普通的二叉查找树会容易失去“平衡”，**极端情况下，二叉查找树会退化成线性的链表，导致插入和查找的复杂度下降到 $O(n)$** ，所以，这也是平衡二叉树设计的初衷。

最小失衡子树

在新插入的结点向上查找，以第一个平衡因子的绝对值超过 1 的结点为根的子树称为最小不平衡子树。也就是说，一棵失衡的树，是有可能有多棵子树同时失衡的，**而这个时候，我们只要调整最小的不平衡子树，就能够将不平衡的树调整为平衡的树。**

AVL 树的平衡调整

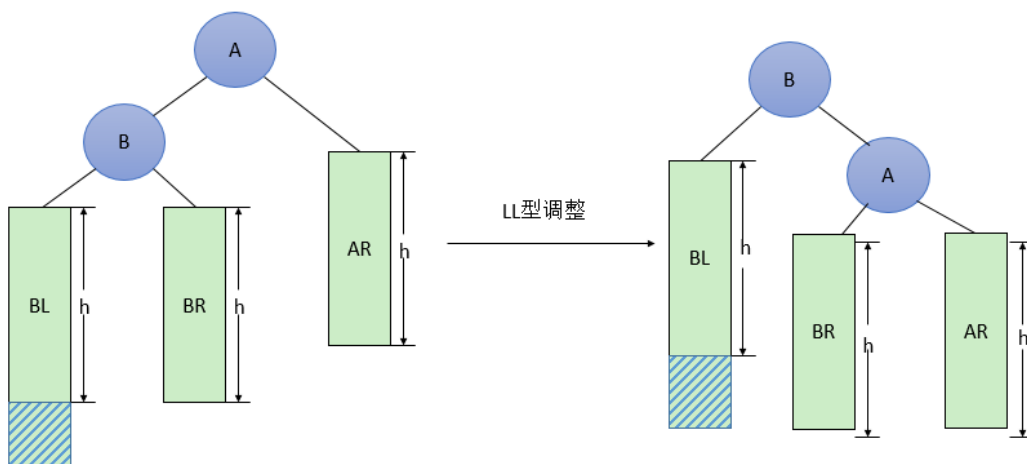
整个实现过程是通过在一棵平衡二叉树中依次插入元素(按照二叉排序树的方式)，若出现不平衡，则要根据新插入的结点与最低不平衡结点的位置关系进行相应的调整。分为 **LL 型、RR 型、LR 型和 RL 型** 4 种类型，各调整方法如下(下面用 A 表示最低不平衡结点):

LL 型调整 (右单旋转)

由于在 A 的左孩子(L)的左子树(L)上插入新结点，使原来平衡二叉树变得不平衡，此时 A 的平衡因子由 1 增至 2。

LL 型调整的一般形式：

- ① 将 A 的左孩子 B 提升为新的根结点；
- ② 将原来的根结点 A 降为 B 的右孩子；

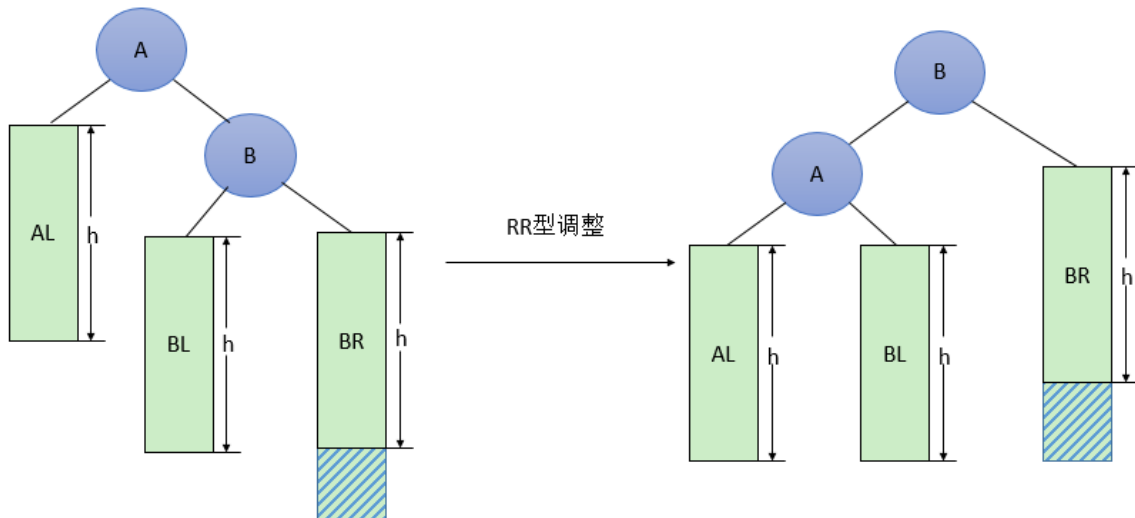


RR 型调整 (左单旋转)

由于在 A 的右孩子(R)的右子树(R)上插入新结点, 使原来平衡二叉树变得不平衡, 此时 A 的平衡因子由-1 变为-2。

RR 型调整的一般形式:

- ① 将 A 的右孩子 B 提升为新的根结点;
- ② 将原来的根结点 A 降为 B 的左孩子

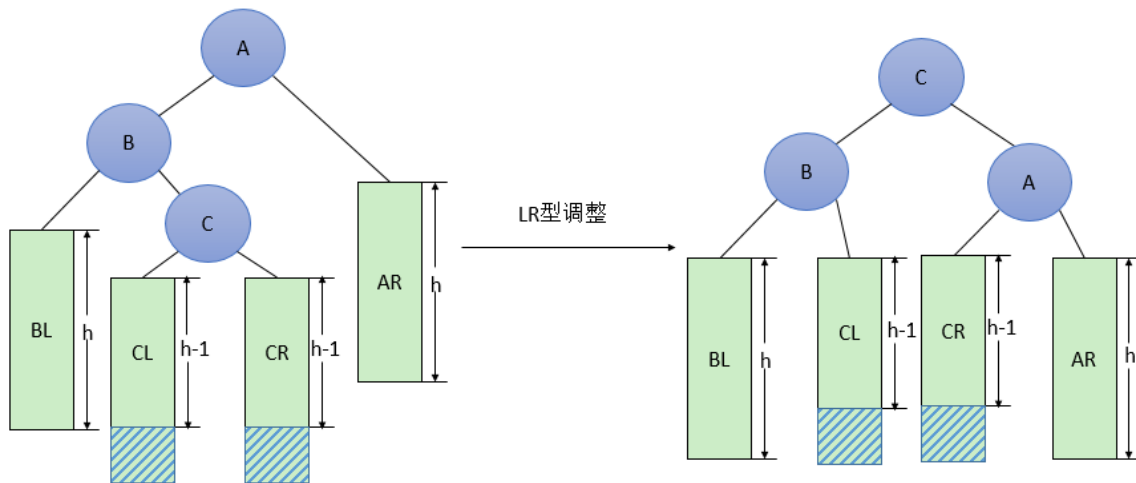


LR 型调整 (先左后右双旋转)

由于在 A 的左孩子(L)的右子树(R)上插入新结点, 使原来平衡二叉树变得不平衡, 此时 A 的平衡因子由 1 变为 2。

LR 型调整的一般形式:

- ① 将 B 的右孩子 C 提升为新的根结点;
- ② 将原来的根结点 A 降为 C 的右孩子;

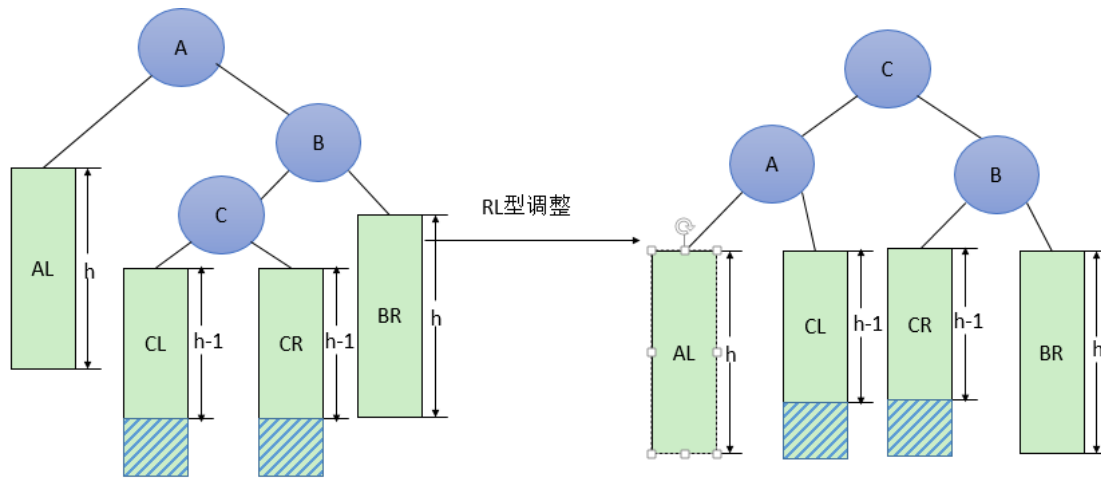


RL 型调整 (先右后左双旋转)

由于在 A 的右孩子(R)的左子树(L)上插入新结点, 使原来平衡二叉树变得不平衡, 此时 A 的平衡因子由-1 变为-2。

RL 型调整的一般形式:

- ① 将 B 的左孩子 C 提升为新的根结点;
- ② ②将原来的根结点 A 降为 C 的左孩子;
- ③ ③各子树按大小关系连接(AL 和 BR 不变, CL 和 CR 分别调整为 A 的右子树和 B 的左子树)。



七、堆

1.堆的定义

堆(heap), 这里所说的堆是数据结构中的堆, 而不是内存模型中的堆。堆通常是一个可以被看做一棵树, 它满足下列性质:

- 1) 堆中任意节点的值总是不大于(不小于)其子节点的值;
- 2) 堆总是一棵完全树。

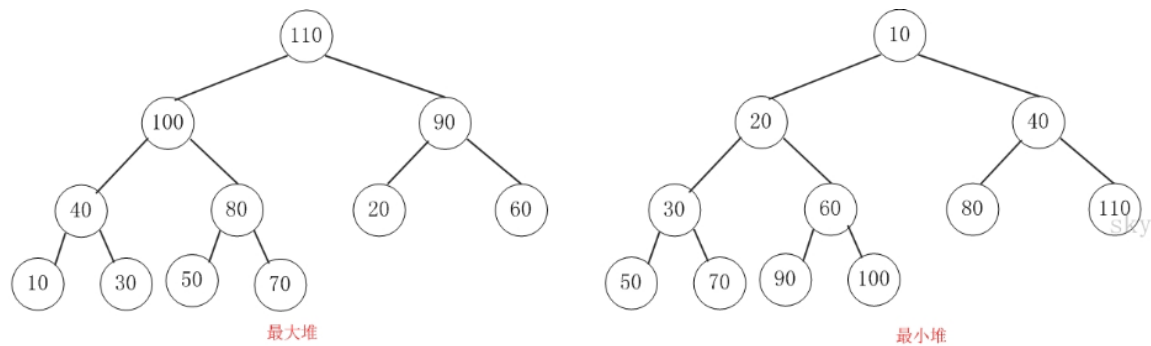
将任意节点不大于其子节点的堆叫做**最小堆或小根堆**, 而将任意节点不小于其子节点的堆叫做**最大堆或大根堆**。常见的堆有二叉堆、左倾堆、斜堆、二项堆、斐波那契堆等等。

