胡伟煌 数据结构学习笔记

书栈(BookStack.CN)

目　录

致谢

当前文档 《胡伟煌 数据结构 学习笔记》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-05-05。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：胡伟煌 https://www.huweihuang.com/data-structure-notes/

文档地址：http://www.bookstack.cn/books/huweihuang-data-structure-notes

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

数据结构学习笔记

> 本系列是 数据结构学习笔记
>
> 更多的学习笔记请参考:
>
> - Kubernetes 学习笔记
> - Golang 学习笔记
> - Linux 学习笔记
> - 数据结构学习笔记
>
> 个人博客: www.huweihuang.com

目录

赞赏

> 如果觉得文章有帮助的话，可以打赏一下，谢谢！

1. 数据结构的基本概念

“数据结构”是研究各种数据的特性以及数据之间存在的关系，进而根据实际应用的要求，合理地组织和存储数据，设计出相应的算法。

数据是对客观事物的符号表示，

- `数据元素（节点）`：数据的基本单位，在程序中通常作为一个整体进行考虑和处理。一个数据元素可以由若干个数据项组成。
- `数据项`：具有独立含义的最小标识单位。例如，一条数据记录可以称为一个数据元素，数据记录的某个字段就是一个数据项。
- `数据结构`：相互之间存在一种或多种特点关系的数据元素的集合。

1.1. 数据的逻辑结构

数据的逻辑结构：数据元素与数据元素之间的逻辑关系。可以分为四类基本结构：

- `集合`：结构中的数据元素属于一个集合（集合类型元素之间过于松散）
- `线性结构`：结构中的数据元素存在一对一的关系
- `树形结构`：结构中的数据元素存在一对多的关系
- `图形结构`：结构中的数据元素存在多对多的关系

数据的逻辑结构可以用以下的二元组来表示：

`S=(D,R)`

其中，D是数据节点的有限集合，R是D上的关系的有限集合，其中每个关系都是从D到D的关系。

例如：

```
1.  S = (D,R)
2.  D = {1,2,3,4}
3.  R = {r}
4.  r = {<1,2>,<1,3>,<3,4>}
```

说明：

- 尖括号表示`有向`关系，例如`<a,b>`，表示a→b
- 圆括号表示`无向`关系，例如`(a,b)`，表示a→b,b→a
- `前驱结点`：中a是b的前驱结点
- `后继结点`：中b是a的后继结点

- 开始结点：没有前驱结点
- 终端结点：没有后继结点
- 内部结点：既有前驱结点，又有后继结点

1.2. 数据的存储结构

数据在计算机中的存储表示称为数据的存储结构，又称物理结构。
数据存储到计算机中即要求存储各节点的 数值 ，又要存储结点与结点之间的 逻辑关系 。
以下介绍四种基本的存储结构： 顺序存储 、 链式存储 、 索引存储 、 散列存储 。

1、顺序存储结构

顺序存储结构是把逻辑上相邻的元素存储在一组连续的存储单元中，其元素之间的逻辑关系由 存储单元地址 间的关系隐含表示。

优点：节省存储空间，只需要存储数据结点，并不需要存储结点的逻辑关系。
缺点：不便于修改，插入和删除某个结点需要修改一系列的结点。

2、链式存储结构

链式存储结构，给每个结点增加指针字段，用于存放临近结点的存储地址，每个结点占用两个连续的存储单元，一个存放数据，一个存放临近结点（前驱/后继结点）的地址。

优点：便于修改，修改时只需要修改结点的指针字段，不需要移动其他结点。
缺点：占用存储空间，因为需要存储结点之间的逻辑关系。因为结点之间不一定相邻，因此不能对结点进行随机访问。

3、索引存储结构

索引存储结构即在存储结点的同时，增加索引表，索引表的索引项为：（关键字，地址），关键字标识结点，地址为结点的指针。各结点的地址在索引表中是依次排列的。

优点：可以快速查找，可以随机访问，方便修改。
缺点：建立索引表增加了时间和空间的开销。

4、散列存储结构

散列存储结构是根据结点的值确定结点的存储地址。以结点作为自变量，通过散列函数算出结果i，再把i作为结点的存储地址。

优点：查找速度快，适用于快速查找和插入的场景。
缺点：只存结点数据，不存结点之间的关系。

1.3. 数据的运算

数据的运算就是施加于数据的操作，例如对一张表进行增删改查操作，一般数据结构中的运算除了加减乘除外还会涉及 `算法问题` 。

1.4. 数据结构与数据类型

按某种 `逻辑关系` 组成的数据元素，按一定的 `存储方式` 存储于计算机中，并在其上定义了一个 `运算` 的集合，称为一个 `数据结构` 。

```
数据结构 = 数据的逻辑结构 + 数据的存储结构 + 数据的运算(算法)
```

`数据类型` 是程序设计语言中对数据结构的实现，数据类型明显或隐含地规定了数据的取值范围、存储方式及允许进行的运算。

常用的数据类型：

1. 基本数据类型
2. 指针类型
3. 数组类型
4. 结构体类型
5. 组合体类型
6. 自定义类型

2. 算法的基本概念

2.1. 算法及其特征

算法是对特定问题求解步骤的描述，是指令的有限序列，每条指令包含一个或多个操作。

特点：

- 有穷性：有限的步骤和有限的时间内完成
- 确定性：每个指令有确定的含义
- 可行性：算法是可以实现的
- 输入性：一个或多个输入
- 输出性：一个或多个输出

2.2. 算法描述

1. 输入语句
2. 输出语句

3. 赋值语句
4. 条件语句
5. 循环语句
6. 返回语句（return）
7. 定义函数语句
8. 调用函数语句

2.3. 算法分析

2.3.1. 时间复杂度

算法分析主要涉及 `时间复杂度` 和 `空间复杂度` 。一般情况我们讨论时间复杂度。

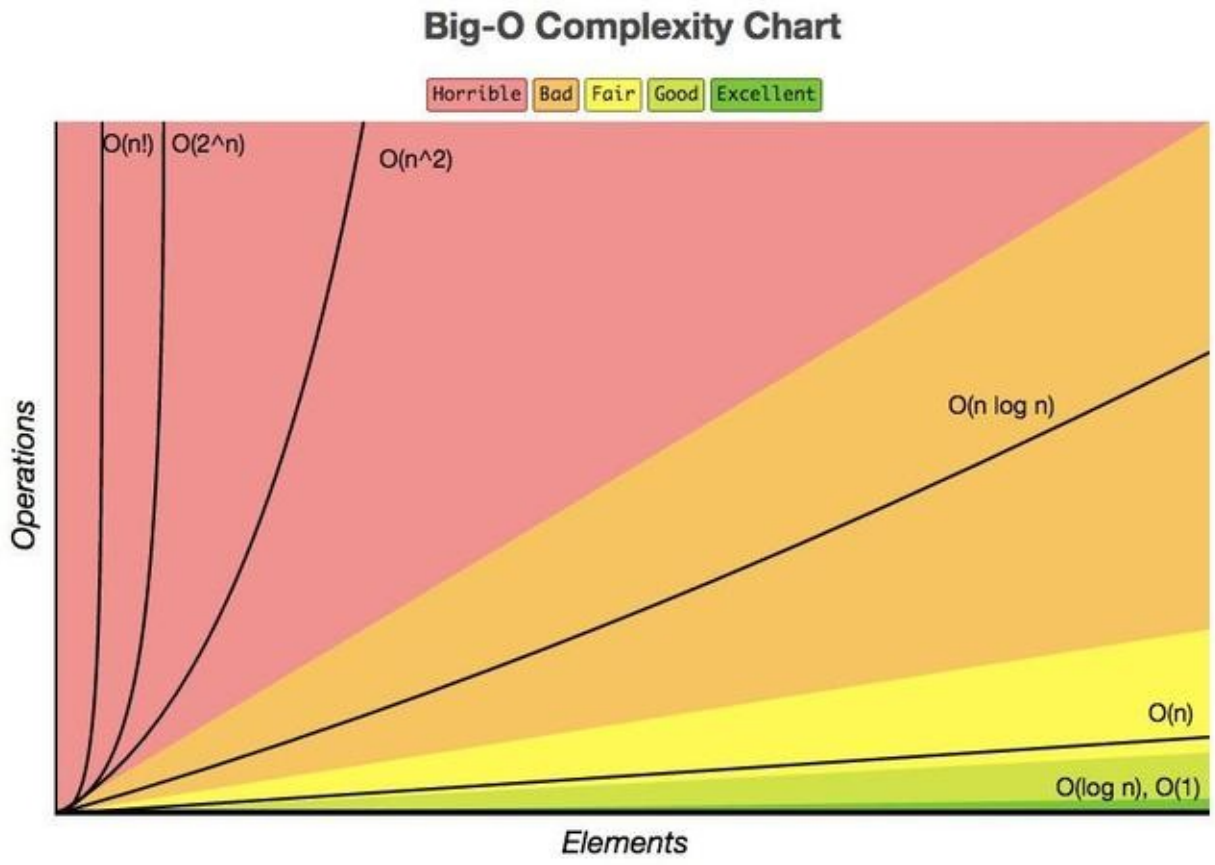
- `频度`：某语句在算法中被执行的次数。
- `T(n)`：所有语句的频度之和，n为问题规模。
- `时间复杂度`：当n趋于无穷大时，T(n)的数量级。记作 `T(n)=O(f(n))`，O的含义是T(n)的数量级。

用数量级O(f(n))表示算法执行时间T(n)时，f(n)一般去简单形式：`1`， `log_2n`， `n`， `nlog_2n`， `n^2`， `n^3`， `2^n`。

时间复杂度的关系如下：

O(1) < O(log_2n) < O(n) < O(nlog_2n) < O(n^2) < O(n^3) < O(2^n)

时间复杂度函数对比（图片来自网络）：



2.3.2. 空间复杂度

一个算法的空间复杂度是指该算法所耗费的存储空间，计算公式计作：S(n) = O(f(n))。其中 n 也为数据的规模，f(n) 在这里指的是 n 所占存储空间的函数。

常用的空间复杂度：

- 空间复杂度 O(1)
- 空间复杂度 O(n)
- 空间复杂度 O(n^2)

文章参考：

《数据结构教程》[清华大学出版社]

- 线性表的基本概念
- 顺序表的基本运算
- 单链表的基本运算
- 链表的基本运算(By-Go)
- 循环单链表的基本运算
- 双链表的基本运算
- 循环双链表的基本运算
- 顺序表求约瑟夫问题
- 两个多项式相加运算

线性表的基本概念

线性表的定义

线性表是由n（n>=0）个结点组成的有限序列，通常表示成（a1,a2,…,an），满足以下特征。

线性表的特征

- 线性表中每个结点至多只有一个前驱结点且至多只有一个后继结点
- 起始结点没有前驱结点
- 终结结点没有后继结点

线性表的基本运算

- 初始化线性表
- 求线性表的长度
- 求线性表的第i个元素
- 按值查找元素，返回元素序号
- 插入元素
- 删除元素
- 输出列表

线性表的存储结构

顺序存储结构

```
#define MAXSIZE 100 /*顺序表的容量*/

typedef char ElemType;

typedef struct
{
    ElemType data[MAXSIZE]; /*存放顺序表的元素*/
    int length;                /*顺序表的实际长度*/
} SqList;
```

链式存储结构

单链表

```
typedef char ElemType;

typedef struct node
{
    ElemType data;        /*数据域*/
    struct node *next;     /*指针域*/
} SLink;
```

双链表

```
typedef char ElemType;

typedef struct node
{
    ElemType data;                /*数据域*/
    struct node *prior,*next;  /*分别指向前驱结点和后继结点的指针*/
} DLink;
```

顺序表的基本运算

线性表的定义

```
#include <stdio.h>
#define MAXSIZE 100 /*顺序表的容量*/

typedef char ElemType;

typedef struct
{
    ElemType data[MAXSIZE]; /*存放顺序表的元素*/
    int length;                /*顺序表的实际长度*/
} SqList;
```

初始化线性表

```
void InitList(SqList &sq) /*初始化线性表*/
{
    sq.length = 0;
}
```

求线性表长度

```
int GetLength(SqList sq) /*求线性表长度*/
{
    return sq.length;
}
```

求线性表中第i个元素

```
int GetElem(SqList sq, int i, ElemType &e) /*求线性表中第i个元素*/
{
    if (i < 1 || i > sq.length) /*无效的i值*/
        return 0;
    else
```

```
    {
        e = sq.data[i - 1];
        return 1;
    }
}
```

按值查找

```
int Locate(SqList sq, ElemType x) /*按值查找*/
{
    int i = 0;
    while (sq.data[i] != x) /*查找值为x的第1个结点*/
        i++;
    if (i > sq.length)
        return (0); /*未找到*/
    else
        return (i + 1);
}
```

插入元素

```
int InsElem(SqList &sq, ElemType x, int i) /*插入元素*/
{
    int j;
    if (i < 1 || i > sq.length + 1) /*无效的参数i*/
        return 0;
    for (j = sq.length; j > i; j--) /*将位置为i的结点及之后的结点后移*/
        sq.data[j] = sq.data[j - 1];
    sq.data[i - 1] = x; /*在位置i处放入x*/
    sq.length++;        /*线性表长度增1*/
    return 1;
}
```

删除元素

```
int DelElem(SqList &sq, int i) /*删除元素*/
{
    int j;
```

```
    if (i < 1 || i > sq.length) /*无效的参数i*/
        return 0;
    for (j = i; j < sq.length; j++) /*将位置为i的结点之后的结点前移*/
        sq.data[j - 1] = sq.data[j];
    sq.length--; /*线性表长度减1*/
    return 1;
}
```

输出线性表

```
void DispList(SqList sq) /*输出线性表*/
{
    int i;
    for (i = 1; i <= sq.length; i++)
        printf("%c ", sq.data[i - 1]);
    printf("\n");
}
```

main

```
void main()
{
    int i;
    ElemType e;
    SqList sq;
    InitList(sq);          /*初始化顺序表sq*/
    InsElem(sq, 'a', 1); /*插入元素*/
    InsElem(sq, 'c', 2);
    InsElem(sq, 'a', 3);
    InsElem(sq, 'e', 4);
    InsElem(sq, 'd', 5);
    InsElem(sq, 'b', 6);
    printf("线性表:");
    DispList(sq);
    printf("长度:%d\n", GetLength(sq));
    i = 3;
    GetElem(sq, i, e);
    printf("第%d个元素:%c\n", i, e);
    e = 'a';
    printf("元素%c是第%d个元素\n", e, Locate(sq, e));
```

```
    i = 4;
    printf("删除第%d个元素\n", i);
    DelElem(sq, i);
    printf("线性表:");
    DispList(sq);
}
```

单链表的基本运算

单链表的定义

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

typedef char ElemType;

typedef struct node
{
    ElemType data;        /*数据域*/
    struct node *next;     /*指针域*/
} SLink;
```

初始化单链表

```
void InitList(SLink *&L)     /*L作为引用型参数*/
{
    L=(SLink *)malloc(sizeof(SLink));  /*创建头结点*L*/
    L->next=NULL;
}
```

求线性表的长度

```
int GetLength(SLink *L)    /*求线性表的长度*/
{
    int i=0;
    SLink *p=L->next;
    while (p!=NULL)
    {
        i++;
        p=p->next;
    }
    return i;
}
```

求线性表中第i个元素

```
int GetElem(SLink *L,int i,ElemType &e)    /*求线性表中第i个元素*/
{
    int j=1;
    SLink *p=L->next;
    if (i<1 || i>GetLength(L))
        return(0);                 /*i参数不正确,返回0*/
    while (j<i)                   /*从第1个结点开始找,查找第i个结点*/
    {
        p=p->next;j++;
    }
    e=p->data;
    return(1);                   /*返回1*/
}
```

按值查找

```
int Locate(SLink *L,ElemType x)    /*按值查找*/
{
    int i=1;
    SLink *p=L->next;
    while (p!=NULL && p->data!=x)      /*从第1个结点开始查找data域为x的结点*/
    {
        p=p->next;
        i++;
    }
    if (p==NULL)
        return(0);
    else
        return(i);
}
```

插入结点

```
int InsElem(SLink *L,ElemType x,int i)    /*插入结点*/
{
    int j=1;
    SLink *p=L,*s;
```

```
    s=(SLink *)malloc(sizeof(SLink));    /*创建data域为x的结点*/
    s->data=x;s->next=NULL;
    if (i<1 || i>GetLength(L)+1)
    return 0;             /*i参数不正确,插入失败,返回0*/
    while (j<i)           /*从头结点开始找,查找第i-1个结点,由p指向它*/
    {
        p=p->next;j++;
    }
    s->next=p->next;      /*将*s的next域指向*p的下一个结点(即第i个结点)*/
    p->next=s;             /*将*p的next域指向*s,这样*s变成第i个结点*/
    return 1;            /*插入运算成功,返回1*/
}
```

删除结点

```
int DelElem(SLink *L,int i)    /*删除结点*/
{
    int j=1;
    SLink *p=L,*q;
    if (i<1 || i>GetLength(L))
        return 0;        /*i参数不正确,插入失败,返回0*/
    while (j<i)             /*从头结点开始,查找第i-1个结点,由p指向它*/
    {
        p=p->next;j++;
    }
    q=p->next;           /*由q指向第i个结点*/
    p->next=q->next;     /*将*p的next指向*q之后结点,即从链表中删除第i个结点*/
    free(q);                 /*释放第i个结点占用的空间*/
    return 1;             /*删除运算成功,返回1*/
}
```

输出单链表

```
void DispList(SLink *L)    /*输出单链表*/
{
    SLink *p=L->next;
    while (p!=NULL)
    {
        printf("%c ",p->data);
        p=p->next;
```

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

main

```
void main()
{
    int i;
    ElemType e;
    SLink *L;
    InitList(L);        /*初始化单链表L*/
    InsElem(L,'a',1);    /*插入元素*/
    InsElem(L,'c',2);
    InsElem(L,'a',3);
    InsElem(L,'e',4);
    InsElem(L,'d',5);
    InsElem(L,'b',6);
    printf("线性表:");DispList(L);
    printf("长度:%d\n",GetLength(L));
    i=3;GetElem(L,i,e);
    printf("第%d个元素:%c\n",i,e);
    e='a';
    printf("元素%c是第%d个元素\n",e,Locate(L,e));
    i=4;printf("删除第%d个元素\n",i);
    DelElem(L,i);
    printf("线性表:");DispList(L);
}
```

By Golang

```go
// Package linkedlist creates a ItemLinkedList data structure for the Item type
package linkedlist

import (
    "fmt"
    "sync"
)

// Item the type of the linked list
type Item interface{}

// Node a single node that composes the list
type Node struct {
    content Item
    next    *Node
}

// ItemLinkedList the linked list of Items
type ItemLinkedList struct {
    head *Node
    size int
    lock sync.RWMutex
}

// Append adds an Item to the end of the linked list
func (ll *ItemLinkedList) Append(t Item) {
    ll.lock.Lock()
    node := Node{t, nil}
    if ll.head == nil {
        ll.head = &node
    } else {
        last := ll.head
        for {
            if last.next == nil {
                break
            }
            last = last.next
        }
```

```
39.         last.next = &node
40.     }
41.     ll.size++
42.     ll.lock.Unlock()
43. }
44.
45. // Insert adds an Item at position i
46. func (ll *ItemLinkedList) Insert(i int, t Item) error {
47.     ll.lock.Lock()
48.     defer ll.lock.Unlock()
49.     if i < 0 || i > ll.size {
50.         return fmt.Errorf("Index out of bounds")
51.     }
52.     addNode := Node{t, nil}
53.     if i == 0 {
54.         addNode.next = ll.head
55.         ll.head = &addNode
56.         return nil
57.     }
58.     node := ll.head
59.     j := 0
60.     for j < i-2 {
61.         j++
62.         node = node.next
63.     }
64.     addNode.next = node.next
65.     node.next = &addNode
66.     ll.size++
67.     return nil
68. }
69.
70. // RemoveAt removes a node at position i
71. func (ll *ItemLinkedList) RemoveAt(i int) (*Item, error) {
72.     ll.lock.Lock()
73.     defer ll.lock.Unlock()
74.     if i < 0 || i > ll.size {
75.         return nil, fmt.Errorf("Index out of bounds")
76.     }
77.     node := ll.head
78.     j := 0
79.     for j < i-1 {
80.         j++
```

```
81.         node = node.next
82.     }
83.     remove := node.next
84.     node.next = remove.next
85.     ll.size--
86.     return &remove.content, nil
87. }
88.
89. // IndexOf returns the position of the Item t
90. func (ll *ItemLinkedList) IndexOf(t Item) int {
91.     ll.lock.RLock()
92.     defer ll.lock.RUnlock()
93.     node := ll.head
94.     j := 0
95.     for {
96.         if node.content == t {
97.             return j
98.         }
99.         if node.next == nil {
100.            return -1
101.        }
102.        node = node.next
103.        j++
104.    }
105. }
106.
107. // IsEmpty returns true if the list is empty
108. func (ll *ItemLinkedList) IsEmpty() bool {
109.     ll.lock.RLock()
110.     defer ll.lock.RUnlock()
111.     if ll.head == nil {
112.         return true
113.     }
114.     return false
115. }
116.
117. // Size returns the linked list size
118. func (ll *ItemLinkedList) Size() int {
119.     ll.lock.RLock()
120.     defer ll.lock.RUnlock()
121.     size := 1
122.     last := ll.head
```

```
123.     for {
124.         if last == nil || last.next == nil {
125.             break
126.         }
127.         last = last.next
128.         size++
129.     }
130.     return size
131. }
132. 
133. // Insert adds an Item at position i
134. func (ll *ItemLinkedList) String() {
135.     ll.lock.RLock()
136.     defer ll.lock.RUnlock()
137.     node := ll.head
138.     j := 0
139.     for {
140.         if node == nil {
141.             break
142.         }
143.         j++
144.         fmt.Print(node.content)
145.         fmt.Print(" ")
146.         node = node.next
147.     }
148.     fmt.Println()
149. }
150. 
151. // Head returns a pointer to the first node of the list
152. func (ll *ItemLinkedList) Head() *Node {
153.     ll.lock.RLock()
154.     defer ll.lock.RUnlock()
155.     return ll.head
156. }
```

循环单链表的基本运算

循环单链表的定义

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

typedef char ElemType;

typedef struct node
{
    ElemType data;            /*数据域*/
    struct node *next;         /*指针域*/
} SLink;
```

初始化循环单链表

```
void InitList(SLink *&L)       /*初始化线性表,L为引用型参数*/
{
    L=(SLink *)malloc(sizeof(SLink));
    L->next=L;
}
```

求线性表的长度

```
int GetLength(SLink *L)    /*求线性表的长度*/
{
    int i=0;
    SLink *p=L->next;
    while (p!=L)
    {
        i++;p=p->next;
    }
    return i;
}
```

求线性表中第i个元素

```
int GetElem(SLink *L,int i,ElemType &e)    /*求线性表中第i个元素*/
{
    int j=1;
    SLink *p=L->next;
    if (i<1 || i>GetLength(L))
        return(0);             /*i参数不正确,返回0*/
    while (j<i)               /*从第1个结点开始,查找第i个结点*/
    {
        p=p->next;
        j++;
    }
    e=p->data;
    return(1);               /*返回1*/
}
```

按值查找

```
int Locate(SLink *L,ElemType x)    /*按值查找*/
{
    int i=1;
    SLink *p=L->next;
    while (p!=L && p->data!=x)      /*从第1个结点开始查找data域为x的结点*/
    {    p=p->next;
        i++;
    }
    if (p==L)
        return(0);
    else
        return(i);
}
```

插入结点

```
int InsElem(SLink *L,ElemType x,int i)    /*插入结点*/
{
    int j=1;
    SLink *p=L,*s;
```

```
    s=(SLink *)malloc(sizeof(SLink));
    s->data=x;s->next=NULL;
    if (i<1 || i>GetLength(L)+1)
        return 0;
    while (j<i)
    {
        p=p->next;j++;
    }
    s->next=p->next;
    p->next=s;
    return 1;
}
```

删除结点

```
int DelElem(SLink *L,int i)    /*删除结点*/
{
    int j=1;
    SLink *p=L,*q;
    if (i<1 || i>GetLength(L))
        return 0;
    while (j<i)
    {
        p=p->next;j++;
    }
    q=p->next;
    p->next=q->next;
    free(q);
    return 1;
}
```

输出线性表

```
void DispList(SLink *L)    /*输出线性表*/
{
    SLink *p=L->next;
    while (p!=L)
    {
        printf("%c ",p->data);p=p->next;
    }
```

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

main

```
void main()
{
    int i;
    ElemType e;
    SLink *L;
    InitList(L);        /*初始化单链表L*/
    InsElem(L,'a',1);    /*插入元素*/
    InsElem(L,'c',2);
    InsElem(L,'a',3);
    InsElem(L,'e',4);
    InsElem(L,'d',5);
    InsElem(L,'b',6);
    printf("线性表:");DispList(L);
    printf("长度:%d\n",GetLength(L));
    i=3;GetElem(L,i,e);
    printf("第%d个元素:%c\n",i,e);
    e='a';
    printf("元素%c是第%d个元素\n",e,Locate(L,e));
    i=4;printf("删除第%d个元素\n",i);
    DelElem(L,i);
    printf("线性表:");DispList(L);
}
```

双链表的基本运算

双链表的定义

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

typedef char ElemType;

typedef struct node
{
    ElemType data;                /*数据域*/
    struct node *prior,*next;  /*分别指向前驱结点和后继结点的指针*/
} DLink;
```

初始化双链表

```
void InitList(DLink *&L)
{
    L=(DLink *)malloc(sizeof(DLink));  /*创建头结点*L*/
    L->prior=L->next=NULL;
}
```

求表长运算

```
int GetLength(DLink *L)    /*求表长运算*/
{
    int i=0;
    DLink *p=L->next;
    while (p!=NULL)
    {
        i++;p=p->next;
    }
    return i;
}
```

求线性表中第i个元素

```
int GetElem(DLink *L,int i,ElemType &e)    /*求线性表中第i个元素*/
{
    int j=1;
    DLink *p=L->next;
    if (i<1 || i>GetLength(L))
        return(0);                 /*i参数不正确,返回0*/
    while (j<i)                   /*从第1个结点开始,查找第i个结点*/
    {
        p=p->next;j++;
    }
    e=p->data;
    return(1);                   /*返回1*/
}
```

按值查找

```
int Locate(DLink *L,ElemType x)    /*按值查找*/
{
    int i=1;
    DLink *p=L->next;
    while (p!=NULL && p->data!=x)      /*从第1个结点开始查找data域为x的结点*/
    {
        p=p->next;
        i++;
    }
    if (p==NULL)
        return(0);
    else
        return(i);
}
```

插入运算

```
int InsElem(DLink *L,ElemType x,int i)    /*插入运算*/
{
    int j=1;
    DLink *p=L,*s;
```

```
    s=(DLink *)malloc(sizeof(DLink));      /*创建data域为x的结点*/
    s->data=x;s->prior=s->next=NULL;
    if (i<1 || i>GetLength(L)+1)        /*i参数不正确,插入失败,返回0*/
        return 0;
    while (j<i)              /*找到第i-1个结点,由p指向它*/
    {
            p=p->next;j++;
    }
    s->next=p->next;           /**s的next域指向*p的下一个结点*/
    s->prior=p;              /**s的prior域指向*p*/
    if (p->next!=NULL)      /*若*p不是最后结点,则将*p之后结点的prior域指向*s*/
        s->next->prior=s;
     p->next=s;              /**p的next域指向*s*/
    return 1;                /*插入运算成功,返回1*/
}
```

删除运算

```
int DelElem(DLink *L,int i)    /*删除运算*/
{
    int j=1;
    DLink *p=L,*q;
    if (i<1 || i>GetLength(L))    /*i参数不正确,删除失败,返回0*/
        return 0;
    while (j<i)                /*找到第i-1个结点,由p指向它*/
    {
           p=p->next;j++;
    }
    q=p->next;                 /*q指向*p的下一个结点,即要删除的结点*/
    p->next=q->next;
    if (q->next!=NULL)        /*若*q不是最后结点,则将*q之后结点的prior域指向*p*/
        q->next->prior=p;
    free(q);                   /*释放第i个结点占用的空间*/
    return 1;                  /*删除运算成功,返回1*/
}
```

输出线性表

```
void DispList(DLink *L)    /*输出线性表*/
{
```

```
    DLink *p=L->next;
    while (p!=NULL)
    {
        printf("%c ",p->data);p=p->next;
    }
    printf("\n");
}
```

main

```
void main()
{
    int i;
    ElemType e;
    DLink *L;
    InitList(L);        /*初始化双链表L*/
    InsElem(L,'a',1);    /*插入元素*/
    InsElem(L,'c',2);
    InsElem(L,'a',3);
    InsElem(L,'e',4);
    InsElem(L,'d',5);
    InsElem(L,'b',6);
    printf("线性表:");DispList(L);
    printf("长度:%d\n",GetLength(L));
    i=3;GetElem(L,i,e);
    printf("第%d个元素:%c\n",i,e);
    e='a';
    printf("元素%c是第%d个元素\n",e,Locate(L,e));
    i=4;printf("删除第%d个元素\n",i);
    DelElem(L,i);
    printf("线性表:");DispList(L);
}
```

循环双链表的基本运算

循环双链表的定义

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

typedef char ElemType;

typedef struct node
{
    ElemType data;                /*数据域*/
    struct node *prior,*next;  /*分别指向前驱结点和后继结点的指针*/
} DLink;
```

初始化循环双链表

```
void InitList(DLink *&L)
{
    L=(DLink *)malloc(sizeof(DLink));
    L->prior=L->next=L;
}
```

求表长运算

```
int GetLength(DLink *L)    /*求表长运算*/
{
    int i=0;
    DLink *p=L->next;
    while (p!=L)
    {
        i++;p=p->next;
    }
    return i;
}
```