2.二叉堆的定义

二叉堆是完全二元树或者是近似完全二元树, 它分为两种: **最大堆和最小堆**。

- 最大堆: 父节点的键值总是大于或等于任何一个子节点的键值;

- 最小堆：父结点的键值总是小于或等于任何一个子节点的键值。示意图如下：



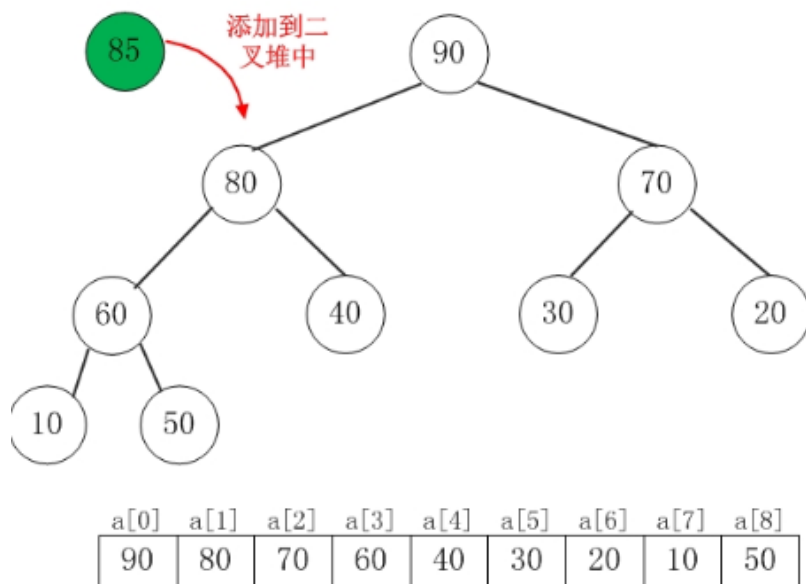
3. 二叉堆的基本操作

3.1 结点插入

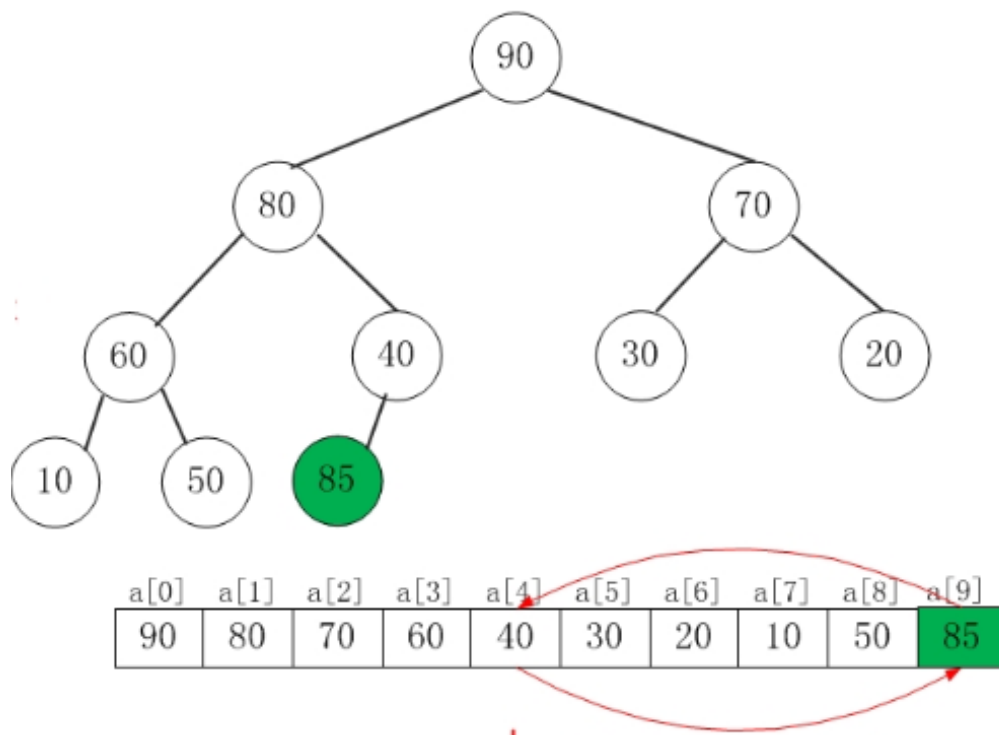
二叉堆的插入步骤

1. 首先把需要插入的元素放到二叉树的最末端（也就是数组中所有元素的最后）
2. 从最末端去向上调整，具体来说就是不断将该节点和其父节点比较，如果父节点的值比插入节点的值小（对最小堆来说，父节点值比插入节点值大），就将父节点移动到当前位置，然后继续向上比较。否则到第 3 步。
3. 如果当前父节点值比插入节点值大（对最小堆来说，插入点值比插入节点值小），就说明当前位置适合放入该插入的节点。那么将当前节点的值替换为插入节点的值，即完成插入操作。

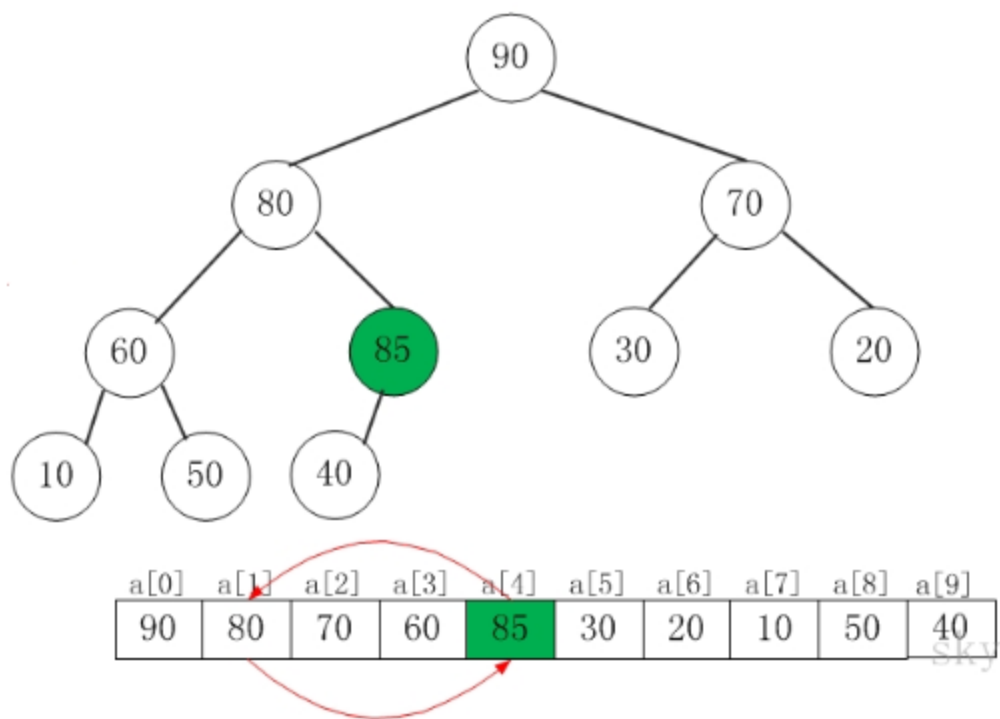
假设在最大堆[90,80,70,60,40,30,20,10,50]种添加 85，需要执行的步骤如下：



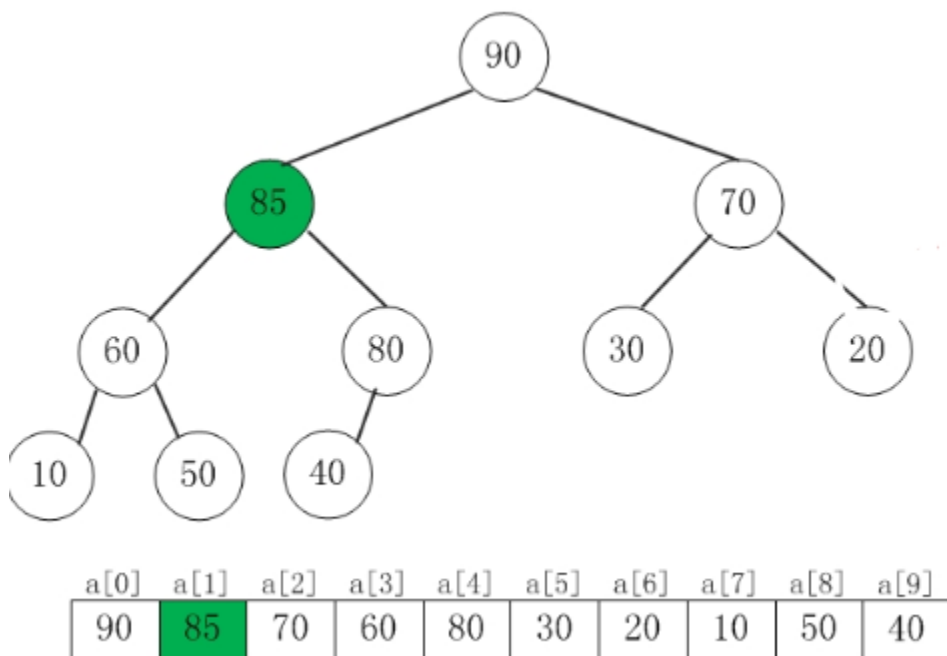
第 1 步，将 85 添加到末尾



第 2 步：85 大于父节点 40，将它和父节点交换；



第 3 步：85 大于父节点 80，将它和父节点交换；



当向最大堆中添加数据时：先将数据加入到最大堆的最后，然后尽可能把这个元素往上挪，直到挪不动为止！

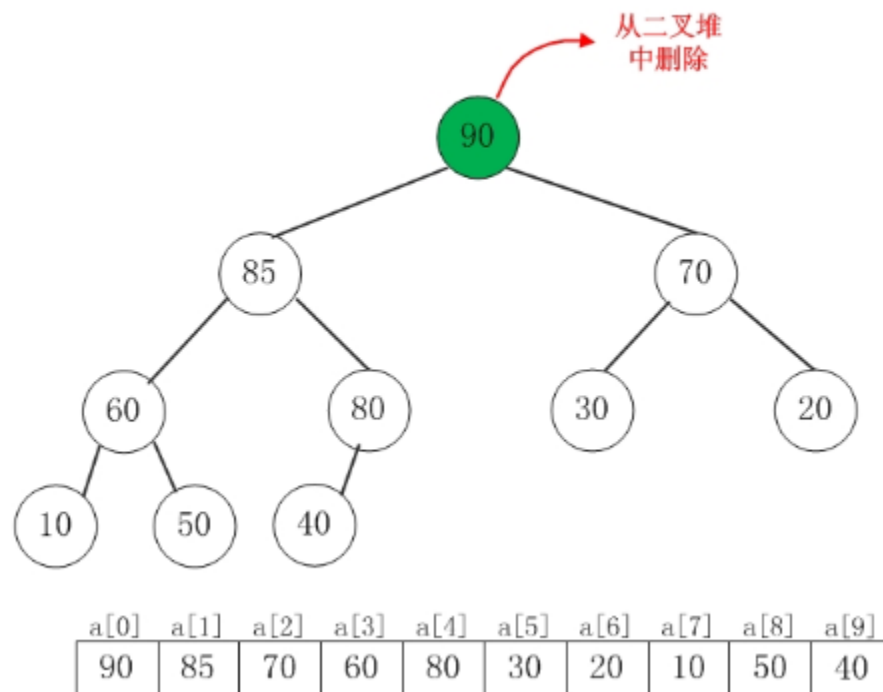
将 85 添加到[90,80,70,60,40,30,20,10,50]中后，最大堆 变成[90,85,70,60,80,30,20,10,50,40]。