求线性表中第i个元素

```
int GetElem(DLink *L,int i,ElemType &e)    /*求线性表中第i个元素*/
{
    int j=1;
    DLink *p=L->next;
    if (i<1 || i>GetLength(L))
        return(0);                 /*i参数不正确,返回0*/
    while (j<i)                   /*从第1个结点开始,查找第i个结点*/
    {
        p=p->next;j++;
    }
    e=p->data;
    return(1);                   /*返回1*/
}
```

按值查找

```
int Locate(DLink *L,ElemType x)    /*按值查找*/
{
    int i=1;
    DLink *p=L->next;
    while (p!=L && p->data!=x)      /*从第1个结点开始查找data域为x的结点*/
    {
        p=p->next;
        i++;
    }
    if (p==L)
        return(0);
    else
        return(i);
}
```

插入运算

```
int InsElem(DLink *L,ElemType x,int i)    /*插入运算*/
{
    int j=1;
    DLink *p=L,*s;
```

```
    s=(DLink *)malloc(sizeof(DLink));
    s->data=x;s->prior=s->next=NULL;
    if (i<1 || i>GetLength(L)+1)
            return 0;
    while (j<i)              /*找到第i-1个结点,由p指向它*/
    {
           p=p->next;j++;
    }
    s->next=p->next;      /*s的next域指向p之后的结点*/
    s->next->prior=s;     /*p之后结点的prior域指向s*/
    p->next=s;             /*p的next域指向s*/
    s->prior=p;            /*s的prior域指向p*/
    return 1;
}
```

删除运算

```
int DelElem(DLink *L,int i)    /*删除运算*/
{
    int j=1;
    DLink *p=L,*q;
    if (i<1 || i>GetLength(L))
            return 0;
    while (j<i)          /*找到第i-1个结点,由p指向它*/
    {
           p=p->next;j++;
    }
    q=p->next;           /*q指向p的下一个结点,即要删除的结点*/
    p->next=q->next;       /*p的next指向q的下一个结点*/
    q->next->prior=p;    /*q的下一个结点的prior域指向p*/
    free(q);             /*释放q所占用的空间*/
    return 1;
}
```

输出线性表

```
void DispList(DLink *L)    /*输出线性表*/
{
    DLink *p=L->next;
    while (p!=L)
```

```
    {
        printf("%c ",p->data);p=p->next;
    }
    printf("\n");
}
```

main

```
void main()
{
    int i;
    ElemType e;
    DLink *L;
    InitList(L);        /*初始化双链表L*/
    InsElem(L,'a',1);    /*插入元素*/
    InsElem(L,'c',2);
    InsElem(L,'a',3);
    InsElem(L,'e',4);
    InsElem(L,'d',5);
    InsElem(L,'b',6);
    printf("线性表:");DispList(L);
    printf("长度:%d\n",GetLength(L));
    i=3;GetElem(L,i,e);
    printf("第%d个元素:%c\n",i,e);
    e='a';
    printf("元素%c是第%d个元素\n",e,Locate(L,e));
    i=4;printf("删除第%d个元素\n",i);
    DelElem(L,i);
    printf("线性表:");DispList(L);
}
```

顺序表求解约瑟夫问题

```c
#include <stdio.h>
#define MaxSize 50

void jose(int n,int m)
{
    int mon[MaxSize];            /*存放n个猴子的编号*/
    int i,d,count;
    for (i=0;i<n;i++)            /*设置猴子的编号*/
        mon[i]=i+1;
    printf("出队前:");            /*输出出列前的编号*/
    for (i=0;i<n;i++)
        printf("%d ",mon[i]);
    printf("\n");
    printf("出队后:");
    count=0;                    /*记录退出圈外的猴子个数*/
    i=-1;                        /*从0号位置的猴子开始计数*/
    while (count<n)
    {
        d=0;
        while (d<m)                /*累计m个猴子*/
        {
            i=(i+1)%n;
            if (mon[i]!=0)
                d++;
        }
        printf("%d ",mon[i]);    /*猴子出列*/
        mon[i]=0;
        count++;                /*出列数增1*/
    }
    printf("\n");
}

void main()
{
    int m,n;
    printf("输入猴子个数n,m:");
    scanf("%d%d",&n,&m);
    jose(n,m);
```

```
39. }
```

两个多项式相加运算

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

typedef struct node
{      float coef;            /*序数*/
       int expn;              /*指数*/
       struct node *next;    /*指向下一个结点的指针*/
} PolyNode;

void InitList(PolyNode *&L)        /*初始化多项式单链表*/
{
    L=(PolyNode *)malloc(sizeof(PolyNode));    /*建立头结点*/
    L->next=NULL;
}

int GetLength(PolyNode *L)        /*求多项式单链表的长度*/
{
    int i=0;
    PolyNode *p=L->next;
    while (p!=NULL)        /*扫描单链表L,用i累计数据结点个数*/
    {
        i++;p=p->next;
    }
    return i;
}

PolyNode *GetElem(PolyNode *L,int i)    /*返回多项式单链表中第i个结点的指针*/
{
    int j=1;
    PolyNode *p=L->next;
    if (i<1 || i>GetLength(L))
           return NULL;
    while (j<i)                /*沿next域找第i个结点*/
    {
           p=p->next;j++;
    }
    return p;
}
```

```
39.
40. PolyNode *Locate(PolyNode *L,float c,int e)    /*在多项式单链表中按值查找*/
41. {
42.     PolyNode *p=L->next;
43.     while (p!=NULL && (p->coef!=c ||p->expn!=e))
44.             p=p->next;
45.     return p;
46. }
47.
48. int InsElem(PolyNode *&L,float c,int e,int i)  /*在多项式单链表中插入一个结点*/
49. {
50.     int j=1;
51.     PolyNode *p=L,*s;
52.     s=(PolyNode *)malloc(sizeof(PolyNode));
53.     s->coef=c;s->expn=e;s->next=NULL;
54.     if (i<1 || i>GetLength(L)+1)
55.         return 0;
56.     while (j<i)         /*查找第i-1个结点*p*/
57.     {
58.         p=p->next;j++;
59.     }
60.     s->next=p->next;
61.     p->next=s;
62.     return 1;
63. }
64.
65. int DelElem(PolyNode *L,int i)         /*在多项式单链表中删除一个结点*/
66. {
67.     int j=1;
68.     PolyNode *p=L,*q;
69.     if (i<1 || i>GetLength(L))
70.         return 0;
71.     while (j<i)         /*在单链表中查找第i-1个结点*p*/
72.     {
73.         p=p->next;j++;
74.     }
75.     q=p->next;
76.     p->next=q->next;
77.     free(q);
78.     return 1;
79. }
80.
```

```
81. void DispList(PolyNode *L)        /*输出多项式单链表的元素值*/
82. {
83.     PolyNode *p=L->next;
84.     while (p!=NULL)
85.     {
86.         printf("(%g,%d) ",p->coef,p->expn);
87.         p=p->next;
88.     }
89.     printf("\n");
90. }
91. void CreaPolyList(PolyNode *&L,float C[],int E[],int n)
92. {
93.     int i;
94.     InitList(L);
95.     for (i=0;i<n;i++)
96.         InsElem(L,C[i],E[i],i+1);
97. }
98.
99. void SortPloy(PolyNode *&L)    /*对L的多项式单链表按expn域递增排序*/
100. {
101.     PolyNode *p=L->next,*q,*pre;
102.     L->next=NULL;
103.     while (p!=NULL)
104.     {
105.         if (L->next==NULL)                    /*处理第1个结点*/
106.         {
107.             L->next=p;p=p->next;
108.             L->next->next=NULL;
109.            }
110.         else                                  /*处理其余结点*/
111.         {
112.             pre=L;q=pre->next;
                 while (q!=NULL && p->expn>q->expn)     /*找q->expn刚大于或等于p->expn
113. 的结点*q的前驱结点*pre*/
114.             {
115.                 pre=q;q=q->next;
116.             }
117.             q=p->next;                       /*在*pre结点之后插入*p*/
118.             p->next=pre->next;
119.             pre->next=p;
120.             p=q;
121.         }
```

```
    }
}

PolyNode *AddPoly(PolyNode *pa,PolyNode *pb)
{
    PolyNode *pc,*p1=pa->next,*p2=pb->next,*p,*tc,*s;
    pc=(PolyNode *)malloc(sizeof(PolyNode));    /*新建头结点*/
    pc->next=NULL;                    /*pc为新建单链表的头结点*/
    tc=pc;                            /*tc始终指向新建单链表的最后结点*/
    while (p1!=NULL && p2!=NULL)
    {
        if (p1->expn<p2->expn)        /*将*p1结点复制到*s并链到pc尾*/
        {
            s=(PolyNode *)malloc(sizeof(PolyNode));
            s->coef=p1->coef;s->expn=p1->expn;s->next=NULL;
            tc->next=s;tc=s;
            p1=p1->next;
        }
        else if (p1->expn>p2->expn)    /*将*p2结点复制到*s并链到pc尾*/
        {
            s=(PolyNode *)malloc(sizeof(PolyNode));
            s->coef=p2->coef;s->expn=p2->expn;s->next=NULL;
            tc->next=s;tc=s;
            p2=p2->next;
        }
        else   /*p1->expn=p2->expn的情况*/
        {
            if (p1->coef+p2->coef!=0) /*序数相加不为0时新建结点*s并链到pc尾*/
            {
                s=(PolyNode *)malloc(sizeof(PolyNode));
                s->coef=p1->coef+p2->coef;s->expn=p1->expn;
                s->next=NULL;
                tc->next=s;tc=s;
            }
            p1=p1->next;p2=p2->next;
        }
    }
    if (p1!=NULL) p=p1;  /*将尚未扫描完的余下结点复制并链接到pc单链表之后*/
    else p=p2;
    while (p!=NULL)
    {
        s=(PolyNode *)malloc(sizeof(PolyNode));
```

```
164.          s->coef=p->coef;s->expn=p->expn;s->next=NULL;
165.          tc->next=s;tc=s;
166.          p=p->next;
167.      }
168.      tc->next=NULL;          /*新建单链表最后结点的next域置空*/
169.      return pc;
170.  }
171.
172.  void main()
173.  {
174.      PolyNode *L1,*L2,*L3;
175.      float C1[]={3,7,5,9},C2[]={-9,8,22};
176.      int E1[]={1,0,17,8},E2[]={8,1,7};
177.      InitList(L1);
178.      InitList(L2);
179.      InitList(L3);
180.      CreaPolyList(L1,C1,E1,4);
181.      CreaPolyList(L2,C2,E2,3);
182.      printf("两多项式相加运算\n");
183.      printf("      原多项式A:");DispList(L1);
184.      printf("      原多项式B:");DispList(L2);
185.      SortPloy(L1);
186.      SortPloy(L2);
187.      printf("排序后的多项式A:");DispList(L1);
188.      printf("排序后的多项式B:");DispList(L2);
189.      L3=AddPoly(L1,L2);
190.      printf("多项式相加结果:");DispList(L3);
191.  }
```

栈的基本概念

栈的定义

栈是一种特殊的线性表，插入和删除操作在表的某一端进行，允许插入和删除操作的一端称为 `栈顶` ，另一端称为 `栈底` 。

可以把栈看作一个竖直的桶，每次只能放入一个元素，先放入的元素在下，后放入的元素在上，后放入的元素先出。

栈的特征

- 后进先出（LIFO）

栈的基本运算

- 初始化栈
- 进栈
- 出栈，即返回栈顶元素，并删除当前的栈顶元素
- 取栈顶元素
- 判断栈空

栈的存储结构

栈的结构定义主要包含以下属性

- data: 一维数组，用于保存栈中的元素
    - StackSize: 栈的大小，即数组的大小
    - ElemType: 元素的类型
- top: 栈顶的指针

顺序存储结构

```
1.  typedef char ElemType;
2.
3.  #define StackSize 100        /*顺序栈的初始分配空间*/
```

```
4.
5.  typedef struct
6.  {
7.      ElemType data[StackSize];    /*保存栈中元素*/
8.      int top;                    /*栈指针*/
9.  } SqStack;
```

链式存储结构

```
1.  typedef char ElemType;
2.
3.  typedef struct lsnode
4.  {
5.      ElemType data;            /*存储结点数据*/
6.      struct lsnode *next;    /*指针域*/
7.  } LinkStack;
```

顺序栈的基本运算

栈的定义

```
#include <stdio.h>

typedef char ElemType;

#define StackSize 100        /*顺序栈的初始分配空间*/

typedef struct
{
    ElemType data[StackSize];    /*保存栈中元素*/
    int top;                    /*栈指针*/
} SqStack;
```

初始化栈

```
void InitStack(SqStack &st)        /*st为引用型参数*/
{
    st.top=-1;
}
```

进栈运算

```
int Push(SqStack &st,ElemType x)    /*进栈运算,st为引用型参数*/
{
    if (st.top==StackSize-1)    /*栈满*/
        return 0;
    else                        /*栈不满*/
    {
        st.top++;
        st.data[st.top]=x;
        return 1;
    }
}
```

出栈运算

```
int Pop(SqStack &st,ElemType &x)        /*出栈运算,st和x为引用型参数*/
{
    if (st.top==-1)        /*栈空*/
        return 0;
    else                /*栈不空*/
    {
        x=st.data[st.top];
        st.top--;
        return 1;
    }
}
```

取栈顶元素

```
int GetTop(SqStack st,ElemType &x)    /*取栈顶元素,x为引用型参数*/
{
    if (st.top==-1)        /*栈空*/
        return 0;
    else
    {
        x=st.data[st.top];
        return 1;
    }
}
```

判断栈空运算

```
int StackEmpty(SqStack st)    /*判断栈空运算*/
{
    if (st.top==-1)    /*栈空*/
        return 1;
    else            /*栈不空*/
        return 0;
}
```

main

```
void main()
{
    SqStack st;
    ElemType e;
    InitStack(st);
    printf("栈%s\n",(StackEmpty(st)==1?"空":"不空"));
    printf("a进栈\n");Push(st,'a');
    printf("b进栈\n");Push(st,'b');
    printf("c进栈\n");Push(st,'c');
    printf("d进栈\n");Push(st,'d');
    printf("栈%s\n",(StackEmpty(st)==1?"空":"不空"));
    GetTop(st,e);
    printf("栈顶元素:%c\n",e);
    printf("出栈次序:");
    while (!StackEmpty(st))
    {
        Pop(st,e);
        printf("%c ",e);
    }
    printf("\n");
}
```

链栈的基本运算

链式栈的定义

```
#include <malloc.h>

typedef char ElemType;

typedef struct lsnode
{
    ElemType data;            /*存储结点数据*/
    struct lsnode *next;    /*指针域*/
} LinkStack;
```

初始化栈

```
void InitStack(LinkStack *&ls)        /*ls为引用型参数*/
{
    ls=NULL;
}
```

进栈运算

```
void Push(LinkStack *&ls,ElemType x)        /*进栈运算,ls为引用型参数*/
{
    LinkStack *p;
    p=(LinkStack *)malloc(sizeof(LinkStack));
    p->data=x;
    p->next=ls;
    ls=p;
}
```

出栈运算

```
int Pop(LinkStack *&ls,ElemType &x)        /*出栈运算,ls为引用型参数*/
{
```

```
    LinkStack *p;
    if (ls==NULL)     /*栈空,下溢出*/
        return 0;
    else
    {
        p=ls;
        x=p->data;
        ls=p->next;
        free(p);
           return 1;
    }
}
```

取栈顶元素运算

```
int GetTop(LinkStack *ls,ElemType &x)    /*取栈顶元素运算*/
{
    if (ls==NULL)     /*栈空,下溢出*/
        return 0;
    else
    {
        x=ls->data;
           return 1;
       }
}
```

判断栈空运算

```
int StackEmpty(LinkStack *ls)    /*判断栈空运算*/
{
    if (ls==NULL)
        return 1;
    else
        return 0;
}
```

main

```
1.  void main()
2.  {
3.      LinkStack *ls;
4.      ElemType e;
5.      InitStack(ls);
6.      printf("栈%s\n",(StackEmpty(ls)==1?"空":"不空"));
7.      printf("a进栈\n");Push(ls,'a');
8.      printf("b进栈\n");Push(ls,'b');
9.      printf("c进栈\n");Push(ls,'c');
10.     printf("d进栈\n");Push(ls,'d');
11.     printf("栈%s\n",(StackEmpty(ls)==1?"空":"不空"));
12.     GetTop(ls,e);
13.     printf("栈顶元素:%c\n",e);
14.     printf("出栈次序:");
15.     while (!StackEmpty(ls))
16.     {
17.         Pop(ls,e);
18.         printf("%c ",e);
19.     }
20.     printf("\n");
21. }
```

By Golang

```go
// Package stack creates a ItemStack data structure for the Item type
package stack

import (
    "sync"
)

// Item the type of the stack
type Item interface{}

// ItemStack the stack of Items
type ItemStack struct {
    items []Item
    lock  sync.RWMutex
}

// New creates a new ItemStack
func (s *ItemStack) New() *ItemStack {
    s.items = []Item{}
    return s
}

// Push adds an Item to the top of the stack
func (s *ItemStack) Push(t Item) {
    s.lock.Lock()
    s.items = append(s.items, t)
    s.lock.Unlock()
}

// Pop removes an Item from the top of the stack
func (s *ItemStack) Pop() *Item {
    s.lock.Lock()
    item := s.items[len(s.items)-1]
    s.items = s.items[0 : len(s.items)-1]
    s.lock.Unlock()
    return &item
}
```

队列的基本概念

队列的定义

队列也是一种特殊的线性表，插入操作在一端进行，称为队头，删除操作在另一端进行，称为队尾。

队列就跟生活中的排队类似，不允许插队，先排的人可以先得到服务。

队列的特征

- 先进先出（FIFO）

队列的基本运算

- 初始化队列
- 入队
- 出队
- 取队头的元素
- 判断队列是否为空

队列的存储结构

顺序存储结构

```
#define QueueSize 100

typedef char ElemType;

typedef struct
{
    ElemType data[QueueSize];    /*保存队中元素*/
    int front,rear;                /*队头和队尾指针*/
} SqQueue;
```

链式存储结构

```
typedef char ElemType;
```

```
2.
3.  typedef struct QNode
4.  {
5.      ElemType data;
6.      struct QNode *next;
7.  } QType;                /*链队中结点的类型*/
8.
9.  typedef struct qptr
10. {
11.     QType *front,*rear;
12. } LinkQueue;            /*链队类型*/
```

顺序队列的基本运算

顺序队列的定义

```
#include <stdio.h>
#define QueueSize 100

typedef char ElemType;

typedef struct
{
    ElemType data[QueueSize];    /*保存队中元素*/
    int front,rear;                /*队头和队尾指针*/
} SqQueue;
```

初始化队列

```
void InitQueue(SqQueue &qu)        /*qu为引用型参数*/
{
       qu.rear=qu.front=0;           /*指针初始化*/
}
```

入队运算

```
int EnQueue(SqQueue &qu,ElemType x)    /*入队运算,qu为引用型参数*/
{
    if ((qu.rear+1)%QueueSize==qu.front)    /*队满*/
        return 0;
    qu.rear=(qu.rear+1)%QueueSize;      /*队尾指针进1*/
    qu.data[qu.rear]=x;
    return 1;
}
```

出队运算

```
int DeQueue(SqQueue &qu,ElemType &x)    /*出队运算,qu和x为引用型参数*/
```

```
{
    if (qu.rear==qu.front)
        return 0;
    qu.front=(qu.front+1)%QueueSize;    /*队头指针进1*/
    x=qu.data[qu.front];
    return 1;
}
```

取队头元素运算

```
int GetHead(SqQueue qu,ElemType &x)        /*取队头元素运算,x为引用型参数*/
{
    if (qu.rear==qu.front)        /*队空*/
        return 0;
    x=qu.data[(qu.front+1)%QueueSize];
    return 1;
}
```

判断队空运算

```
int QueueEmpty(SqQueue qu)        /*判断队空运算*/
{
    if (qu.rear==qu.front)        /*队空*/
        return 1;
    else
        return 0;
}
```

main

```
void main()
{
    SqQueue qu;
    ElemType e;
    InitQueue(qu);
    printf("队%s\n",(QueueEmpty(qu)==1?"空":"不空"));
    printf("a进队\n");EnQueue(qu,'a');
    printf("b进队\n");EnQueue(qu,'b');
```

```
    printf("c进队\n");EnQueue(qu,'c');
    printf("d进队\n");EnQueue(qu,'d');
    printf("队%s\n",(QueueEmpty(qu)==1?"空":"不空"));
    GetHead(qu,e);
    printf("队头元素:%c\n",e);
    printf("出队次序:");
    while (!QueueEmpty(qu))
    {
        DeQueue(qu,e);
        printf("%c ",e);
    }
    printf("\n");
}
```

链式队列的基本运算

链式队列的定义

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

typedef char ElemType;

typedef struct QNode
{
    ElemType data;
    struct QNode *next;
} QType;                /*链队中结点的类型*/

typedef struct qptr
{
    QType *front,*rear;
} LinkQueue;            /*链队类型*/
```

初始化队列

```
void InitQueue(LinkQueue *&lq)            /*lq为引用型参数*/
{
    lq=(LinkQueue *)malloc(sizeof(LinkQueue));
    lq->rear=lq->front=NULL;             /*初始情况*/
}
```

入队运算

```
void EnQueue(LinkQueue *&lq,ElemType x)    /*入队运算,lq为引用型参数*/
{
    QType *s;
    s=(QType *)malloc(sizeof(QType));      /*创建新结点，插入到链队的末尾*/
    s->data=x;s->next=NULL;
    if (lq->front==NULL && lq->rear==NULL)     /*空队*/
        lq->rear=lq->front=s;
```

```
    else
    {
        lq->rear->next=s;
        lq->rear=s;
    }
}
```

出队运算

```
int DeQueue(LinkQueue *&lq,ElemType &x)     /*出队运算,lq和x均为引用型参数*/
{
    QType *p;
    if (lq->front==NULL && lq->rear==NULL)  /*空队*/
        return 0;
    p=lq->front;
    x=p->data;
    if (lq->rear==lq->front)  /*若原队列中只有一个结点,删除后队列变空*/
        lq->rear=lq->front=NULL;
    else
        lq->front=lq->front->next;
    free(p);
    return 1;
}
```

取队头元素运算

```
int GetHead(LinkQueue *lq,ElemType &x)     /*取队头元素运算,x为引用型参数*/
{
    if (lq->front==NULL && lq->rear==NULL)    /*队空*/
        return 0;
    x=lq->front->data;
    return 1;
}
```

判断队空运算

```
int QueueEmpty(LinkQueue *lq)    /*判断队空运算*/
{
```

```
    if (lq->front==NULL && lq->rear==NULL)
        return 1;
    else
        return 0;
}
```

main

```
void main()
{
    LinkQueue *lq;
    ElemType e;
    InitQueue(lq);
    printf("队%s\n",(QueueEmpty(lq)==1?"空":"不空"));
    printf("a进队\n");EnQueue(lq,'a');
    printf("b进队\n");EnQueue(lq,'b');
    printf("c进队\n");EnQueue(lq,'c');
    printf("d进队\n");EnQueue(lq,'d');
    printf("队%s\n",(QueueEmpty(lq)==1?"空":"不空"));
    GetHead(lq,e);
    printf("队头元素:%c\n",e);
    printf("出队次序:");
    while (!QueueEmpty(lq))
    {
        DeQueue(lq,e);
        printf("%c ",e);
    }
    printf("\n");
}
```

By Golang

```go
// Package queue creates a ItemQueue data structure for the Item type
package queue

import (
    "sync"
)

// Item the type of the queue
type Item interface{}

// ItemQueue the queue of Items
type ItemQueue struct {
    items []Item
    lock  sync.RWMutex
}

// New creates a new ItemQueue
func (s *ItemQueue) New() *ItemQueue {
    s.lock.Lock()
    s.items = []Item{}
    s.lock.Unlock()
    return s
}

// Enqueue adds an Item to the end of the queue
func (s *ItemQueue) Enqueue(t Item) {
    s.lock.Lock()
    s.items = append(s.items, t)
    s.lock.Unlock()
}

// Dequeue removes an Item from the start of the queue
func (s *ItemQueue) Dequeue() *Item {
    s.lock.Lock()
    item := s.items[0]
    s.items = s.items[1:len(s.items)]
    s.lock.Unlock()
    return &item
```

```
39. }
40.
41. // Front returns the item next in the queue, without removing it
42. func (s *ItemQueue) Front() *Item {
43.     s.lock.RLock()
44.     item := s.items[0]
45.     s.lock.RUnlock()
46.     return &item
47. }
48.
49. // IsEmpty returns true if the queue is empty
50. func (s *ItemQueue) IsEmpty() bool {
51.     s.lock.RLock()
52.     defer s.lock.RUnlock()
53.     return len(s.items) == 0
54. }
55.
56. // Size returns the number of Items in the queue
57. func (s *ItemQueue) Size() int {
58.     s.lock.RLock()
59.     defer s.lock.RUnlock()
60.     return len(s.items)
61. }
```

看病排队问题

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

typedef struct QNode
{
    char data[10];
    struct QNode *next;
} QType;        /*链队结点类型*/

typedef struct
{
    QType *front,*rear;
} LinkQueue;    /*链队类型*/

void SeeDoctor()
{
    int sel,flag=1;
    LinkQueue *lq;
    QType *s;
    char name[10];
    lq=(LinkQueue *)malloc(sizeof(LinkQueue));
    lq->front=(QType *)malloc(sizeof(QType));
    lq->front->next=NULL;
    lq->rear=lq->front;
    while (flag==1)     /*未下班时循环执行*/
    {
        printf("1:排队 2:看医生 3:查看排队 0:下班  请选择:");
        scanf("%d",&sel);
        switch(sel)
        {
        case 0:
            if (lq->front!=lq->rear)     /*队不空*/
                    printf("  >>请排队的患者明天就医\n");
            flag=0;
            break;
        case 1:
            printf("  >>输入患者姓名:");scanf("%s",name);
```

```
            s=(QType *)malloc(sizeof(QType));
            strcpy(s->data,name);s->next=NULL;
            lq->rear->next=s;lq->rear=s;
            break;
          case 2:
            if (lq->front==lq->rear)    /*队空*/
                printf("  >>没有排队的患者\n");
            else
            {
                s=lq->front->next;
                if (lq->rear==s)
                    lq->rear=lq->front;
                printf("  >>患者%s看医生\n",s->data);
                lq->front->next=s->next;
                free(s);
            }
            break;
          case 3:
            if (lq->front==lq->rear)    /*队空*/
                printf("  >>没有排列的患者\n");
            else
            {
                s=lq->front->next;
                printf("  >>排队患者:");
                while (s!=NULL)
                {
                    printf("%s ",s->data);
                    s=s->next;
                }
                printf("\n");
            }
            break;
        }
    }
}

void main()
{
    SeeDoctor();
}
```

- 串的基本概念
- 顺序串基本运算
- 链串基本运算

字符串的基本概念

字符串的定义

字符串的由零个或多个的字符组成的有限序列，一般表示为”a1a2…an”。

字符串的特征

- 串中字符的个数称为串的长度
- 任意连续的字符组成的子序列称为该串的子串
- 子串的位置为子串第一个字符在原串中的位置

字符串的基本运算

- 串的赋值
- 串的复制
- 求串的长度
- 判断两个串是否相等
- 串的拼接
- 求子串
- 查找子串的位置
- 插入子串
- 删除子串
- 替换子串
- 输出串

字符串的存储结构

顺序存储结构

```
#define MaxSize 100  /*最多字符个数*/

typedef struct
{
    char ch[MaxSize];    /*存放串字符*/
       int len;            /*存放串的实际长度*/
} SqString;               /*顺序串类型*/
```

链式存储结构

```
typedef struct node
{
    char data;            /*存放字符*/
    struct node *next;    /*指针域*/
} LinkString;
```

顺序串的基本运算

顺序串的定义

```
#include <stdio.h>
#define MaxSize 100  /*最多字符个数*/

typedef struct
{
    char ch[MaxSize];    /*存放串字符*/
       int len;            /*存放串的实际长度*/
} SqString;               /*顺序串类型*/
```

赋值运算

```
void Assign(SqString &s,char t[])    /*串赋值运算*/
{
    int i=0;
    while (t[i]!='\0')
    {
        s.ch[i]=t[i];
        i++;
    }
    s.len=i;
}
```

复制运算

```
void StrCopy(SqString &s,SqString t)    /*串复制运算*/
{
    int i;
    for (i=0;i<t.len;i++)
        s.ch[i]=t.ch[i];
    s.len=t.len;
}
```

求串长运算

```
int StrLength(SqString s)    /*求串长运算*/
{
    return(s.len);
}
```

判断串相等运算

```
int StrEqual(SqString s,SqString t)    /*判断串相等运算*/
{
    int i=0;
    if (s.len!=t.len)        /*串长不同时返回0*/
        return(0);
    else
    {
        for (i=0;i<s.len;i++)
            if (s.ch[i]!=t.ch[i]) /*有一个对应字符不同时返回0*/
                return(0);
        return(1);
    }
}
```

串连接运算

```
SqString Concat(SqString s,SqString t)    /*串连接运算*/
{
    SqString r;
    int i,j;
    for (i=0;i<s.len;i++)        /*将s复制到r*/
        r.ch[i]=s.ch[i];
    for (j=0;j<t.len;j++)        /*将t复制到r*/
        r.ch[s.len+j]=t.ch[j];
    r.len=i+j;
    return(r);                    /*返回r*/
}
```

求子串运算