3.2 结点删除

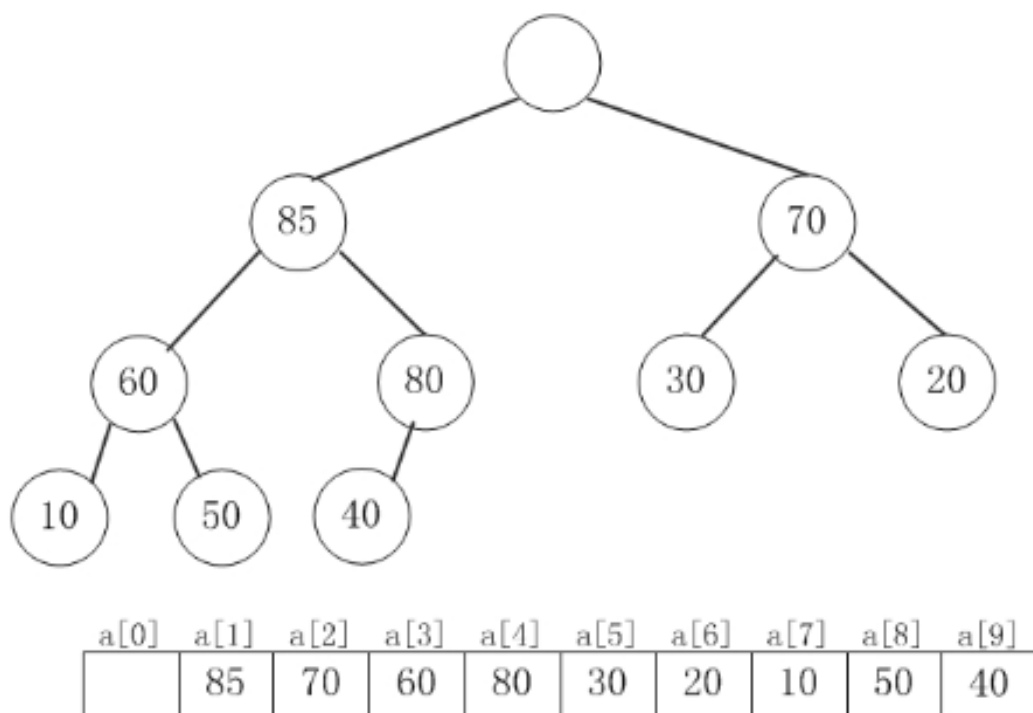
二叉堆的删除步骤

1. 首先查找到需要删除的元素的在数组中对应的下标。
2. 将该下标对应的元素换成二叉树最末端的元素（即数组中最后一个元素）。
3. 从当前下标的位置向下去调整，具体来说，首先计算出当前下标的左孩子的下标，如果当前节点有右孩子，那么比较左孩子和右孩子谁更大。将插入节点的值与左孩子和右孩子中最大的元素相比较，如果插入节点的值更小（对最小堆来说，插入节点值比当前孩子节点值大），那么将孩子节点的值放至其父亲节点，更新当前下标为替换的孩子节点下标，然后继续向下调整。否则到第 4 步。
4. 将插入节点与左孩子和右孩子中最大的元素相比较，如果插入节点的值更大（对最小堆来说，插入节点值比当前节点值大），说明插入节点应该放到当前位置。放入之后，即完成删除操作。

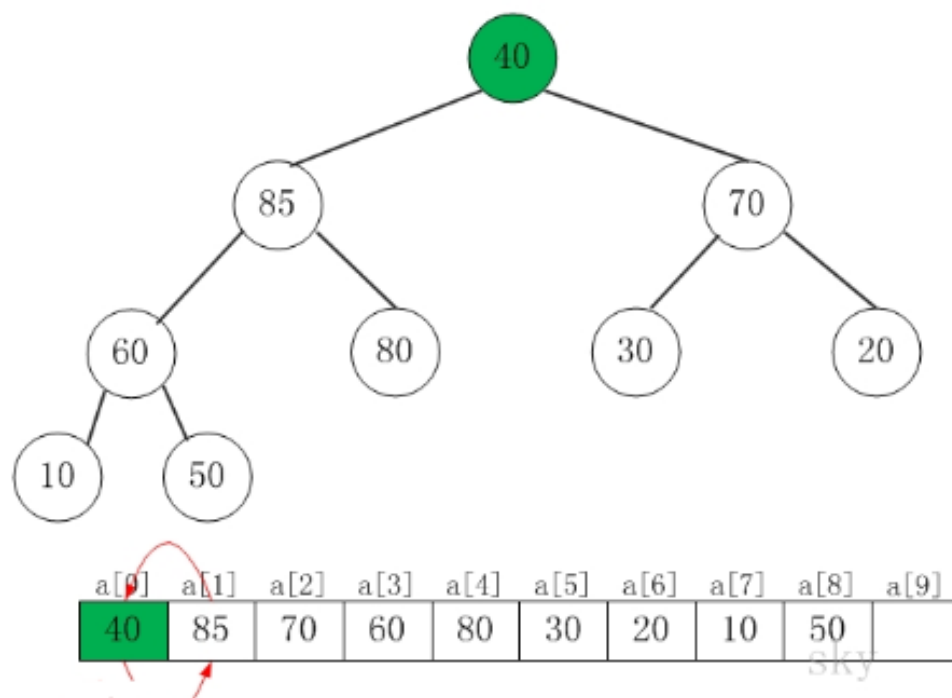
假设从最大堆[90,85,70,60,80,30,20,10,50,40]中删除 90，需要执行的步骤如下：



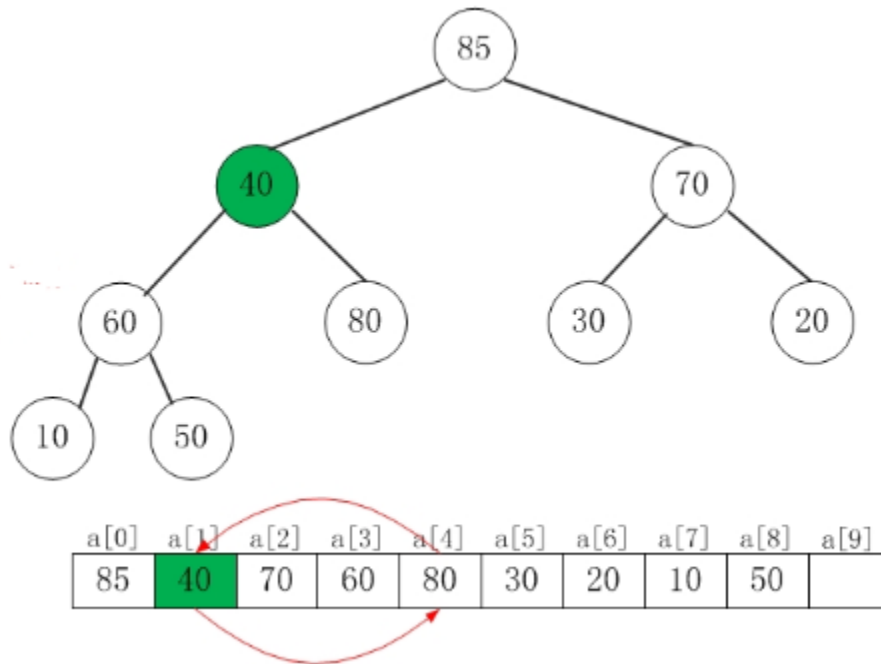
第 1 步：将 90 删除；



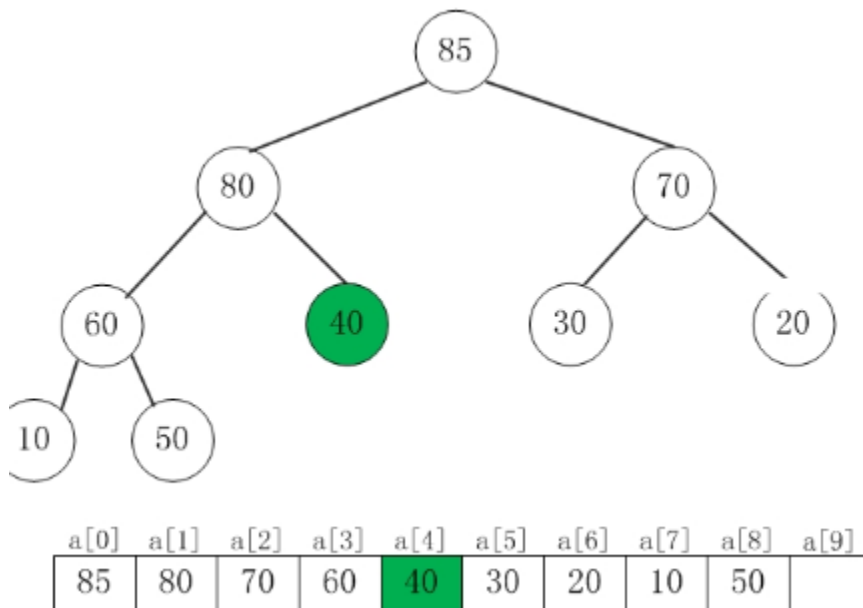
第 2 步：用末尾元素替换被删除的数据



第 3 步：40 小于子节点，选取较大的子节点和它交换；



第 4 步：40 小于子节点，选取较大的子节点和它交换；



从[90,85,70,60,80,30,20,10,50,40]删除 90 之后，最大堆变成了 [85,80,70,60,40,30,20,10,50]。

如上图所示，当从最大堆中删除数据时：先删除该数据，然后用最大堆中最后一个的元素插入这个空位；接着，把这个“空位”尽量往上挪，直到剩余的数据变成一个最大堆。

八、图、搜索算法（广度优先、深度优先）

1.概念、定义

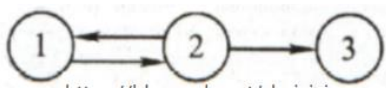
1.1 图的种类

图（Graph）是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为： $G(V, E)$ ，其中， G 表示一个图， V 是图中 G 中顶点的集合， E 是图 G 中边的集合。

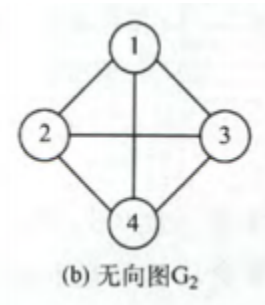
图有两种：

- 无向图
- 有向图

有向图：若 E 是有向边（简称弧）的有限集合时，则 G 为有向图。弧是顶点的有序对，记为 $\langle v, w \rangle$ ，其中 v, w 是顶点， v 是弧尾， w 是弧头。称为从顶点 v 到顶点 w 的弧。



无向图：若 E 是无向边（简称边）的有限集合时，则 G 为无向图。边是顶点的无序对，记为 (v, w) 或 (w, v) ，且有 $(v, w) = (w, v)$ 。其中 v, w 是顶点。



1.2 简单图

简单图 满足以下两条内容：

- 1) 不存在重复边
- 2) 不存在顶点到自身的边

所以上面的有向图和无向图都是简单图。与简单图 相对的是多重图，即：两个结点直接边数多于一条，又允许顶点通过同一条边与自己关联。

1.3 完全图

无向图中任意两点之间都存在边，称为**无向完全图**；如无向图中的示例就是完全图。

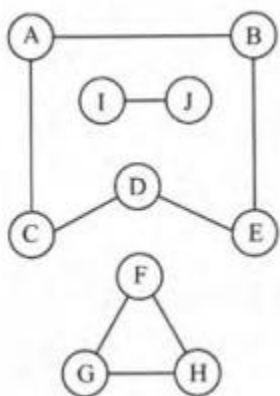
有向图中任意两点之间都存在方向相反的两条弧，称为**有向完全图**。



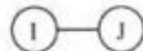
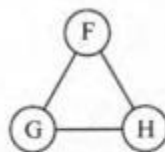
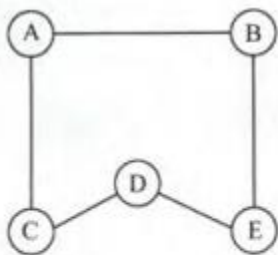
(c) 有向完全图 G_3