```
SqString SubStr(SqString s,int i,int j)    /*求子串运算*/
{
    SqString t;
    int k;
    if (i<1 || i>s.len || j<1 || i+j>s.len+1)
        t.len=0;            /*参数错误时返回空串*/
    else
    {
        for (k=i-1;k<i+j;k++)
            t.ch[k-i+1]=s.ch[k];
        t.len=j;
    }
    return(t);
}
```

查找子串位置运算

```
int Index(SqString s,SqString t)    /*查找子串位置运算*/
{
    int i=0,j=0,k;                /*i和j分别扫描主串s和子串t*/
    while (i<s.len && j<t.len)
    {
        if (s.ch[i]==t.ch[j]) /*对应字符相同时,继续比较下一对字符*/
        {
            i++;j++;
        }
        else                /*否则,主子串指针回溯重新开始下一次匹配*/
        {
            i=i-j+1;j=0;
        }
    }
    if (j>=t.len)
        k=i-t.len+1;/*求出第一个字符的位置*/
    else
        k=-1;        /*置特殊值-1*/
    return(k);
}
```

子串插入运算

```
int InsStr(SqString &s,int i,SqString t)    /*子串插入运算*/
{
    int j;
    if (i>s.len+1)
        return(0);                    /*位置参数值错误*/
    else
    {
        for (j=s.len;j>=i-1;j--)     /*将s.ch[i-1]-s.ch[s.len-1]*/
            s.ch[j+t.len]=s.ch[j];    /*后移t.len个位置*/
        for (j=0;j<t.len;j++)
            s.ch[i+j-1]=t.ch[j];
        s.len=s.len+t.len;             /*修改s串长度*/
        return(1);
    }
}
```

子串删除运算

```
int DelStr(SqString &s,int i,int j)    /*子串删除运算*/
{
    int k;
    if (i<1 || i>s.len || j<1 || i+j>s.len+1)
        return(0);        /*位置参数值错误*/
    else
    {
        for (k=i+j-1;k<s.len;k++)     /*将s的第i+j位置之后的字符前移j位*/
            s.ch[k-j]=s.ch[k];
        s.len=s.len-j;                /*修改s的长度*/
        return(1);
    }
}
```

子串替换运算

```
SqString RepStrAll(SqString s,SqString s1,SqString s2)    /*子串替换运算*/
{
    int i;
    i=Index(s,s1);
    while (i>=0)
```

```
    {
        DelStr(s,i,s1.len);    /*删除*/
        InsStr(s,i,s2);        /*插入*/
        i=Index(s,s1);
    }
    return(s);
}
```

输出串运算

```
void DispStr(SqString s)    /*输出串运算*/
{
    int i;
    for (i=0;i<s.len;i++)
        printf("%c",s.ch[i]);
    printf("\n");
}
```

main

```
void main()
{
    SqString s1,s2,s3,s4,s5,s6,s7;
    Assign(s1,"abcd");
    printf("s1:");DispStr(s1);
    printf("s1的长度:%d\n",StrLength(s1));
    printf("s1=>s2\n");
    StrCopy(s2,s1);
    printf("s2:");DispStr(s2);
    printf("s1和s2%s\n",(StrEqual(s1,s2)==1?"相同":"不相同"));
    Assign(s3,"12345678");
    printf("s3:");DispStr(s3);
    printf("s1和s3连接=>s4\n");
    s4=Concat(s1,s3);
    printf("s4:");DispStr(s4);
    printf("s3[2..5]=>s5\n");
    s5=SubStr(s3,2,4);
    printf("s5:");DispStr(s5);
    Assign(s6,"567");
    printf("s6:");DispStr(s6);
```

```
21.     printf("s6在s3中位置:%d\n",Index(s3,s6));
22.     printf("从s3中删除s3[3..6]字符\n");
23.     DelStr(s3,3,4);
24.     printf("s3:");DispStr(s3);
25.     printf("从s4中将s6替换成s1=>s7\n");
26.     s7=RepStrAll(s4,s6,s1);
27.     printf("s7:");DispStr(s7);
28. }
```

链串的基本运算

链串的定义

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

typedef struct node
{
    char data;            /*存放字符*/
    struct node *next;    /*指针域*/
} LinkString;
```

赋值运算

```
void Assign(LinkString *&s,char t[])
{
    int i=0;
    LinkString *q,*tc;
    s=(LinkString *)malloc(sizeof(LinkString)); /*建立头结点*/
    s->next=NULL;
    tc=s;                    /*tc指向s串的尾结点*/
    while (t[i]!='\0')
    {
        q=(LinkString *)malloc(sizeof(LinkString));
        q->data=t[i];
        tc->next=q;tc=q;
        i++;
    }
    tc->next=NULL;        /*终端结点的next置NULL*/
}
```

复制运算

```
void StrCopy(LinkString *&s,LinkString *t)    /*t=>s*/
{
    LinkString *p=t->next,*q,*tc;
```

```
    s=(LinkString *)malloc(sizeof(LinkString)); /*建立头结点*/
    s->next=NULL;
    tc=s;                /*tc指向s串的尾结点*/
    while (p!=NULL)        /*复制t的所有结点*/
    {
        q=(LinkString *)malloc(sizeof(LinkString));
        q->data=p->data;
        tc->next=q;tc=q;
        p=p->next;
    }
    tc->next=NULL;        /*终端结点的next置NULL*/
}
```

求串长运算

```
int StrLength(LinkString *s)
{
    int n=0;
    LinkString *p=s->next;
    while (p!=NULL)        /*扫描串s的所有结点*/
    {
        n++;p=p->next;
    }
    return(n);
}
```

判断串相等运算

```
int StrEqual(LinkString *s,LinkString *t)
{
    LinkString *p=s->next,*q=t->next;
    while (p!=NULL && q!=NULL)    /*比较两串的当前结点*/
    {
        if (p->data!=q->data)    /*data域不等时返回0*/
            return(0);
        p=p->next;q=q->next;
    }
    if (p!=NULL || q!=NULL)        /*两串长度不等时返回0*/
        return(0);
    return(1);
```

```
}
```

串连接运算

```
LinkString *Concat(LinkString *s,LinkString *t)
{
    LinkString *p=s->next,*q,*tc,*str;
    str=(LinkString *)malloc(sizeof(LinkString));  /*建立头结点*/
    str->next=NULL;
    tc=str;                 /*tc总是指向新链表的尾结点*/
    while (p!=NULL)         /*将s串复制给str*/
    {
        q=(LinkString *)malloc(sizeof(LinkString));
        q->data=p->data;
        tc->next=q;tc=q;
        p=p->next;
    }
    p=t->next;
    while (p!=NULL)         /*将t串复制给str*/
    {
        q=(LinkString *)malloc(sizeof(LinkString));
        q->data=p->data;
        tc->next=q;tc=q;
        p=p->next;
    }
    tc->next=NULL;
    return(str);
}
```

求子串运算

```
LinkString *SubStr(LinkString *s,int i,int j)
{
    int k=1;
    LinkString *p=s->next,*q,*tc,*str;
    str=(LinkString *)malloc(sizeof(LinkString));  /*建立头结点*/
    str->next=NULL;
    tc=str;                             /*tc总是指向新链表的尾结点*/
    while (k<i && p!=NULL)
    {
```

```
        p=p->next;k++;
    }
    if (p!=NULL)
    {
        k=1;
        while (k<=j && p!=NULL)        /*复制j个结点*/
        {
            q=(LinkString *)malloc(sizeof(LinkString));
            q->data=p->data;
            tc->next=q;tc=q;
            p=p->next;
            k++;
        }
        tc->next=NULL;
    }
    return(str);
}
```

查找子串位置运算

```
int Index(LinkString *s,LinkString *t)
{
    LinkString *p=s->next,*p1,*q,*q1;
    int i=0;
    while (p!=NULL)           /*循环扫描s的每个结点*/
    {
        q=t->next;            /*子串总是从第一个字符开始比较*/
        if (p->data==q->data)/*判定两串当前字符相等*/
        {            /*若首字符相同,则判定s其后字符是否与t的依次相同*/
            p1=p->next;q1=q->next;
            while (p1!=NULL && q1!=NULL && p1->data==q1->data)
            {
                p1=p1->next;
                q1=q1->next;
            }
            if (q1==NULL)    /*若都相同,则返回相同的子串的起始位置*/
                return(i);
        }
        p=p->next;i++;
    }
    return(-1);        /*若不是子串,返回-1*/
```

```
}
```

子串插入运算

```
int InsStr(LinkString *&s,int i,LinkString *t)
{
    int k;
    LinkString *q=s->next,*p,*str;
    StrCopy(str,t);        /*将t复制到str*/
    p=str;str=str->next;
    free(p);            /*删除str的头结点*/
    for (k=1;k<i;k++)    /*在s中找到第i-1个结点,由p指向它,q指向下一个结点*/
    {
        if (q==NULL)    /*位置参数i错误*/
            return(0);
        p=q;
        q=q->next;
    }
    p->next=str;        /*将str链表链接到*p之后*/
    while (str->next!=NULL)    /*由str指向尾结点*/
        str=str->next;
    str->next=q;        /*将*q链接到*str之后*/
    return(1);
}
```

子串删除运算

```
int DelStr(LinkString *&s,int i,int j)
{
    int k;
    LinkString *q=s->next,*p,*t;
    for (k=1;k<i;k++)    /*在s中找到第i-1个结点,由p指向它,q指向下一个结点*/
    {
        if (q==NULL)        /*位置参数i错误*/
            return(0);
        p=q;
        q=q->next;
    }
    for (k=1;k<=j;k++)    /*删除*p之后的j个结点,并由q指向下一个结点*/
    {
```

```
        if (q==NULL)    /*长度参数j错误*/
            return(0);
        t=q;
        q=q->next;
        free(t);
    }
    p->next=q;
    return(1);
}
```

子串替换运算

```
LinkString *RepStrAll(LinkString *s,LinkString *s1,LinkString *s2)
{
    int i;
    i=Index(s,s1);
    while (i>=0)
    {
        DelStr(s,i+1,StrLength(s1));    /*删除串s1*/
        InsStr(s,i+1,s2);               /*插入串s2*/
        i=Index(s,s1);
    }
    return(s);
}
```

输出串运算

```
void DispStr(LinkString *s)
{
    LinkString *p=s->next;
    while (p!=NULL)
    {
        printf("%c",p->data);
        p=p->next;
    }
    printf("\n");
}
```

main

```
void main()
{
    LinkString *s1,*s2,*s3,*s4,*s5,*s6,*s7;
    Assign(s1,"abcd");
    printf("s1:");DispStr(s1);
    printf("s1的长度:%d\n",StrLength(s1));
    printf("s1=>s2\n");
    StrCopy(s2,s1);
    printf("s2:");DispStr(s2);
    printf("s1和s2%s\n",(StrEqual(s1,s2)==1?"相同":"不相同"));
    Assign(s3,"12345678");
    printf("s3:");DispStr(s3);
    printf("s1和s3连接=>s4\n");
    s4=Concat(s1,s3);
    printf("s4:");DispStr(s4);
    printf("s3[2..5]=>s5\n");
    s5=SubStr(s3,2,4);
    printf("s5:");DispStr(s5);
    Assign(s6,"567");
    printf("s6:");DispStr(s6);
    printf("s6在s3中位置:%d\n",Index(s3,s6));
    printf("从s3中删除s3[3..6]字符\n");
    DelStr(s3,3,4);
    printf("s3:");DispStr(s3);
    printf("从s4中将s6替换成s1=>s7\n");
    s7=RepStrAll(s4,s6,s1);
    printf("s7:");DispStr(s7);
}
```

- 二叉树的基本概念
- 二叉树基本运算
- 二叉树基本运算(By-Go)
- 二叉树4种遍历算法
- 哈夫曼树

二叉树的基本概念

二叉树的定义

树形结构是一种非线性结构，二叉树是度为2，即子结点的个数最多为2的有序树（左右子树是有次序的）。最重要，应用最广泛的一种树。

完全二叉树

在一个二叉树中，除了最后一层外，其余的其他层都是满的，并且最后一层或者是满的，或者是在右边缺少连续若干个结点，则该树称为 `完全二叉树` 。

满二叉树是一种特殊的完全二叉树，即所有的层的结点都是满的。

树的基本术语

- 结点的度：该节点的后继节点的个数
- 树的度：所有节点的度的最大值
- 分支结点
- 叶子结点
- 孩子结点
- 双亲结点
- 子孙结点
- 祖先结点
- 兄弟结点
- 结点层数
- 树的深度（高度）：树中结点的最大的层数
- 有序树：左右子树是有次序的
- 无序树：左右子树是无次序的
- 森林：不同树的集合

二叉树的性质

- 二叉树上叶子节点的个数等于度为2的结点的个数加1
- 二叉树上第i层上至多有2^(i-1)个结点（i>1）
- 深度为h的二叉树至多有2^h-1个结点

二叉树的基本运算

- 创建二叉树
- 求二叉树的高度
- 求二叉树结点的个数
- 求二叉树叶子结点的个数
- 用括号表示法输出二叉树
- 用凹入表示法输出二叉树

二叉树的存储结构

顺序存储结构

```
typedef ElemType SqBinTree[MaxSize]
```

链式存储结构

```
#define MaxSize 100
#define MaxWidth 40

typedef char ElemType;

typedef struct tnode
{
    ElemType data;
    struct tnode *lchild,*rchild;
} BTNode;
```

二叉树的基本运算

二叉树的定义

```
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100
#define MaxWidth 40

typedef char ElemType;

typedef struct tnode
{
    ElemType data;
    struct tnode *lchild,*rchild;
} BTNode;
```

由str创建二叉链

```
void CreateBTree(BTNode * &bt,char *str)    /*由str创建二叉链bt*/
{
    BTNode *St[MaxSize],*p=NULL;
    int top=-1,k,j=0;
    char ch;
    bt=NULL;            /*建立的二叉树初始时为空*/
    ch=str[j];
    while (ch!='\0')      /*str未扫描完时循环*/
    {
              switch(ch)
        {
        case '(':top++;St[top]=p;k=1; break;    /*为左孩子结点*/
        case ')':top--;break;
        case ',':k=2; break;                    /*为孩子结点右结点*/
        default:p=(BTNode *)malloc(sizeof(BTNode));
                p->data=ch;p->lchild=p->rchild=NULL;
                if (bt==NULL)                   /**p为二叉树的根结点*/
                    bt=p;
                else                              /*已建立二叉树根结点*/
```

```
                {    switch(k)
                    {
                    case 1:St[top]->lchild=p;break;
                    case 2:St[top]->rchild=p;break;
                    }
                }
        }
        j++;
        ch=str[j];
    }
}
```

求二叉树高度

```
int BTHeight(BTNode *bt)    /*求二叉树高度*/
{
       int lchilddep,rchilddep;
       if (bt==NULL) return(0);             /*空树的高度为0*/
       else
    {    lchilddep=BTHeight(bt->lchild);    /*求左子树的高度为lchilddep*/
          rchilddep=BTHeight(bt->rchild);    /*求右子树的高度为rchilddep*/
        return (lchilddep>rchilddep)? (lchilddep+1):(rchilddep+1);
       }
}
```