1.4 连通、连通图、连通分量

在无向图中，两顶点有路径存在，就称为**连通**的。若图中任意两顶点都连通，同此图为**连通图**。无向图中的极大连通子图称为**连通分量**。



(a) 无向图 G_4

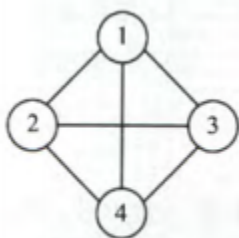


(b) G_4 的三个连通分量

1.5 生成树和生成森林

连通图的生成树是包含图中全部顶点的一个极小连通子图，若图中有 n 个顶点，则生成树有 $n-1$ 条边。所以对于生成树而言，若砍去一条边，就会变成非连通图。

在非连通图中，连通分量的生成树构成了非连通图的生成森林。



(b) 无向图 G_2

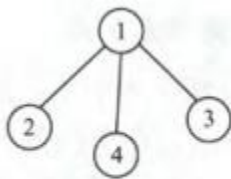


图 5-4 图 G_2 的一个生成树

1.6 顶点的度、入度和出度

顶点的度为以该顶点为一个端点的边的数目。

对于无向图，顶点的边数为度，度数之和是顶点边数的两倍。

对于有向图，入度是以顶点为终点，出度相反。有向图的全部顶点入度之和等于出度之和且等于边数。顶点的度等于入度与出度之和。

注意：入度与出度是针对有向图来说的。

1.7 边的权和网

图中每条边上标有某种含义的数值，该数值称为该边的权值。这种图称为带权图，也称作网。

1.8 路径、路径长度和回路

两顶点之间的路径指顶点之间经过的顶点序列，经过路径上边的数目称为路径长度。若有 n 个顶点，且边数大于 $n-1$ ，此图一定有环。

1.9 简单路径、简单回路

顶点不重复出现的路径称为简单路径。

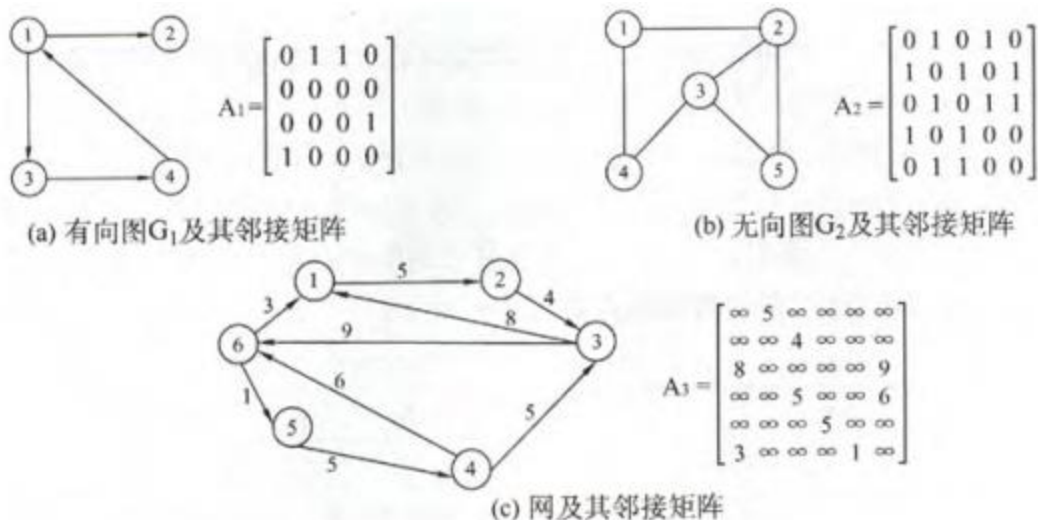
除第一个顶点和最后一个顶点外，其余顶点不重复出现的回路称为简单回路。

2. 图的存储结构

图的存储结构除了要存储图中的各个顶点本身的信息之外，还要存储顶点与顶点之间的关系，因此，图的结构也比较复杂。常用的图的存储结构有邻接矩阵和邻接表等。

1.1 邻接矩阵

图的邻接矩阵（Adjacency Matrix）存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组（称为邻接矩阵）存储图中的边或弧的信息。



图的邻接矩阵存储结构定义：

```
#define MaxVertexNum 100 //顶点数目的最大值
typedef char VertexType; //顶点的数据类型
typedef int EdgeType; //带权图中边上权值的数据类型
typedef struct{
    VertexType Vex[MaxVertexNum]; //顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum]; //邻接矩阵，边表
    int vexnum, arcnum; //图的当前顶点数和弧数
}MGraph;
```

1.2 邻接表

邻接表由**表头节点**和**表节点**两部分组成，图中每个顶点均对应一个存储在数组中的表头节点。如果这个表头节点所对应的顶点存在邻接节点，则把邻接节点依次存放于表头节点所指向的单向链表中。

首先要明确，一个图包含了一个顶点表，而顶点表里的每一项又包含一个边表。

顶点表：该表包含了图里的所有顶点，顶点表的每一项用于存储该顶点的属性（例如该结点对应的值），以及指向其边表的第一个结点的指针。

边表：某个顶点的边表存放了与其相邻的结点。

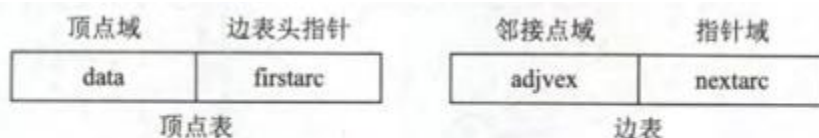
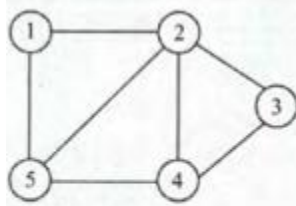
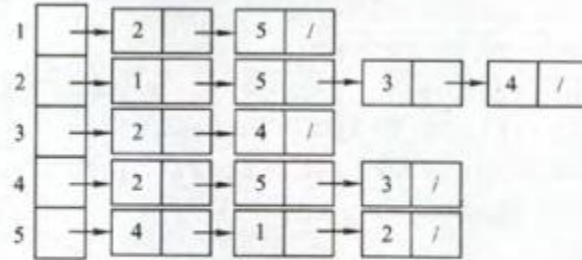


图 5-6 顶点表和边表结点结构

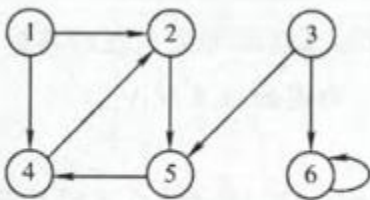


(a) 无向图G

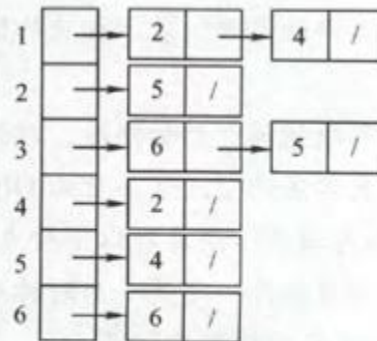


(b) 图G的邻接表表示

图 5-7 无向图邻接表法实例



(a) 有向图G



(b) 有向图G的邻接表表示

图 5-8 有向图邻接表法实例

```
#define MaxVertexNum 100 //图中顶点数目的最大值
typedef struct ArcNode{ //边表结点
    int adjvex; //该弧所指向的顶点的位置
    struct ArcNode *next; //指向下一条弧的指针
    //InfoType info; //网的边权值
}ArcNode;
typedef struct VNode{ //顶点表结点
    VertexType data; //顶点信息
    ArcNode *first; //指向第一条依附该顶点的弧的指针
}VNode, AdjList[MaxVertexNum];
typedef struct{
    AdjList vertices; //邻接表
    int vexnum, arcnum; //图的顶点数和弧数
} ALGraph; //ALGraph 是以邻接表存储的图类型
```

3.图的遍历 (BFS、DFS)

图的遍历和树的遍历类似，希望从图中某一顶点出发访遍图中其余顶点，且使每一个顶点仅被访问一次，这一过程就叫图的遍历。

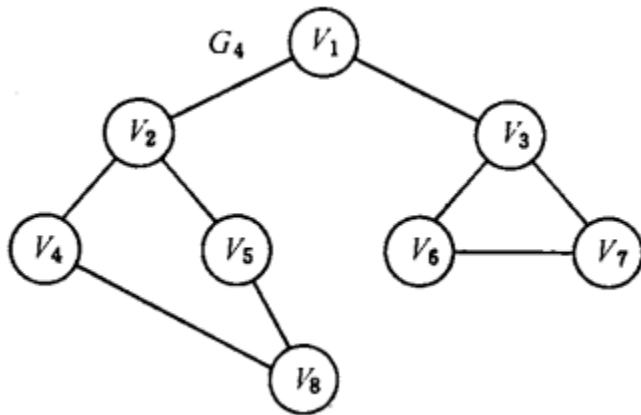
对于图的遍历来说，如何避免因回路陷入死循环，就需要科学地设计遍历方案，通过有两种遍历次序方案：**广度优先搜索**和**深度优先搜索**。

广度优先搜索

广度优先搜索类似于二叉树的层序遍历算法，它的基本思想是：利用队列的结构，先从开始节点的邻居开始遍历，先检索一个节点是否满足要求，若满足要求，则结束搜索，若不满足则将该节点弹出队列，将该节点的邻居加入队列，最终完成遍历或找到满足要求的节点。

广度优先搜索在搜索访问一层时，需要记住已被访问的顶点，以便在访问下层顶点时，从已被访问的顶点出发搜索访问其邻接点。所以在广度优先搜索中需要设置一个队列 Queue，使已被访问的顶点顺序由队尾进入队列。在搜索访问下层顶点时，先从队首取出一个已被访问的上层顶点，再从该顶点出发搜索访问它的各个邻接点。

Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和广度优先搜索类似的思想。



其访问序列为：

$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8$

```
int visited[Vertex_Num];
LinkQueue Q;

//广度优先
void BFSTraverse(Graph G) {
```

```

    for (int i = 1; i <= G.vexnum; ++i) {
        visited[i] = FALSE;
    }

    InitQueue(Q);

    for (int i = 1; i <= G.vexnum; ++i) {
        if (!visited[i]) {
            BFS(G,i);
        }
    }
    DestroyQueue(Q);
}

void BFS(Graph G, int v) {
    visit(v);
    visited[v] = TRUE;
    EnQueue(Q, v);
    while (!IsEmpty(Q)) {
        VertexType u;
        DeQueue(Q, u);
        ArcNode *e = G.vertices[u].first;
        while (e) {
            VertexType w = e->adjvex;
            if (!visited[w]) {
                visit(w);
                visited[w] = TRUE;
                EnQueue(Q, w);
            }
            e = e->next;
        }
    }
}

```

深度优先搜索

深度优先搜索类似于树的先序遍历；它的思想是从一个顶点 V_0 开始，沿着一条路一直走到底，如果发现不能到达目标解，那就返回到上一个节点，然后从另一条路开始走到底，这种尽量往深处走的概念即是深度优先的概念。

还是上图结构，其访问序列为：

$$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow v_3 \rightarrow v_6 \rightarrow v_7$$

```

//深度优先
void DFSTraverse(Graph G) {

```

```

    for (int i = 1; i <= G.vexnum; ++i) {
        visited[i] = FALSE;
    }

    for (int j = 1; j <= G.vexnum; ++j) {
        if (!visited[j]) {
            DFS(G, j);
        }
    }
}

void DFS(Graph G, int v) {
    visit(v);
    visited[v] = TRUE;

    ArcNode *e = G.vertices[v].first;
    while (e) {
        VertexType w = e->adjvex;
        if (!visited[w]) {
            DFS(G, w);
        }
        e = e->next;
    }
}

```

测试用例 (demo_6_1_graph)

使用上面的图存储结构，分别进行 BFS 和 DFS，结果如下：

```

int main() {
    Graph G;
    CreateGraph(&G);

    printf("\ninit graph:\n");
    for (int i = 1; i <= G.vexnum; i++)
    {
        printf("|v%d|->", G.vertices[i].data);
        ArcNode *e = G.vertices[i].first;
        while (e != NULL)
        {
            printf("%d->", e->adjvex);
            e = e->next;
        }
        printf("null\n");
    }

    printf("\nBFS result:\n");
    BFSTraverse(G);
    printf("null\n");

    printf("DFS result:\n");
}

```



```

    DFSTraverse(G);
    printf("null\n");
    return 0;
}

```

```

$ ./result.exe

init graph:
|v1|->2->3->null
|v2|->4->5->null
|v3|->6->7->null
|v4|->8->null
|v5|->8->null
|v6|->7->null
|v7|->null
|v8|->null

BFS result:
1-> 2-> 3-> 4-> 5-> 6-> 7-> 8->null
DFS result:
1-> 2-> 4-> 8-> 5-> 3-> 6-> 7->null

```

4. 图的应用

4.1 最小生成树

普利姆 (prim) 算法

Prim 时间复杂度为： $O(|V|^2)$ ，不依赖 $|E|$ ，它适用于求解边稠密的图的最小生成树。

克鲁斯卡尔算法：

时间复杂度为： $O(|E|\log|E|)$ ，克鲁斯卡尔算法适用于求解边疏密而顶点较多的图的最小生成树。

4.2 最短路径

Dijkstra 算法

求解的是单元最短路径问题，即求带权有向图中某个源点到其余各顶点的最短路径；

Floyd 算法

求解的是所有顶点之间的最短路径，即求出 v_i 和 v_j 之间的最短路径和最短路径长度；

4.3 拓扑排序

若在带权的有向图中，以顶点表示事件，以边（或者弧）表示活动，弧的权值表示活动的开销，则此带权有向图称为用边表示活动的网，简称：(AOV 网 (Activity On Vertex Network))。

拓扑排序是一个比较常用的图论算法，经常用于完成有依赖关系的任务的排序。

对于任何有向图而言，其拓扑排序为其所有结点的一个线性排序（对于同一个有向图而言可能存在多个这样的结点排序）。该排序满足这样的条件——对于图中的任意两个结点 u 和 v ，若存在一条有向边从 u 指向 v ，则在拓扑排序中 u 一定出现在 v 前面。

一个无环的有向图称为有向无环图，在 AOV 网中不存在有向环。

4.4 关键路径

AOE 网：在一个表示工程的带权有向图中，用顶点表示事件（如 V_0 ），用有向边表示活动（如 $\langle v_0, v_1 \rangle = a_1$ ），边上的权值表示活动的持续时间，称这样的有向图为边表示的活动的网，简称 AOE 网 (activity on edge network)

源点：在 AOE 网中，没有入边的顶点称为源点；如顶点 V_0

终点：在 AOE 网中，没有出边的顶点称为终点；如顶点 V_3

AOE 网的性质：

- 1) 只有在进入某顶点的活动都已经结束，该顶点所代表的事件才发生；例如， a_1 和 a_2 活动都结束了，顶点 V_2 所代表的事件才会发生。
- 2) 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才开始；例如，只有顶点 V_1 所代表的事件结束之后，活动 a_2 和 a_4 才会开始。

在 AOE 网中，所有活动都完成才能到达终点，因此完成整个工程所必须花费的时间（即最短工期）应该为源点到终点的最大路径长度。具有最大路径长度的路径称为**关键路径**。关键路径上的活动称为**关键活动**。

九、排序算法

常用排序算法有：

- **插入排序**
 - 直接插入排序

- 希尔排序
- 交换排序
 - 冒泡排序
 - 快速排序
- 选择排序
 - 简单选择排序
 - 堆排序
- 归并排序
- 基数排序

1.插入排序

1.1 概念

基本思想：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中，直到全部记录插入完成。

时间复杂度： $O(n^2)$

空间复杂度： $O(1)$

稳定性： 稳定

适用于顺序存储和链式存储的线性表；

1.2 代码

```
void InsertSort(int a[], int n)
{
    for (int i = 1; i < n; i++) {
        if (a[i] < a[i - 1]) {
            int j = i - 1;
            int x = a[i];
            while (j > -1 && x < a[j]) {
                a[j + 1] = a[j];
                j--;
            }
            a[j + 1] = x;
        }
    }
}
```

```

    }
    print(a, n, i); //打印每次排序后的结果
}
}

```

1.3 示例(demo_7_1_insert_sorting.cpp)

直接插入排序过程如下：

初始序列：	4, 5, 1, 2, 6, 3	
第一趟：	4, 5, 1, 2, 6, 3	(将 5 插入到{4})
第二趟：	1, 4, 5, 2, 6, 3	(将 1 插入到{4, 5})
第三趟：	1, 2, 4, 5, 6, 3	(将 2 插入到{1, 4, 5})
第四趟：	1, 2, 4, 5, 6, 3	(将 6 插入到{1, 2, 4, 5})
第五趟：	1, 2, 3, 4, 5, 6	(将 3 插入到{1, 2, 4, 5, 6})

测试用例

```

int main() {
    int arr[6] = { 4, 5, 1, 2, 6, 3 };
    int sLen = sizeof(arr) / sizeof(ElementType);

    printf("init arr:\n");
    for (int j = 0; j < sLen; j++) {
        printf("%d ", arr[j]);
    }
    printf("\n\nresult:\n");
    InsertSort(arr, sLen);
    return 0;
}

```

```

$ ./result.exe
init arr:
4 5 1 2 6 3

result:
1: 4 5 1 2 6 3
2: 1 4 5 2 6 3
3: 1 2 4 5 6 3
4: 1 2 4 5 6 3
5: 1 2 3 4 5 6

```

2. 希尔排序

2.1 概念

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法。

插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率；

希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

时间复杂度：当 n 在特定范围时，约为 $O(n^{1.3})$ ，在最坏的情况下为 $O(n^2)$ ；

空间复杂度： $O(1)$ ；

稳定性：不稳定

适用性：希尔排序算法仅适用于当线性表尾顺序存储的情况；

2.2 代码

```
void ShellSort(int a[], int n)
{
    int dk;
    int i, j;
    int temp;
    for (dk = n / 2; dk > 0; dk /= 2) //用来控制步长,最后递减到1
    {
        for (i = dk; i < n; i++)
        {
            temp = a[i];
            for (j = i - dk; j >= 0 && temp < a[j]; j -= dk)
            {
                a[j + dk] = a[j];
            }
            a[j + dk] = temp;
        }
        print(a, n, dk); //打印每次排序后的结果
    }
}
```

2.3 示例(demo_7_2_shell_sort.cpp)

```
int main()
{
    int a[] = { 5, 18, 11, 38, 160, 363, 174, 19, 793, 100 };
    int sLen = sizeof(a) / sizeof(int);
```

```

printf("init arr:\n");
for (int i = 0; i < sLen; i++) {
    printf("%d ", a[i]);
}

ShellSort(a, 10);

printf("\nshell sort result: \n");
for (int j = 0; j < sLen; j++)
{
    printf("%d ", a[j]);
}
printf("\n");
return 0;
}

```

```

init arr:
5 18 11 38 160 363 174 19 793 100
dk = 5: 5 18 11 38 100 363 174 19 793 160
dk = 2: 5 18 11 19 100 38 174 160 793 363
dk = 1: 5 11 18 19 38 100 160 174 363 793
shell sort result:
5 11 18 19 38 100 160 174 363 793

```

3.冒泡排序

3.1 概念

冒泡排序是一种[交换排序](#)。

算法思想：重复地走访过要排序的元素列，依次比较两个相邻的元素，如果他们的顺序（如从大到小、首字母从 A 到 Z）错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列 已经排序完成。

冒泡排序中所产生的有序子序列一定是全局有序的，这样每一趟排序都会将一个元素放置到最终的位置上。

时间复杂度：在算法中加入 flag 标志位，最好情况下，当数组顺序有序时，第一遍遍历后，不再进行比较， $O(n)$,平均时间复杂度 $O(n^2)$;

空间复杂度： $O(1)$

稳定性：稳定

3.2 代码

```
void BubbleSort(int a[], int n) {
    for (int i = 0; i < n; i++) {
        int flag = FALSE;
        for (int j = n - 1; j > i; j--) {
            if (a[j-1] > a[j]) { //swap
                int temp;
                temp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = temp;
                flag = true;
            }
        }

        print(a, n, i+1); //用于打印每次排序后的序列

        if (flag = FALSE) {
            return;
        }
    }
}
```

3.3 示例(demo_7_3_bubble_sort. cpp)

```
int main()
{
    int a[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    int sLen = sizeof(a) / sizeof(int);

    printf("init arr:\n");
    for (int i = 0; i < sLen; i++) {
        printf("%d ", a[i]);
    }

    BubbleSort(a, sLen);

    printf("\nbubble sort result: \n");
    for (int j = 0; j < sLen; j++)
    {
        printf("%d ", a[j]);
    }
    printf("\n");
    return 0;
}
```

```

$ ./result.exe
init arr:
10 9 8 7 6 5 4 3 2 1
| 1 |: 1 10 9 8 7 6 5 4 3 2
| 2 |: 1 2 10 9 8 7 6 5 4 3
| 3 |: 1 2 3 10 9 8 7 6 5 4
| 4 |: 1 2 3 4 10 9 8 7 6 5
| 5 |: 1 2 3 4 5 10 9 8 7 6
| 6 |: 1 2 3 4 5 6 10 9 8 7
| 7 |: 1 2 3 4 5 6 7 10 9 8
| 8 |: 1 2 3 4 5 6 7 8 10 9
| 9 |: 1 2 3 4 5 6 7 8 9 10
|10 |: 1 2 3 4 5 6 7 8 9 10
bubble sort result:
1 2 3 4 5 6 7 8 9 10

```

4. 快速排序

4.1 概念

快速排序是一种分治的排序算法，通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

快速排序中，每一趟排序后都将一个元素放到最终位置上。

快速排序的效率和划分是否对称相关，在最理想的状态下，在做到平衡划分的情况下，效率大大提高；**快速排序是所有内部排序算法中平均性能最优的排序算法。**

时间复杂度：平均情况下 $O(n * \log n)$ ，最坏情况下 $O(n^2)$ ；

空间复杂度： $O(n * \log n)$

稳定性：不稳定

4.2 代码

```

void QuickSort(int a[], int low, int high) {
    if (low < high) {
        int pivotPos = Partition(a, low, high);
        QuickSort(a, low, pivotPos-1);
    }
}