求二叉树的结点个数

```
int NodeCount(BTNode *bt)    /*求二叉树bt的结点个数*/
{
    int num1,num2;
        if (bt==NULL)    /*空树结点个数为0*/
            return 0;
        else
        {    num1=NodeCount(bt->lchild);    /*求出左子树的结点数*/
            num2=NodeCount(bt->rchild);    /*求出右子树的结点数*/
            return (num1+num2+1);
    }
}
```

求二叉树的叶子结点个数

```
int LeafCount(BTNode *bt)    /*求二叉树bt的叶子结点个数*/
{
    int num1,num2;
        if (bt==NULL)    /*空树叶子结点个数为0*/
            return 0;
        else if (bt->lchild==NULL && bt->rchild==NULL)
            return 1;    /*若为叶子结点返回1*/
        else
        {    num1=LeafCount(bt->lchild);    /*求出左子树的叶子结点数*/
            num2=LeafCount(bt->rchild);    /*求出右子树的叶子结点数*/
            return (num1+num2);
    }
}
```

以括号表示法输出二叉树

```
void DispBTree(BTNode *bt)    /*以括号表示法输出二叉树*/
{
    if (bt!=NULL)
    {
        printf("%c",bt->data);
        if (bt->lchild!=NULL || bt->rchild!=NULL)
        {
            printf("(");
            DispBTree(bt->lchild);        /*递归处理左子树*/
            if (bt->rchild!=NULL)
                printf(",");
            DispBTree(bt->rchild);        /*递归处理右子树*/
            printf(")");
        }
    }
}
```

以凹入表示法输出一棵二叉树

```
void DispBTree1(BTNode *bt)  /*以凹入表示法输出一棵二叉树*/
{
```

```
    BTNode *St[MaxSize],*p;
    int Level[MaxSize][2],top=-1,n,i,width=4;
    char type;        /*取值L表示为左结点,R表示为右结点,B表示为根结点*/
    if (bt!=NULL)
    {
        top++;
        St[top]=bt;                          /*根结点入栈*/
        Level[top][0]=width;
        Level[top][1]=2;                  /*2表示是根*/
        while (top>-1)
        {
            p=St[top];                       /*退栈并凹入显示该结点值*/
            n=Level[top][0];
            switch(Level[top][1])
            {
            case 0:type='L';break;           /*左结点之后输出(L)*/
            case 1:type='R';break;           /*右结点之后输出(R)*/
            case 2:type='B';break;           /*根结点之后前输出(B)*/
            }
            for (i=1;i<=n;i++)               /*其中n为显示场宽,字符以右对齐显示*/
                printf(" ");
            printf("%c(%c)",p->data,type);
            for (i=n+1;i<=MaxWidth;i+=2)
                printf("—");
            printf("\n");
            top--;
            if (p->rchild!=NULL)
            {                                /*将右子树根结点入栈*/
                top++;
                St[top]=p->rchild;
                Level[top][0]=n+width;       /*场宽增width,即缩width格后再输出*/
                Level[top][1]=1;          /*1表示是右子树*/
            }
            if (p->lchild!=NULL)
            {                                /*将左子树根结点入栈*/
                top++;
                St[top]=p->lchild;
                Level[top][0]=n+width;    /*显示场宽增width*/
                Level[top][1]=0;          /*0表示是左子树*/
            }
        }
    }
```

```
}
```

main

```
void main()
{
    BTNode *bt;
    CreateBTree(bt,"A(B(D,E(G,H)),C(,F(I)))");    /*构造图5.10(a)所示的二叉树*/
    printf("二叉树bt:");DispBTree(bt);printf("\n");
    printf("bt的高度:%d\n",BTHeight(bt));
    printf("bt的结点数:%d\n",NodeCount(bt));
    printf("bt的叶子结点数:%d\n",LeafCount(bt));
    printf("bt凹入表示:\n");DispBTree1(bt);printf("\n");
}
```

By Golang

```
    // Package binarysearchtree creates a ItemBinarySearchTree data structure for
 1. the Item type
 2. package binarysearchtree
 3.
 4. import (
 5.     "fmt"
 6.     "sync"
 7. )
 8.
 9. // Item the type of the binary search tree
10. type Item interface{}
11.
12. // Node a single node that composes the tree
13. type Node struct {
14.     key   int
15.     value Item
16.     left  *Node //left
17.     right *Node //right
18. }
19.
20. // ItemBinarySearchTree the binary search tree of Items
21. type ItemBinarySearchTree struct {
22.     root *Node
23.     lock sync.RWMutex
24. }
25.
26. // Insert inserts the Item t in the tree
27. func (bst *ItemBinarySearchTree) Insert(key int, value Item) {
28.     bst.lock.Lock()
29.     defer bst.lock.Unlock()
30.     n := &Node{key, value, nil, nil}
31.     if bst.root == nil {
32.         bst.root = n
33.     } else {
34.         insertNode(bst.root, n)
35.     }
36. }
37.
38. // internal function to find the correct place for a node in a tree
```

```
39. func insertNode(node, newNode *Node) {
40.     if newNode.key < node.key {
41.         if node.left == nil {
42.             node.left = newNode
43.         } else {
44.             insertNode(node.left, newNode)
45.         }
46.     } else {
47.         if node.right == nil {
48.             node.right = newNode
49.         } else {
50.             insertNode(node.right, newNode)
51.         }
52.     }
53. }
54.
55. // InOrderTraverse visits all nodes with in-order traversing
56. func (bst *ItemBinarySearchTree) InOrderTraverse(f func(Item)) {
57.     bst.lock.RLock()
58.     defer bst.lock.RUnlock()
59.     inOrderTraverse(bst.root, f)
60. }
61.
62. // internal recursive function to traverse in order
63. func inOrderTraverse(n *Node, f func(Item)) {
64.     if n != nil {
65.         inOrderTraverse(n.left, f)
66.         f(n.value)
67.         inOrderTraverse(n.right, f)
68.     }
69. }
70.
71. // PreOrderTraverse visits all nodes with pre-order traversing
72. func (bst *ItemBinarySearchTree) PreOrderTraverse(f func(Item)) {
73.     bst.lock.Lock()
74.     defer bst.lock.Unlock()
75.     preOrderTraverse(bst.root, f)
76. }
77.
78. // internal recursive function to traverse pre order
79. func preOrderTraverse(n *Node, f func(Item)) {
80.     if n != nil {
```

```
81.         f(n.value)
82.         preOrderTraverse(n.left, f)
83.         preOrderTraverse(n.right, f)
84.     }
85. }
86.
87. // PostOrderTraverse visits all nodes with post-order traversing
88. func (bst *ItemBinarySearchTree) PostOrderTraverse(f func(Item)) {
89.     bst.lock.Lock()
90.     defer bst.lock.Unlock()
91.     postOrderTraverse(bst.root, f)
92. }
93.
94. // internal recursive function to traverse post order
95. func postOrderTraverse(n *Node, f func(Item)) {
96.     if n != nil {
97.         postOrderTraverse(n.left, f)
98.         postOrderTraverse(n.right, f)
99.         f(n.value)
100.    }
101. }
102.
103. // Min returns the Item with min value stored in the tree
104. func (bst *ItemBinarySearchTree) Min() *Item {
105.     bst.lock.RLock()
106.     defer bst.lock.RUnlock()
107.     n := bst.root
108.     if n == nil {
109.         return nil
110.     }
111.     for {
112.         if n.left == nil {
113.             return &n.value
114.         }
115.         n = n.left
116.     }
117. }
118.
119. // Max returns the Item with max value stored in the tree
120. func (bst *ItemBinarySearchTree) Max() *Item {
121.     bst.lock.RLock()
122.     defer bst.lock.RUnlock()
```

```
123.      n := bst.root
124.      if n == nil {
125.          return nil
126.      }
127.      for {
128.          if n.right == nil {
129.              return &n.value
130.          }
131.          n = n.right
132.      }
133.  }
134.
135.  // Search returns true if the Item t exists in the tree
136.  func (bst *ItemBinarySearchTree) Search(key int) bool {
137.      bst.lock.RLock()
138.      defer bst.lock.RUnlock()
139.      return search(bst.root, key)
140.  }
141.
142.  // internal recursive function to search an item in the tree
143.  func search(n *Node, key int) bool {
144.      if n == nil {
145.          return false
146.      }
147.      if key < n.key {
148.          return search(n.left, key)
149.      }
150.      if key > n.key {
151.          return search(n.right, key)
152.      }
153.      return true
154.  }
155.
156.  // Remove removes the Item with key `key` from the tree
157.  func (bst *ItemBinarySearchTree) Remove(key int) {
158.      bst.lock.Lock()
159.      defer bst.lock.Unlock()
160.      remove(bst.root, key)
161.  }
162.
163.  // internal recursive function to remove an item
164.  func remove(node *Node, key int) *Node {
```

```
165.      if node == nil {
166.          return nil
167.      }
168.      if key < node.key {
169.          node.left = remove(node.left, key)
170.          return node
171.      }
172.      if key > node.key {
173.          node.right = remove(node.right, key)
174.          return node
175.      }
176.      // key == node.key
177.      if node.left == nil && node.right == nil {
178.          node = nil
179.          return nil
180.      }
181.      if node.left == nil {
182.          node = node.right
183.          return node
184.      }
185.      if node.right == nil {
186.          node = node.left
187.          return node
188.      }
189.      leftmostrightside := node.right
190.      for {
191.          //find smallest value on the right side
192.          if leftmostrightside != nil && leftmostrightside.left != nil {
193.              leftmostrightside = leftmostrightside.left
194.          } else {
195.              break
196.          }
197.      }
198.      node.key, node.value = leftmostrightside.key, leftmostrightside.value
199.      node.right = remove(node.right, node.key)
200.      return node
201.  }
202.
203.  // String prints a visual representation of the tree
204.  func (bst *ItemBinarySearchTree) String() {
205.      bst.lock.Lock()
206.      defer bst.lock.Unlock()
```

```
207.      fmt.Println("------------------------------------------------")
208.      stringify(bst.root, 0)
209.      fmt.Println("------------------------------------------------")
210.  }
211.
212.  // internal recursive function to print a tree
213.  func stringify(n *Node, level int) {
214.      if n != nil {
215.          format := ""
216.          for i := 0; i < level; i++ {
217.              format += "       "
218.          }
219.          format += "---[ "
220.          level++
221.          stringify(n.left, level)
222.          fmt.Printf(format+"%d\n", n.key)
223.          stringify(n.right, level)
224.      }
225.  }
```

二叉树的四种遍历算法

```
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100
#define MaxWidth 40

typedef char ElemType;

typedef struct tnode
{
    ElemType data;
    struct tnode *lchild,*rchild;
} BTNode;

void CreateBTree(BTNode * &bt,char *str)    /*由str创建二叉链bt*/
{
    BTNode *St[MaxSize],*p=NULL;
    int top=-1,k,j=0;
    char ch;
    bt=NULL;            /*建立的二叉树初始时为空*/
    ch=str[j];
    while (ch!='\0')      /*str未扫描完时循环*/
    {
              switch(ch)
        {
        case '(':top++;St[top]=p;k=1; break;    /*为左孩子结点*/
        case ')':top--;break;
        case ',':k=2; break;                    /*为孩子结点右结点*/
        default:p=(BTNode *)malloc(sizeof(BTNode));
                p->data=ch;p->lchild=p->rchild=NULL;
                if (bt==NULL)                   /**p为二叉树的根结点*/
                    bt=p;
                else                              /*已建立二叉树根结点*/
                {    switch(k)
                    {
                    case 1:St[top]->lchild=p;break;
                    case 2:St[top]->rchild=p;break;
                    }
                }
```

```
        }
        j++;
        ch=str[j];
    }
}

void DispBTree(BTNode *bt)    /*以括号表示法输出二叉树*/
{
    if (bt!=NULL)
    {
        printf("%c",bt->data);
        if (bt->lchild!=NULL || bt->rchild!=NULL)
        {
            printf("(");
            DispBTree(bt->lchild);        /*递归处理左子树*/
            if (bt->rchild!=NULL)
                printf(",");
            DispBTree(bt->rchild);        /*递归处理右子树*/
            printf(")");
        }
    }
}

// 先序遍历序列
void PreOrder(BTNode *bt)
{
    if (bt!=NULL)
    {
        printf("%c ",bt->data);
        PreOrder(bt->lchild);
        PreOrder(bt->rchild);
    }
}

// 中序遍历序列
void InOrder(BTNode *bt)
{
    if (bt!=NULL)
    {
        InOrder(bt->lchild);
        printf("%c ",bt->data);
        InOrder(bt->rchild);
```

```
    }
}

// 后序遍历序列
void PostOrder(BTNode *bt)
{
    if (bt!=NULL)
    {
        PostOrder(bt->lchild);
        PostOrder(bt->rchild);
        printf("%c ",bt->data);
    }
}

// 层次遍历序列
void LevelOrder(BTNode *b)
{
    BTNode *p;
    BTNode *qu[MaxSize];            /*定义环形队列,存放结点指针*/
    int front,rear;                   /*定义队头和队尾指针*/
    front=rear=-1;                  /*置队列为空队列*/
    rear++;
    qu[rear]=b;                       /*根结点指针进入队列*/
    while (front!=rear)               /*队列不为空*/
    {    front=(front+1)%MaxSize;
        p=qu[front];                /*队头出队列*/
        printf("%c ",p->data);        /*访问结点*/
        if (p->lchild!=NULL)        /*有左孩子时将其进队*/
        {    rear=(rear+1)%MaxSize;
            qu[rear]=p->lchild;
        }
        if (p->rchild!=NULL)        /*有右孩子时将其进队*/
        {    rear=(rear+1)%MaxSize;
            qu[rear]=p->rchild;
        }
    }
}

void main()
{
    BTNode *bt;
    CreateBTree(bt,"A(B(D,E(G,H)),C(,F(I)))");    /*构造图5.10(a)所示的二叉树*/
```

```
123.      printf("二叉树bt:");DispBTree(bt);printf("\n");
124.      printf("先序遍历序列:");PreOrder(bt);printf("\n");
125.      printf("中序遍历序列:");InOrder(bt);printf("\n");
126.      printf("后序遍历序列:");PostOrder(bt);printf("\n");
127.      printf("层次遍历序列:");LevelOrder(bt);printf("\n");
128.  }
```