```



```

        QuickSort(a, pivotPos+1, high);
    }
}

int Partition(int a[], int low, int high) {
    int pivot = a[low];
    int tmp1 = low;    //变量仅用于打印
    int tmp2 = high;   //变量仅用于打印

    while (low < high) {
        while (low < high && a[high] >= pivot) {
            high--;
        }
        a[low] = a[high];

        while (low < high && a[low] <= pivot) {
            low++;
        }
        a[high] = a[low];
    }

    a[low] = pivot;
    print(a, tmp1, tmp2, pivot);
    return low;
}

```

4.3 示例(demo_7_4_quick_sort.cpp)

```

int main()
{
    int a[] = { 56, 39, 28, 47, 16, 75, 34, 44, 21, 99 };
    int sLen = sizeof(a) / sizeof(int);

    printf("init arr:\n");
    for (int i = 0; i < sLen; i++) {
        printf("%d ", a[i]);
    }

    QuickSort(a, 0, sLen-1);

    printf("\nquick sort result: \n");
    for (int j = 0; j < sLen; j++)
    {
        printf("%d ", a[j]);
    }
    printf("\n");
    return 0;
}

```

```

$ ./result.exe
init arr:
56 39 28 47 16 75 34 44 21 99
from 0 to 9, pivot = 56 | 21 39 28 47 16 44 34 56 75 99
from 0 to 6, pivot = 21 | 16 21 28 47 39 44 34
from 2 to 6, pivot = 28 | 28 47 39 44 34
from 3 to 6, pivot = 47 | 34 39 44 47
from 3 to 5, pivot = 34 | 34 39 44
from 4 to 5, pivot = 39 | 39 44
from 8 to 9, pivot = 75 | 75 99
quick sort result:
16 21 28 34 39 44 47 56 75 99

```

5. 简单选择排序

5.1 概念

算法思想：选择排序，从头至尾扫描序列，找出最小的一个元素，和第一个元素交换，接着从剩下的元素中继续这种选择和交换方式，最终得到一个有序序列。

简单选择排序的每一趟排序可以确定一个元素的最终位置。

时间复杂度： $O(n^2)$

空间复杂度： $O(1)$

稳定性：不稳定

5.2 代码

```

void SelectSort(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        if (min != i) {
            int tmp = a[min];
            a[min] = a[i];
            a[i] = tmp;
        }
    }
}

```

```

        print(a, n, i, min);
    }
}

```

5.3 示例(demo_7_5_select_sort. cpp)

```

int main()
{
    int a[] = { 56, 39, 28, 47, 16, 75, 34, 44, 21, 99 };
    int sLen = sizeof(a) / sizeof(int);

    printf("init arr:\n");
    for (int i = 0; i < sLen; i++) {
        printf("%d ", a[i]);
    }

    SelectSort(a, sLen);

    printf("\nselect sort result: \n");
    for (int j = 0; j < sLen; j++)
    {
        printf("%d ", a[j]);
    }
    printf("\n");
    return 0;
}

```

```

$ ./result.exe
init arr:
56 39 28 47 16 75 34 44 21 99

|0| compared value 56 , min value 16 , 16
|1| compared value 39 , min value 21 , 16 21
|2| compared value 28 , min value 28 , 16 21 28
|3| compared value 47 , min value 34 , 16 21 28 34
|4| compared value 56 , min value 39 , 16 21 28 34 39
|5| compared value 75 , min value 44 , 16 21 28 34 39 44
|6| compared value 47 , min value 47 , 16 21 28 34 39 44 47
|7| compared value 75 , min value 56 , 16 21 28 34 39 44 47 56
|8| compared value 75 , min value 75 , 16 21 28 34 39 44 47 56 75
select sort result:
16 21 28 34 39 44 47 56 75 99

```

6. 堆排序

5.1 概念

堆排序的基本思想：

- 1) 将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。
- 2) 将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余 $n-1$ 个元素重新构成一个堆，这样会得到 n 个元素的次小值。
- 3) 如此反复执行，便能得到一个有序序列。

时间复杂度： $O(n * \log n)$

空间复杂度： $O(1)$

稳定性：不稳定

5.2 代码

//向下调整构造初始堆

```
void HeapAdjust(int a[], int k, int n)
{
    int tmp = a[k];

    for (int i = 2 * k + 1; i <= n; i = i * 2 + 1) {
        if (i < (n-1) && a[i] < a[i + 1]) {
            i++;
        }

        if (tmp > a[i]) {
            break;
        }
        else {
            a[k] = a[i];
            k = i;
        }
    }
    a[k] = tmp;
}

void HeapSort(int a[], int n)
{
    int temp;
    //通过循环初始化顶堆
    for (int i = n / 2 - 1; i >= 0; i--) {
```

```

        HeapAdjust(a, i, n-1);
    }

    print(a, n-1, n);
    for (int i = n-1; i > 0; i--)
    {
        temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        //重新调整为顶堆
        print2(a, i, n);
        HeapAdjust(a, 0, i-1);
        print(a, i-1, n);
    }
}

```

5.3 示例(demo_7_6_heap_sort)

```

int main()
{
    int a[] = { 1, 56, 39, 28, 47, 16 };
    int sLen = sizeof(a) / sizeof(int);

    printf("init arr:   len = %d\n", sLen);
    for (int i = 0; i < sLen; i++) {
        printf("%d ", a[i]);
    }

    HeapSort(a, sLen);

    printf("\nheap sort result: \n");

    for (int j = 0; j < sLen; j++)
    {
        printf("%d ", a[j]);
    }
}

```

```

$ ./result.exe
init arr:   len = 6
1 56 39 28 47 16
max heap 0-5: 56 47 39 28 1 16
sort        16 47 39 28 1 56

max heap 0-4: 47 28 39 16 1    56
sort        1 28 39 16 47    56

max heap 0-3: 39 28 1 16    47 56
sort        16 28 1 39    47 56

max heap 0-2: 28 16 1    39 47 56
sort        1 16 28    39 47 56

max heap 0-1: 16 1    28 39 47 56
sort        1 16    28 39 47 56

max heap 0-0: 1    16 28 39 47 56
heap sort result:
1 16 28 39 47 56

```

7. 归并排序

6.1 概念

归并排序是利用递归与分治的技术将数据序列划分为越来越小的半子表，再对半子表排序，最后再用递归方法将排好序的半子表合并成越来越大的有序序列。

算法核心： 第一步:划分半子表； 第二步：合并半子表。

时间复杂度： $O(n * \log n)$

空间复杂度： 操作中需要 n 个单元的辅助控件，所以空间复杂度为： $O(n)$

稳定性： 稳定

6.2 代码

```
void Merge(int A[], int low, int mid, int high) {

    for (int k = low; k <= high; k++) { //A的所有元素复制到辅助数组B
        B[k] = A[k];
    }

    int i = low;
    int j = mid + 1;
    int k = i;

    while (i <= mid && j <= high) { //比较B中左右两端的元素
        if (B[i] <= B[j]) {
            A[k++] = B[i++];
        }
        else {
            A[k++] = B[j++];
        }
    }

    while (i <= mid) { //若第一个表未检测完，复制
        A[k++] = B[i++];
    }

    while (j <= high) { //若第二个表未检测完，复制
        A[k++] = B[j++];
    }
}

void MergeSort(int A[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        MergeSort(A, low, mid);
        MergeSort(A, mid+1, high);
        Merge(A, low, mid, high);
        print(A, low, high);
    }
}
```

示例(demo_7_7_merge_sort.cpp)

```
int main() {
    int a[] = { 41, 56, 39, 28, 47, 16, 22, 33 };
    int sLen = sizeof(a) / sizeof(int);
    len = sLen;

    printf("init arr:   len = %d\n", sLen);
    for (int i = 0; i < sLen; i++) {
        printf("%d ", a[i]);
    }

    B = (int *)malloc(sLen*sizeof(int));
```

```

MergeSort(a, 0, sLen-1);

printf("\nmerge sort result: \n");

for (int j = 0; j < sLen; j++)
{
    printf("%d ", a[j]);
}

return 0;
}

```

```

$ ./result.exe
init arr:   len = 8
41 56 39 28 47 16 22 33
41 56
      28 39
28 39 41 56
           16 47
                22 33
                16 22 33 47
16 22 28 33 39 41 47 56
merge sort result:
16 22 28 33 39 41 47 56

```

8. 基数排序

8.1 概念

基数排序是一种很特别的排序方法，**它不是基于比较进行排序的算法**，而是多关键字排序思想。借助‘分配’和‘收集’两种操作，

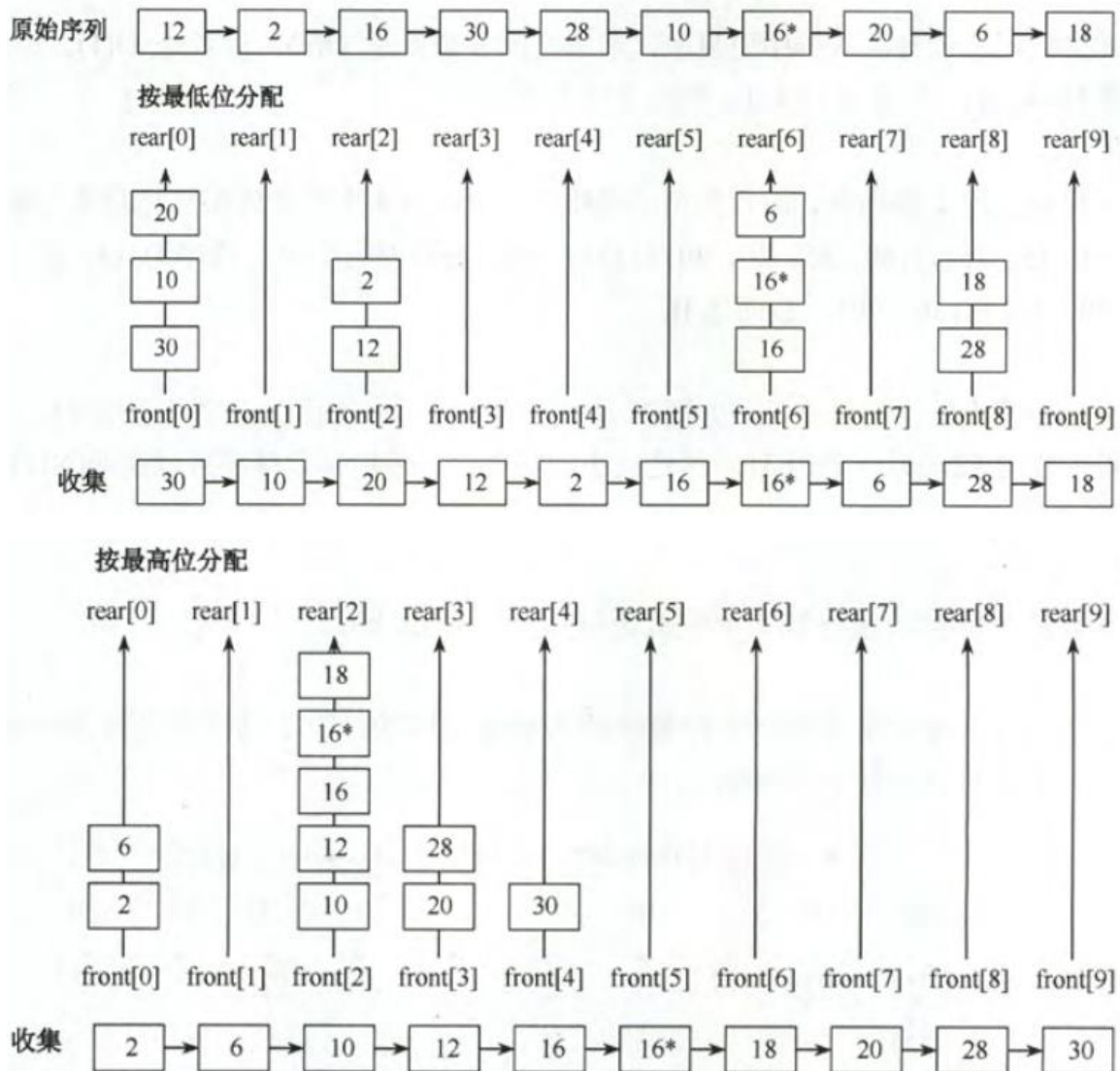
对单逻辑关键字进行排序。基数排序又分为最高位优先(MSD)排序和最低位优先(LSD)排序。

MSD：先从高位开始进行排序，对每个关键字，可采用计数排序

LSD：先从低位开始进行排序，对每个关键字，可采用桶排序

8.2 示例

如待排的排序序列为{12,2,16,30,28,10,16,20,6,18}，使用链式队列的基数排序的排序过程如下：



需要通过 2 次“分配”和“收集”完成排序。

时间复杂度： 基数排序需要进行 d 趟分配和收集，一趟分配需要 $O(n)$ ，一趟收集需要 $O(r)$ ，所以基数排序的时间复杂度为 $O(d*(n+r))$ ；

空间复杂度： 基数排序需要辅助空间为 r (r 个队列)，所以空间复杂度为 $O(r)$ ；

稳定性： 稳定

9. 各种内部排序算法的比较和应用

比较

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2-路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

应用

- 若 n 较小，可以采用[直接插入排序](#)或[简单选择排序](#)；
- 若文件的初始状态已按关键字基本有序，则选用[直接插入](#)或[冒泡排序](#)为宜。
- 若 n 较大，则应采用时间复杂度为 $O(n * \log n)$ 的排序算法：[快速排序](#)、[堆排序](#)、[归并排序](#)。
- 在基于“比较”的排序算法中，可以证明，至少需要 $O(n * \log n)$ 的时间。
- 若 n 很大，记录的关键字位数较少，且可以分解时，采用[基数排序](#)较好。

十、递归、迭代思想

递归

“递归”过程是指函数调用自身的过程，一般用于一些算法，例如求一个数的阶乘、计算斐波那契数列等。

优点： 代码量少，方便

缺点： 逻辑复杂，函数调用都有栈帧的开辟，调用次数过多会产生栈溢出异常
StackOverflowError

设计一个递归过程时，必须至少测试一个可以终止此递归的条件，并且必须对在合理的递归调用次数内不满足此类条件的情况进行处理。

使用递归算法求解斐波那契数列：

```
int fib_recursive(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return fib_recursive(n - 1) + fib_recursive(n - 2);
    }
}
```

迭代

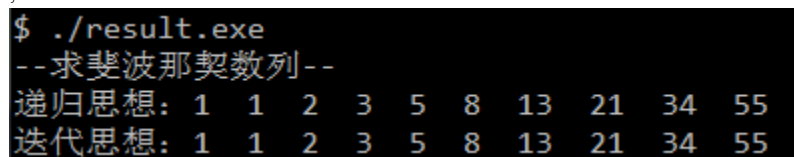
迭代是重复反馈过程的活动，其目的通常是为了逼近所需目标或结果。每一次对过程的重复称为一次“迭代”，而**每一次迭代得到的结果会作为下一次迭代的初始值**。

重复执行一系列运算步骤，从前面的量依次求出后面的量的过程。此过程的每一次结果，都是由对前一次所得结果施行相同的运算步骤得到的。

缺点： 周期长、成本很高

使用迭代思想求解斐波那契数列：

```
int fib_iter(int n) {
    int i, temp0, temp1, temp2;
    if (n == 0 || n == 1) return 1;
    temp1 = 1;
    temp2 = 1;
    for (i = 2; i <= n; i++) {
        temp0 = temp1 + temp2;
        temp2 = temp1;
        temp1 = temp0;
    }
    return temp0;
}
```



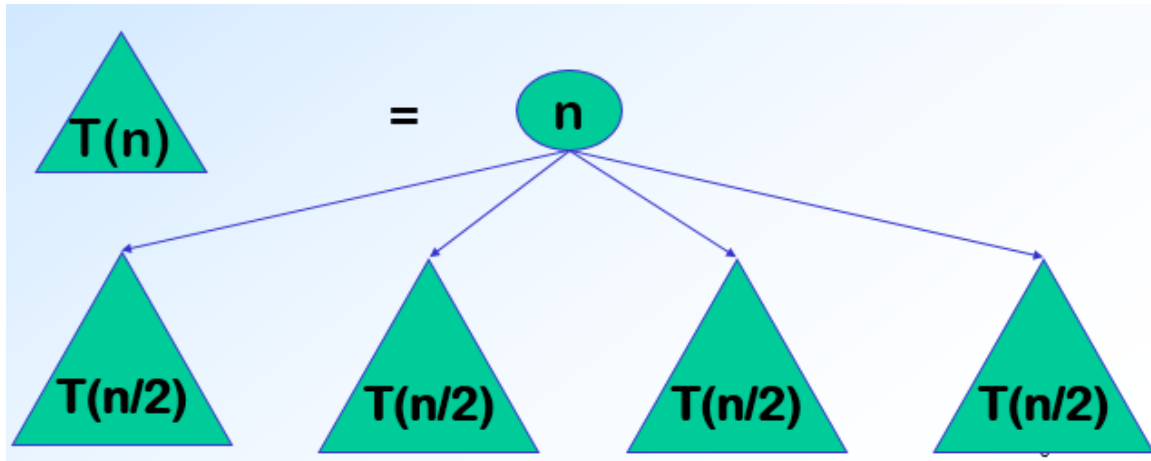
```
$ ./result.exe
--求斐波那契数列--
递归思想: 1 1 2 3 5 8 13 21 34 55
迭代思想: 1 1 2 3 5 8 13 21 34 55
```

十一、分治思想

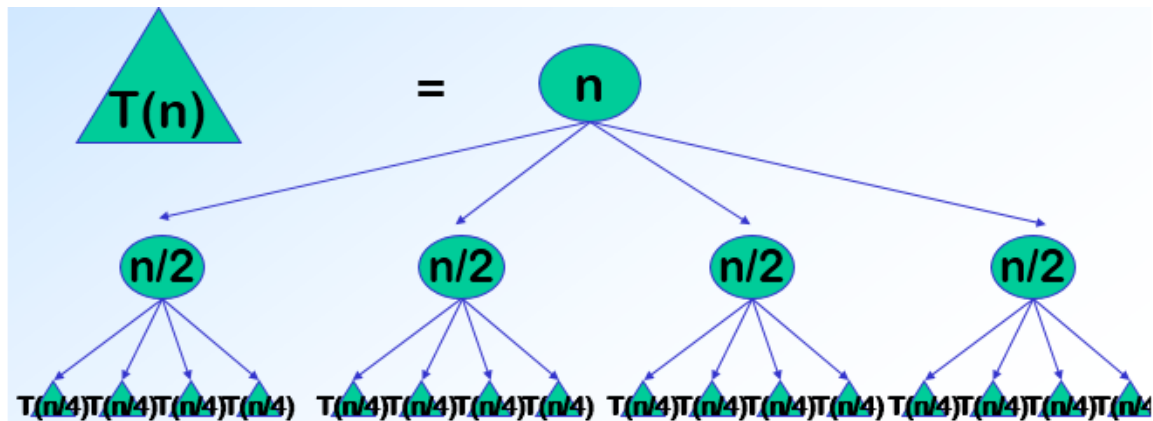
分治法的设计思想：将一个难以直接解决的大问题，分割成一些规模较小的相互独立的相同问题，以便各个击破，分而治之。

比如：

1. 将求解的较大规模的问题分割成 k 个更小规模的子问题。



2. 对这 k 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 k 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。



分治思想的应用

1. 快速排序、归并排序

算法在前面已实现，这里不再赘述。

2. 二分查找

代码

```
int BinarySearch(int a[], int n, int key) {
    int low, high, mid;
    low = 0;
    high = n - 1;

    while (low <= high) {
        print(a,n,low,high);
        mid = (low + high) / 2;
        if (key < a[mid]) {
            high = mid - 1;
        }
        else if(key > a[mid]){
            low = mid + 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}
```

```
$ ./result.exe
查找关键字 32
11 12 15 22 28 32 39 43 47 54 67 70 73 78 82 85 90 99
11 12 15 22 28 32 39 43
28 32 39 43
result: idx = 5
查找关键字 31
11 12 15 22 28 32 39 43 47 54 67 70 73 78 82 85 90 99
11 12 15 22 28 32 39 43
28 32 39 43
28
result: idx = -1
```

十二、贪心算法

1.1 基本思路

贪心算法的基本思路是：从问题的某一个初始解出发一步一步地进行，**根据某个优化测度，每一步都要确保能获得局部最优解。每一步只考虑一个数据，他的选取应该满足局部优化的条件。**若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止。

1.2 求解过程

1. 建立数学模型来描述问题；
2. 把求解的问题分成若干个子问题；
3. 对每一子问题求解，得到子问题的局部最优解；
4. 把子问题的解局部最优解合成原来求解问题的一个解。

1.3 适用场景

狭义的贪心算法指的是解最优化问题的一种特殊方法，解决过程中总是做出当下最好的选择，因为具有最优子结构的特点，**局部最优解 可以得到全局最优解；这种贪心算法是动态规划的一种特例。能用贪心解决的问题，也可以用动态规划解决。**

实际上，贪心算法适用的情况很少。一般对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，它需要证明后才能真正运用到题目的算法中。贪心策略一旦经过证明成立后，它就是一种高效的算法。

下面分析背包问题：

背包问题

有一个背包，背包容量是 $M=150$ 。有 7 个物品，物品可以分割成任意大小。要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

物品 A B C D E F G

重量 35 30 60 50 40 10 25

价值 10 40 30 50 35 40 30

分析：

目标函数： $\sum p_i$ 最大

约束条件是装入的物品总重量不超过背包容量： $\sum w_i \leq M$ ($M=150$)

- (1) 根据贪心的策略，每次挑选价值最大的物品装入背包，得到的结果是否最优？
- (2) 每次挑选所占重量最小的物品装入是否能得到最优解？
- (3) 每次选取单位重量价值最大的物品，成为解本题的策略。

对于例题中的 3 种贪心策略，都是无法成立（无法被证明）的，解释如下：

- (1) 贪心策略：选取价值最大者。反例：

假设背包容量 $W = 30$

物品：A B C

重量：28 12 12

价值：30 20 20

根据策略，首先选取物品 A，接下来就无法再选取了，可是，选取 B、C 则更好。

(2) 贪心策略：选取重量最小。它的反例与第一种策略的反例差不多。

(3) 贪心策略：选取单位重量价值最大的物品。反例：

假设背包容量 $W = 30$ 物品：A B C

重量：28 20 10

价值：28 20 10

根据策略，三种物品单位重量价值一样，程序无法依据现有策略 做出判断，如果选择 A，因为 $(B+C) > A$ ，则答案错误。

1.4 应用

贪心算法有很多经典的应用，比如霍夫曼编码（Huffman Coding）、Prim 和 克鲁斯卡尔最小生成树算法、还有 Dijkstra 单源最短路径算法。

Prim 最小生成树算法

基本思想：对于图 G 而言， V 是所有顶点的集合；现在，设置两个新的集合 U 和 T ，其中 U 用于存放 G 的最小生成树中的顶点， T 存放 G 的最小生成树中的边。