哈夫曼树

```c
#include <stdio.h>
#define N 50        /*叶子结点数*/
#define M 2*N-1        /*树中结点总数*/

typedef struct
{
    char data;        /*结点值*/
    double weight;    /*权重*/
    int parent;        /*双亲结点*/
    int lchild;        /*左孩子结点*/
    int rchild;        /*右孩子结点*/
} HTNode;

typedef struct
{
    char cd[N];        /*存放哈夫曼码*/
    int start;
} HCode;

void CreateHT(HTNode ht[],int n)
{
    int i,k,lnode,rnode;
    double min1,min2;
    for (i=0;i<2*n-1;i++)            /*所有结点的相关域置初值-1*/
        ht[i].parent=ht[i].lchild=ht[i].rchild=-1;
    for (i=n;i<2*n-1;i++)            /*构造哈夫曼树*/
    {
        min1=min2=32767;            /*lnode和rnode为最小权重的两个结点位置*/
        lnode=rnode=-1;
        for (k=0;k<=i-1;k++)
            if (ht[k].parent==-1)    /*只在尚未构造二叉树的结点中查找*/
            {
                if (ht[k].weight<min1)
                {
                    min2=min1;rnode=lnode;
                    min1=ht[k].weight;lnode=k;
                }
                else if (ht[k].weight<min2)
```

```
                {
                    min2=ht[k].weight;rnode=k;
                }
            }
        ht[i].weight=ht[lnode].weight+ht[rnode].weight;
        ht[i].lchild=lnode;ht[i].rchild=rnode;
        ht[lnode].parent=i;
        ht[rnode].parent=i;
    }
}

void CreateHCode(HTNode ht[],HCode hcd[],int n)
{
    int i,f,c;
    HCode hc;
    for (i=0;i<n;i++)    /*根据哈夫曼树求哈夫曼编码*/
    {
        hc.start=n;c=i;
        f=ht[i].parent;
        while (f!=-1)    /*循序直到树根结点*/
        {
            if (ht[f].lchild==c)    /*处理左孩子结点*/
                hc.cd[hc.start--]='0';
            else                    /*处理右孩子结点*/
                hc.cd[hc.start--]='1';
            c=f;f=ht[f].parent;
        }
        hc.start++;        /*start指向哈夫曼编码最开始字符*/
        hcd[i]=hc;
    }
}

void DispHCode(HTNode ht[],HCode hcd[],int n)
{
    int i,k;
    double sum=0,m=0;
    int j;
    printf("输出哈夫曼编码:\n"); /*输出哈夫曼编码*/
    for (i=0;i<n;i++)
    {
        j=0;
        printf("   %c:",ht[i].data);
```

```
81.         for (k=hcd[i].start;k<=n;k++)
82.         {
83.             printf("%c",hcd[i].cd[k]);
84.             j++;
85.         }
86.         m+=ht[i].weight;
87.         sum+=ht[i].weight*j;
88.         printf("\n");
89.     }
90. }
91. 
92. void main()
93. {
94.     int n=5,i;        /*n表示初始字符串的个数*/
95.     char str[]={'a','b','c','d','e'};
96.     double fnum[]={4,2,1,7,3};
97.     HTNode ht[M];
98.     HCode hcd[N];
99.     for (i=0;i<n;i++)
100.     {
101.         ht[i].data=str[i];
102.         ht[i].weight=fnum[i];
103.     }
104.     printf("\n");
105.     CreateHT(ht,n);
106.     CreateHCode(ht,hcd,n);
107.     DispHCode(ht,hcd,n);
108.     printf("\n");
109. }
```

图的基本概念

图的定义

图是一种非线性结构，其中的元素是多对多的关系。

图是由非空的顶点的集合和描述顶点关系即边的集合组成。

图的特征

图的基本术语

- 有向图：边是有方向的图
- 无向图：边是无方向的图

图的存储结构

图的连接矩阵

```
#define MAXVEX  100

typedef char VertexType[3];         /*定义VertexType为char数组类型*/

typedef struct vertex
{
    int adjvex;                 /*顶点编号*/
    VertexType data;             /*顶点的信息*/
} VType;                        /*顶点类型*/

typedef struct graph
{
    int n,e;                    /*n为实际顶点数,e为实际边数*/
    VType vexs[MAXVEX];            /*顶点集合*/
    int edges[MAXVEX][MAXVEX];    /*边的集合*/
} AdjMatix;                        /*图的邻接矩阵类型*/
```

图的连接表

```c
#define MAXVEX 100

typedef char VertexType[3];

typedef struct edgenode
{
    int adjvex;                 /*邻接点序号*/
    int value;                 /*边的权值*/
    struct edgenode *next;        /*下一条边的顶点*/
} ArcNode;                        /*每个顶点建立的单链表中结点的类型*/

typedef struct vexnode
{
    VertexType data;              /*结点信息*/
    ArcNode *firstarc;            /*指向第一条边结点*/
} VHeadNode;                     /*单链表的头结点类型*/

typedef struct
{
    int n,e;                    /*n为实际顶点数,e为实际边数*/
    VHeadNode adjlist[MAXVEX];    /*单链表头结点数组*/
} AdjList;                        /*图的邻接表类型*/
```

图的遍历

广度优先搜索

深度优先搜索

有向图连接矩阵

图的定义

```
#include <stdio.h>
#define MAXVEX  100

typedef char VertexType[3];         /*定义VertexType为char数组类型*/

typedef struct vertex
{
    int adjvex;                 /*顶点编号*/
    VertexType data;             /*顶点的信息*/
} VType;                        /*顶点类型*/

typedef struct graph
{
    int n,e;                    /*n为实际顶点数,e为实际边数*/
    VType vexs[MAXVEX];            /*顶点集合*/
    int edges[MAXVEX][MAXVEX];    /*边的集合*/
} AdjMatix;                        /*图的邻接矩阵类型*/
```

创建图

```
int CreateMatix(AdjMatix &g)
{
    int i,j,k,b,t;
    int w;
    printf("顶点数(n)和边数(e):");
    scanf("%d%d",&g.n,&g.e);
    for (i=0;i<g.n;i++)
    {
        printf("   序号为%d的顶点信息:",i);
        scanf("%s",g.vexs[i].data);
        g.vexs[i].adjvex=i;         /*顶点编号为i*/
    }
    for (i=0;i<g.n;i++)
        for (j=0;j<g.n;j++)
```

```
            g.edges[i][j]=0;
    for (k=0;k<g.e;k++)
    {
        printf("   序号为%d的边=>",k);
        printf("  起点号 终点号 权值:");
        scanf("%d%d%d",&b,&t,&w);
        if (b<g.n && t<g.n && w>0)
            g.edges[b][t]=w;
        else
        {
            printf("输入错误!\n");
            return(0);
        }
    }
    return(1);
}
```

列出图

```
void DispMatix(AdjMatix g)
{
    int i,j;
    printf("\n图的邻接矩阵:\n");
    for (i=0;i<g.n;i++)
    {
        for (j=0;j<g.n;j++)
            printf("%3d",g.edges[i][j]);
        printf("\n");
    }
}
```

main

```
void main()
{
    AdjMatix g;
    CreateMatix(g);
    DispMatix(g);
}
```

有向图连接链表

图的定义

```
#include <stdio.h>
#include <malloc.h>
#define MAXVEX 100

typedef char VertexType[3];

typedef struct edgenode
{
    int adjvex;                 /*邻接点序号*/
    int value;                  /*边的权值*/
    struct edgenode *next;      /*下一条边的顶点*/
} ArcNode;                      /*每个顶点建立的单链表中结点的类型*/

typedef struct vexnode
{
    VertexType data;            /*结点信息*/
    ArcNode *firstarc;          /*指向第一条边结点*/
} VHeadNode;                    /*单链表的头结点类型*/

typedef struct
{
    int n,e;                    /*n为实际顶点数,e为实际边数*/
    VHeadNode adjlist[MAXVEX];  /*单链表头结点数组*/
} AdjList;                      /*图的邻接表类型*/
```

创建图

```
int CreateAdjList(AdjList *&G)    /*建立有向图的邻接表*/
{
    int i,b,t,w;
    ArcNode *p;
    G=(AdjList *)malloc(sizeof(AdjList));
    printf("顶点数(n),边数(e):");
    scanf("%d%d",&G->n,&G->e);
```

```
    for (i=0;i<G->n;i++)
    {
        printf("   序号为%d的顶点信息:", i);
        scanf("%s",G->adjlist[i].data);
        G->adjlist[i].firstarc=NULL;
    }
    for (i=0;i<G->e;i++)
    {
        printf("   序号为边=>",i);
        printf(" 起点号 终点号 权值:");
        scanf("%d%d%d",&b,&t,&w);
        if (b<G->n && t<G->n && w>0)
        {
            p=(ArcNode *)malloc(sizeof(ArcNode));    /*建立*p结点*/
            p->value=w;p->adjvex=t;
            p->next=G->adjlist[b].firstarc;     /**p插入到adjlist[b]的单链表之首*/
            G->adjlist[b].firstarc=p;
        }
        else
        {
            printf("输入错误!\n");
            return(0);
        }
    }
    return(1);
}
```

列出图

```
void DispAdjList(AdjList *G)
{
    int i;
    ArcNode *p;
    printf("图的邻接表表示如下:\n");
    for (i=0;i<G->n;i++)
    {
        printf("  [%d,%3s]=>",i,G->adjlist[i].data);
        p=G->adjlist[i].firstarc;
        while (p!=NULL)
        {
            printf("(%d,%d)->",p->adjvex,p->value);
```

```
            p=p->next;
        }
        printf("∧\n");
    }
}
```

main

```
void main()
{
    AdjList *G;
    CreateAdjList(G);
    DispAdjList(G);
}
```

By Golang

```go
// Package graph creates a ItemGraph data structure for the Item type
package graph

import (
    "fmt"
    "sync"
)

// Item the type of the binary search tree
type Item interface{}

// Node a single node that composes the tree
type Node struct {
    value Item
}

func (n *Node) String() string {
    return fmt.Sprintf("%v", n.value)
}

// ItemGraph the Items graph
type ItemGraph struct {
    nodes []*Node
    edges map[Node][]*Node
    lock  sync.RWMutex
}

// AddNode adds a node to the graph
func (g *ItemGraph) AddNode(n *Node) {
    g.lock.Lock()
    g.nodes = append(g.nodes, n)
    g.lock.Unlock()
}

// AddEdge adds an edge to the graph
func (g *ItemGraph) AddEdge(n1, n2 *Node) {
    g.lock.Lock()
    if g.edges == nil {
```

```
39.         g.edges = make(map[Node][]*Node)
40.     }
41.     g.edges[*n1] = append(g.edges[*n1], n2)
42.     g.edges[*n2] = append(g.edges[*n2], n1)
43.     g.lock.Unlock()
44. }
45.
46. // AddEdge adds an edge to the graph
47. func (g *ItemGraph) String() {
48.     g.lock.RLock()
49.     s := ""
50.     for i := 0; i < len(g.nodes); i++ {
51.         s += g.nodes[i].String() + " -> "
52.         near := g.edges[*g.nodes[i]]
53.         for j := 0; j < len(near); j++ {
54.             s += near[j].String() + " "
55.         }
56.         s += "\n"
57.     }
58.     fmt.Println(s)
59.     g.lock.RUnlock()
60. }
61.
62. // Traverse implements the BFS traversing algorithm
63. func (g *ItemGraph) Traverse(f func(*Node)) {
64.     g.lock.RLock()
65.     q := NodeQueue{}
66.     q.New()
67.     n := g.nodes[0]
68.     q.Enqueue(*n)
69.     visited := make(map[*Node]bool)
70.     for {
71.         if q.IsEmpty() {
72.             break
73.         }
74.         node := q.Dequeue()
75.         visited[node] = true
76.         near := g.edges[*node]
77.
78.         for i := 0; i < len(near); i++ {
79.             j := near[i]
80.             if !visited[j] {
```

```
81.                 q.Enqueue(*j)
82.                 visited[j] = true
83.             }
84.         }
85.         if f != nil {
86.             f(node)
87.         }
88.     }
89.     g.lock.RUnlock()
90. }
```

- 广度优先遍历
- 深度优先遍历

广度优先遍历

图的遍历

给定一个图G=(V,E)和V(G)中的任一顶点v，从v出发，顺着G的边访问G中的所有顶点，且每个顶点仅被访问一次，这一过程称为遍历图。

一般设置一个辅助数组visited[]，用来标记顶点是否被访问过，初始状态为0，访问过则设置为1。

广度优先遍历

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

#define MAXVEX 100

typedef char VertexType[3];         /*定义VertexType为char数组类型*/

typedef struct vertex
{
    int adjvex;                      /*顶点编号*/
    VertexType data;                /*顶点的信息*/
} VType;                           /*顶点类型*/

typedef struct graph
{
    int n,e;                       /*n为实际顶点数,e为实际边数*/
    VType vexs[MAXVEX];             /*顶点集合*/
    int edges[MAXVEX][MAXVEX];     /*边的集合*/
} AdjMatix;                         /*图的邻接矩阵类型*/

typedef struct edgenode
{
    int adjvex;                    /*邻接点序号*/
    int value;                     /*边的权值*/
    struct edgenode *next;          /*下一条边的顶点*/
} ArcNode;                          /*每个顶点建立的单链表中结点的类型*/

```

```
typedef struct vexnode
{
    VertexType data;                /*结点信息*/
    ArcNode *firstarc;              /*指向第一条边结点*/
} VHeadNode;                       /*单链表的头结点类型*/

typedef struct
{
    int n,e;                       /*n为实际顶点数,e为实际边数*/
    VHeadNode adjlist[MAXVEX];     /*单链表头结点数组*/
} AdjList;                         /*图的邻接表类型*/

void DispAdjList(AdjList *G)
{
    int i;
    ArcNode *p;
    printf("图的邻接表表示如下:\n");
    for (i=0;i<G->n;i++)
    {
        printf("  [%d,%3s]=>",i,G->adjlist[i].data);
        p=G->adjlist[i].firstarc;
        while (p!=NULL)
        {
            printf("(%d,%d)->",p->adjvex,p->value);
            p=p->next;
        }
        printf("∧\n");
    }
}

void MatToList(AdjMatix g,AdjList *&G)  /*例6.3算法:将邻接矩阵g转换成邻接表G*/
{
    int i,j;
    ArcNode *p;
    G=(AdjList *)malloc(sizeof(AdjList));
    for (i=0;i<g.n;i++)                    /*给邻接表中所有头结点的指针域置初值*/
    {
        G->adjlist[i].firstarc=NULL;
        strcpy(G->adjlist[i].data,g.vexs[i].data);
    }
    for (i=0;i<g.n;i++)                    /*检查邻接矩阵中每个元素*/
        for (j=g.n-1;j>=0;j--)
```