从所有 $u \in U, v \in (V-U)$ ($V-U$ 表示除去 U 的所有顶点) 的边中选取权值最小的边 (u, v) ，将顶点 v 加入集合 U 中，将边 (u, v) 加入集合 T 中，如此不断重复，直到 $U=V$ 为止，最小生成树构造完毕，这时集合 T 中包含了最小生成树中的所有边。

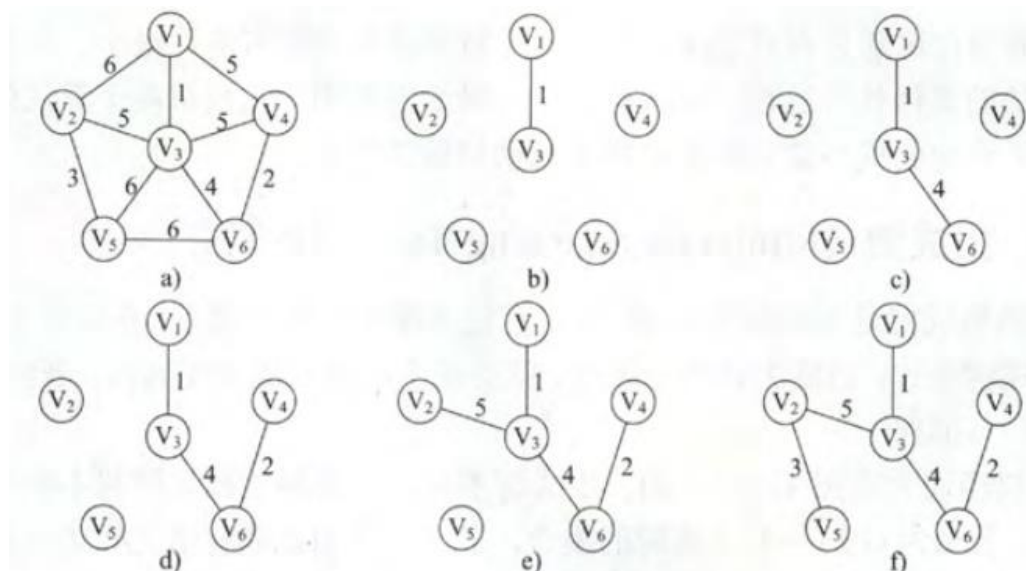


图 5-20 Prim 算法构造最小生成树的过程

克鲁斯卡尔最小生成树算法

基本思想：

设一个有 n 个顶点的连通网络为 $G(V, E)$ ，最初先构造一个只有 n 个顶点，没有边的非连通图 $T = \{V, \emptyset\}$ ，图中每个顶点自成一连通分量。

在 E 中选择一条具有权植最小的边时，若该边的两个顶点落在不同的连通分量上，则将此边加入到 T 中；否则，即这条边的两个顶点落到同一连通分量上，则将此边舍去（此后永不选用这条边），重新选择一条权植最小的边。

如此重复下去，直到所有顶点在同一连通分量上为止。

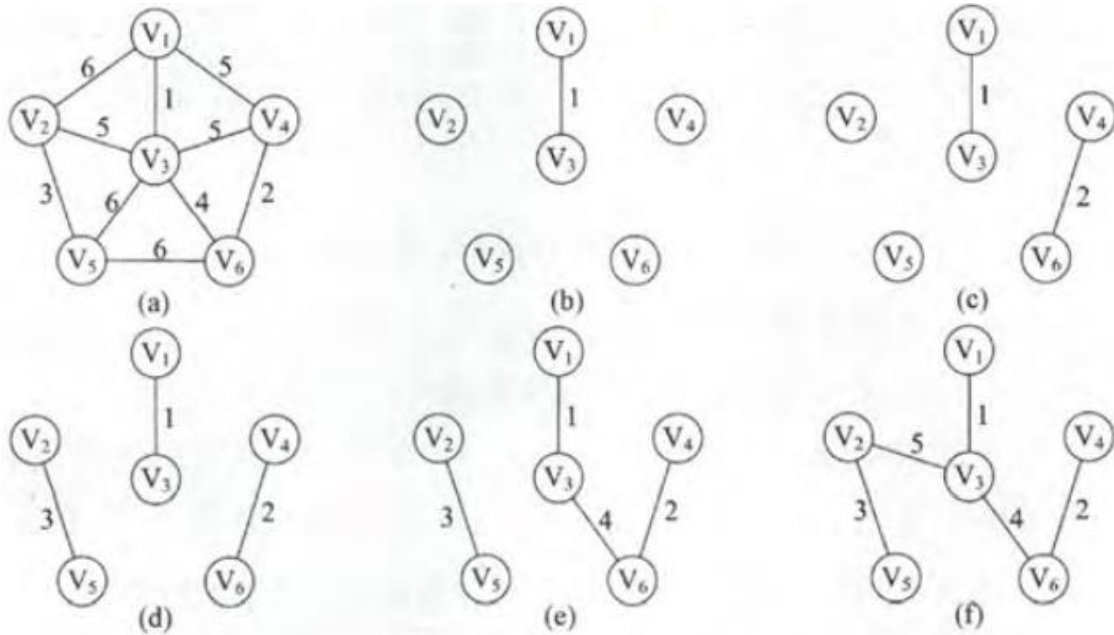


图 5-21 Kruskal 算法构造最小生成树的过程

十三、动态规划

1.1 动态规划过程

每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，**这种多阶段最优化决策解决问题的过程就称为动态规划**。

基本思想与分治法类似，也是**将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息**。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解，如下斐波那契数列求解）。

1.2 动态规划适用场景

能采用动态规划求解的问题的一般要具有 3 个性质：

(1) **最优化原理：**如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

(2) **无后效性：**即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某个状态以后的过程不会影响以前的状态，只与当前状态有关。

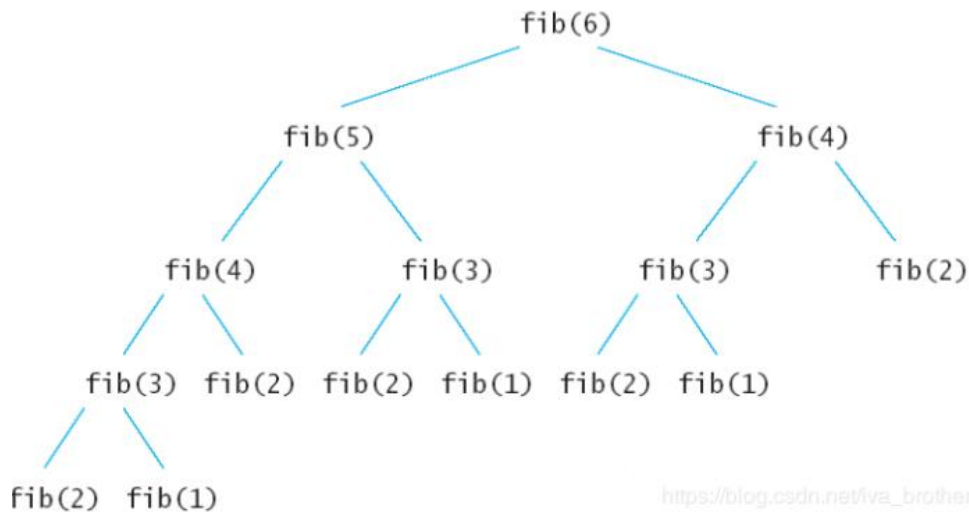
(3) **有重叠子问题：**即子问题之间不独立，一个子问题在下一阶段决策中可能被多次使用到。（**该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势**）

1.3 解题的一般步骤

- 1) 找出最优解的性质，刻画其结构特征和最优子结构特征；
- 2) 递归地定义最优值，刻画原问题解与子问题解间的关系；
- 3) 以自底向上的方式计算出各个子问题、原问题的最优值，并避免子问题的重复计算；
- 4) 根据计算最优值时得到的信息，构造最优解。

1.4 示例

斐波那契数列



上面的每个节点都会被执行一次,导致同样的节点被重复的执行,比如 fib(2)被执行了 5 次。这样导致时间上的浪费,如果递归调用也会导致空间的浪费,导致栈溢出的问题。

上面的递归的求解方式就是分治算法,由于它的子问题是相互相关的,此时利用分治法就做了很多重复的工作,它会反复求解那些公共子子问题。而动态规划算法对每一个子子问题只求解一次,将其保存在一个表格中,从而避免重复计算。

```
int fib(int n)
{
    int* arr = (int *)malloc(sizeof(int) * (n+1));
    if (n <= 0) {
        return n;
    }

    arr[0] = 0;
    arr[1] = 1;

    for (int i = 2; i <= n; i++) {
        arr[i] = arr[i - 1] + arr[i - 2];
    }

    return arr[n];
}
```

最长上升子序列 (LIS)

动态规划的一个特点就是当前解可以由上一个阶段的解推出，由此，**把我们要求的问题简化成一个更小的子问题。子问题具有相同的求解方式**，只不过是规模小了而已。最长上升子序列就符合这一特性。

我们要求 n 个数的最长上升子序列，可以求前 $n-1$ 个数的最长上升子序列，再跟第 n 个数进行判断。

求前 $n-1$ 个数的最长上升子序列，可以通过求前 $n-2$ 个数的最长上升子序列……直到求前 1 个数的最长上升子序列，此时 LIS 当然为 1。

$$dp_i = \max \{dp_j + 1\} \quad \text{for } j < i \text{ and } A_j < A_i$$

求 2 7 1 5 6 4 3 8 9 的最长上升子序列。我们定义 $d(i)$ ($i \in [1, n]$) 来表示前 i 个数以 $A[i]$ 结尾的最长上升子序列长度。

前 1 个数 $d(1)=1$ 子序列为 2;

前 2 个数 7 前面有 2 小于 7 $d(2)=d(1)+1=2$ 子序列为 2 7

前 3 个数 在 1 前面没有比 1 更小的，1 自身组成长度为 1 的子序列 $d(3)=1$ 子序列为 1

前 4 个数 5 前面有 2 小于 5 $d(4)=d(1)+1=2$ 子序列为 2 5

前 5 个数 6 前面有 2 5 小于 6 $d(5)=d(4)+1=3$ 子序列为 2 5 6

前 6 个数 4 前面有 2 小于 4 $d(6)=d(1)+1=2$ 子序列为 2 4

前 7 个数 3 前面有 2 小于 3 $d(3)=d(1)+1=2$ 子序列为 2 3

前 8 个数 8 前面有 2 5 6 小于 8 $d(8)=d(5)+1=4$ 子序列为 2 5 6 8

前 9 个数 9 前面有 2 5 6 8 小于 9 $d(9)=d(8)+1=5$ 子序列为 2 5 6 8 9

$d(i)=\max\{d(1), d(2), \dots, d(i)\}$ 我们可以看出这 9 个数的 LIS 为 $d(9)=5$

```
int LIS(int A[], int n) {
    int* arr = (int *)malloc(sizeof(int) * (n));
    int len = 1;

    for (int i = 0; i < n; i++) {
        arr[i] = 1;
        for (int j = 0; j < i; j++) {
```

```
        if (A[j] <= A[i] && (arr[j] + 1) >= arr[i]) {  
            arr[i] = arr[j] + 1;  
        }  
    }  
  
    if (arr[i]>len) {  
        len = arr[i];  
    }  
}  
free(arr);  
return len;  
}
```

```
$ ./result.exe  
--求斐波那契数列--  
动态规划: 1 1 2 3 5 8 13 21 34 55  
  
--最长上升子序列--  
{2,7,1,5,6,4,3,8,9} : 最长子序列长度: 5
```