```
            if (g.edges[i][j]!=0)        /*邻接矩阵的当前元素不为0*/
            {
                p=(ArcNode *)malloc(sizeof(ArcNode));/*创建一个结点*p*/
                p->value=g.edges[i][j];p->adjvex=j;
                p->next=G->adjlist[i].firstarc;          /*将*p链到链表后*/
                G->adjlist[i].firstarc=p;
            }
    G->n=g.n;G->e=g.e;
}

void BFS(AdjList *G,int vi)        /*对邻接表g从顶点vi开始进行广宽优先遍历*/
{
    int i,v,visited[MAXVEX];
    int Qu[MAXVEX],front=0,rear=0;        /*循环队列*/
    ArcNode *p;
    for (i=0;i<G->n;i++)                /*给visited数组置初值0*/
        visited[i]=0;
    printf("%d ",vi);                    /*访问初始顶点*/
    visited[vi]=1;                       /*置已访问标识*/
    rear=(rear=1)%MAXVEX;                /*循环移动队尾指针*/
    Qu[rear]=vi;                        /*初始顶点进队*/
    while (front!=rear)                    /*队列不为空时循环*/
    {
        front=(front+1) % MAXVEX;
        v=Qu[front];                    /*顶点v出队*/
        p=G->adjlist[v].firstarc;        /*找v的第一个邻接点*/
        while (p!=NULL)                    /*找v的所有邻接点*/
        {
            if (visited[p->adjvex]==0)    /*未访问过则访问之*/
            {
                visited[p->adjvex]=1;    /*置已访问标识*/
                printf("%d ",p->adjvex);/*访问该点并使之入队列*/
                rear=(rear+1) % MAXVEX;    /*循环移动队尾指针*/
                Qu[rear]=p->adjvex;        /*顶点p->adjvex进队*/
            }
            p=p->next;                    /*找v的下一个邻接点*/
        }
    }
}

void main()
{
```

```
113.      int i,j;
114.      AdjMatix g;
115.      AdjList *G;
          int a[5][5]={ {0,1,0,1,0},{1,0,1,0,0},{0,1,0,1,1},{1,0,1,0,1},{0,0,1,1,0}
116.  };
117.      char *vname[MAXVEX]={"a","b","c","d","e"};
118.      g.n=5;g.e=12;    /*建立图6.1(a)的无向图,每1条无向边算为2条有向边*/
119.      for (i=0;i<g.n;i++)
120.          strcpy(g.vexs[i].data,vname[i]);
121.      for (i=0;i<g.n;i++)
122.          for (j=0;j<g.n;j++)
123.              g.edges[i][j]=a[i][j];
124.      MatToList(g,G);         /*生成邻接表*/
125.      DispAdjList(G);         /*输出邻接表*/
126.      printf("从顶点0的广度优先遍历序列:\n");
127.      printf("\t");BFS(G,0);printf("\n");
128.  }
```

深度优先遍历

基本思想

- 从图G中某个顶点vi出发，访问vi，然后选择一个与vi相邻且没有被访问过的顶点v访问，再从v出发选择一个与v相邻且未被访问的顶点vj访问，依次访问。
- 如果当前已被访问的顶点的所有邻接顶点都已被访问，则回退到已被访问的顶点序列中最后一个拥有未被访问的相邻顶点w，从w出发按相同的方法继续遍历，直到所有的顶点都被访问到。

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

#define MAXVEX 100

typedef char VertexType[3];         /*定义VertexType为char数组类型*/

typedef struct vertex
{
    int adjvex;                      /*顶点编号*/
    VertexType data;               /*顶点的信息*/
} VType;                           /*顶点类型*/

typedef struct graph
{
    int n,e;                      /*n为实际顶点数,e为实际边数*/
    VType vexs[MAXVEX];             /*顶点集合*/
    int edges[MAXVEX][MAXVEX];    /*边的集合*/
} AdjMatix;                         /*图的邻接矩阵类型*/

typedef struct edgenode
{
    int adjvex;                   /*邻接点序号*/
    int value;                    /*边的权值*/
    struct edgenode *next;          /*下一条边的顶点*/
} ArcNode;                          /*每个顶点建立的单链表中结点的类型*/

typedef struct vexnode
{
```

```
31.     VertexType data;                /*结点信息*/
32.     ArcNode *firstarc;              /*指向第一条边结点*/
33. } VHeadNode;                      /*单链表的头结点类型*/
34. 
35. typedef struct
36. {
37.     int n,e;                      /*n为实际顶点数,e为实际边数*/
38.     VHeadNode adjlist[MAXVEX];    /*单链表头结点数组*/
39. } AdjList;                         /*图的邻接表类型*/
40. 
41. void DispAdjList(AdjList *G)
42. {
43.     int i;
44.     ArcNode *p;
45.     printf("图的邻接表表示如下:\n");
46.     for (i=0;i<G->n;i++)
47.     {
48.         printf("  [%d,%3s]=>",i,G->adjlist[i].data);
49.         p=G->adjlist[i].firstarc;
50.         while (p!=NULL)
51.         {
52.             printf("(%d,%d)->",p->adjvex,p->value);
53.             p=p->next;
54.         }
55.         printf("∧\n");
56.     }
57. }
58. 
59. void MatToList(AdjMatix g,AdjList *&G)  /*例6.3算法:将邻接矩阵g转换成邻接表G*/
60. {
61.     int i,j;
62.     ArcNode *p;
63.     G=(AdjList *)malloc(sizeof(AdjList));
64.     for (i=0;i<g.n;i++)                     /*给邻接表中所有头结点的指针域置初值*/
65.     {
66.         G->adjlist[i].firstarc=NULL;
67.         strcpy(G->adjlist[i].data,g.vexs[i].data);
68.     }
69.     for (i=0;i<g.n;i++)                     /*检查邻接矩阵中每个元素*/
70.         for (j=g.n-1;j>=0;j--)
71.             if (g.edges[i][j]!=0)         /*邻接矩阵的当前元素不为0*/
72.             {
```

```
                p=(ArcNode *)malloc(sizeof(ArcNode));/*创建一个结点*p*/
                p->value=g.edges[i][j];p->adjvex=j;
                p->next=G->adjlist[i].firstarc;          /*将*p链到链表后*/
                G->adjlist[i].firstarc=p;
            }
    G->n=g.n;G->e=g.e;
}

int visited[MAXVEX];
void DFS(AdjList *g,int vi)        /*对邻接表G从顶点vi开始进行深度优先遍历*/
{
    ArcNode *p;
    printf("%d ",vi);            /*访问vi顶点*/
    visited[vi]=1;                /*置已访问标识*/
    p=g->adjlist[vi].firstarc;    /*找vi的第一个邻接点*/
    while (p!=NULL)               /*找vi的所有邻接点*/
    {
        if (visited[p->adjvex]==0)
            DFS(g,p->adjvex);    /*从vi未访问过的邻接点出发深度优先搜索*/
        p=p->next;                /*找vi的下一个邻接点*/
    }
}

void DFS1(AdjList *G,int vi)        /*非递归深度优先遍历算法*/
{
    ArcNode *p;
    ArcNode *St[MAXVEX];
    int top=-1,v;
    printf("%d ",vi);                    /*访问vi顶点*/
    visited[vi]=1;                       /*置已访问标识*/
    top++;                               /*将初始顶点vi的firstarc指针进栈*/
    St[top]=G->adjlist[vi].firstarc;
    while (top>-1)                       /*栈不空循环*/
    {
        p=St[top];top--;                /*出栈一个顶点为当前顶点*/
        while (p!=NULL)                   /*循环搜索其相邻顶点*/
        {
            v=p->adjvex;                /*取相邻顶点的编号*/
            if (visited[v]==0)            /*若该顶点未访问过*/
            {
                printf("%d ",v);        /*访问v顶点*/
                visited[v]=1;            /*置访问标识*/
```

```
                top++;                      /*将该顶点的第1个相邻顶点进栈*/
                St[top]=G->adjlist[v].firstarc;
                break;                      /*退出当前顶点的搜索*/
            }
            p=p->next;                      /*找下一个相邻顶点*/
        }
    }
}

void main()
{
    int i,j;
    AdjMatix g;
    AdjList *G;
    int a[5][5]={ {0,1,0,1,0},{1,0,1,0,0},{0,1,0,1,1},{1,0,1,0,1},{0,0,1,1,0}
};
    char *vname[MAXVEX]={"a","b","c","d","e"};
    g.n=5;g.e=12;    /*建立图6.1(a)的无向图,每1条无向边算为2条有向边*/
    for (i=0;i<g.n;i++)
        strcpy(g.vexs[i].data,vname[i]);
    for (i=0;i<g.n;i++)
        for (j=0;j<g.n;j++)
            g.edges[i][j]=a[i][j];
    MatToList(g,G);         /*生成邻接表*/
    DispAdjList(G);         /*输出邻接表*/
    for (i=0;i<g.n;i++)    visited[i]=0; /*顶点标识置初值*/
    printf("从顶点0的深度优先遍历序列:\n");
    printf("  递归算法:");DFS(G,0);printf("\n");
    for (i=0;i<g.n;i++)    visited[i]=0; /*顶点标识置初值*/
    printf("  非递归算法:");DFS1(G,0);printf("\n");
}
```

- 普里姆算法
- 克鲁斯卡尔算法

普里姆算法

```
#include <stdio.h>
#define MAXVEX 100
#define INF 32767       /*INF表示∞*/

void Prim(int cost[][MAXVEX],int n,int v)
/*输出最小生成树的每条边*/
{
    int lowcost[MAXVEX],min;
    int closest[MAXVEX],i,j,k;
    for (i=0;i<n;i++)               /*给lowcost[]和closest[]置初值*/
    {
        lowcost[i]=cost[v][i];
            closest[i]=v;
    }
    for (i=1;i<n;i++)               /*找出n-1个顶点*/
    {
        min=INF;
        for (j=0;j<n;j++)       /*在(V-U)中找出离U最近的顶点k*/
            if (lowcost[j]!=0 && lowcost[j]<min)
            {
                min=lowcost[j];
                k=j;
            }
        printf("  边(%d,%d)权为:%d\n",closest[k],k,min);
        lowcost[k]=0;               /*标记k已经加入U*/
        for (j=0;j<n;j++)       /*修改数组lowcost和closest*/
            if (cost[k][j]!=0 && cost[k][j]<lowcost[j])
            {
                lowcost[j]=cost[k][j];
                closest[j]=k;
            }
    }
}

void main()
{
    int n=7;
    int cost[7][MAXVEX]={
```

```
39.         {0,50,60,INF,INF,INF,INF},
40.         {50,0,INF,65,40,INF,INF},
41.         {60,INF,0,52,INF,INF,45},
42.         {INF,65,52,0,50,30,42},
43.         {INF,40,INF,50,0,70,INF},
44.         {INF,INF,INF,30,70,0,INF},
45.         {INF,INF,45,42,INF,INF,0}};
46.     printf("最小生成树:\n");Prim(cost,n,0);
47. }
```

克鲁斯卡尔算法

```
#include <stdio.h>
#define MAXVEX 100

typedef struct
{
    int u;      /*边的起始顶点*/
    int v;      /*边的终止顶点*/
    int w;      /*边的权值*/
} Edge;

void Kruskal(Edge E[],int n,int e)
{
    int i,j,m1,m2,sn1,sn2,k;
    int vset[MAXVEX];
    for (i=0;i<n;i++) vset[i]=i;    /*初始化辅助数组*/
    k=1;                             /*k表示构造最小生成树的第几条边,初值为1*/
    j=0;                             /*E中边的下标,初值为0*/
    while (k<n)                      /*生成的边数小于n时循环*/
    {
        m1=E[j].u;m2=E[j].v;        /*取一条边的头尾顶点*/
        sn1=vset[m1];sn2=vset[m2];     /*分别得到两个顶点所属的集合编号*/
        if (sn1!=sn2)                   /*两顶点属不同的集合,该边是最小生成树的边*/
        {
            printf("  边(%d,%d)权为:%d\n",m1,m2,E[j].w);
            k++;                       /*生成边数增1*/
            for (i=0;i<n;i++)          /*两个集合统一编号*/
                if (vset[i]==sn2)      /*集合编号为sn2的改为sn1*/
                    vset[i]=sn1;
        }
        j++;                           /*扫描下一条边*/
    }
}

void main()
{
    int n=7,e=10;
    Edge E[]={
        {3,5,30},{1,4,40},{3,6,42},
```

```
39.         {2,6,45},{0,1,50},{3,4,50},
40.         {2,3,52},{0,2,60},{1,3,65},
41.         {4,5,70}};
42.     printf("最小生成树:\n");Kruskal(E,n,e);
43. }
```

- 狄克斯特拉算法
- 弗洛伊德算法

狄克斯特拉算法

```
#include <stdio.h>
#define MAXVEX 100
#define INF 32767

void Dijkstra(int cost[][MAXVEX],int n,int v)
{
    int dist[MAXVEX],path[MAXVEX];
    int s[MAXVEX];
    int mindis,i,j,u,pre;
    for (i=0;i<n;i++)
    {
        dist[i]=cost[v][i];           /*距离初始化*/
        s[i]=0;                        /*s[]置空*/
        if (cost[v][i]<INF)            /*路径初始化*/
            path[i]=v;
        else
            path[i]=-1;
    }
    s[v]=1;path[v]=0;                /*源点编号v放入s中*/
    for (i=0;i<n;i++)                 /*循环直到所有顶点的最短路径都求出*/
    {
        mindis=INF;
        u=-1;
        for (j=0;j<n;j++)             /*选取不在s中且具有最小距离的顶点u*/
            if (s[j]==0 && dist[j]<mindis)
            {
                u=j;
                mindis=dist[j];
            }
        if (u!=-1)                        /*找到最小距离的顶点u*/
        {    s[u]=1;                     /*顶点u加入s中*/
            for (j=0;j<n;j++)              /*修改不在s中的顶点的距离*/
                if (s[j]==0)
                    if (cost[u][j]<INF && dist[u]+cost[u][j]<dist[j])
                    {
                        dist[j]=dist[u]+cost[u][j];
                        path[j]=u;
                    }
```

```
        }
    }
    printf("\n Dijkstra算法求解如下:\n");
    for (i=0;i<n;i++)                /*输出最短路径长度,路径逆序输出*/
    {
        if (i!=v)
        {
            printf("  %d->%d:",v,i);
            if (s[i]==1)
            {
                printf("路径长度为%2d ",dist[i]);
                pre=i;
                printf("路径逆序为");
                while (pre!=v)        /*一直求解到初始顶点*/
                {
                    printf("%d,",pre);
                    pre=path[pre];
                }
                printf("%d\n",pre);
            }
            else
                printf("不存在路径\n");
        }
    }
}

void main()
{
    int cost[6][MAXVEX]={        /*图6.9的代价矩阵*/
        {0,50,10,INF,INF,INF},
        {INF,0,15,50,10,INF},
        {20,INF,0,15,INF,INF},
        {INF,20,INF,0,35,INF},
        {INF,INF,INF,30,0,INF},
        {INF,INF,INF,3,INF,0}};
    Dijkstra(cost,6,1);
    printf("\n");
}
```

弗洛伊德算法

```
#include <stdio.h>
#define MAXVEX 100
#define INF 32767

void Floyed(int cost[][MAXVEX],int n)
{
    int A[MAXVEX][MAXVEX],path[MAXVEX][MAXVEX];
    int i,j,k,pre;
    for (i=0;i<n;i++)        /*置初值*/
        for (j=0;j<n;j++)
        {
            A[i][j]=cost[i][j];
            path[i][j]=-1;
        }
    for (k=0;k<n;k++)
    {
        for (i=0;i<n;i++)
            for (j=0;j<n;j++)
                if (A[i][j]>(A[i][k]+A[k][j]))
                {
                    A[i][j]=A[i][k]+A[k][j];
                    path[i][j]=k;
                }
    }
    printf("\n Floyed算法求解如下:\n");
    for (i=0;i<n;i++)        /*输出最短路径*/
        for (j=0;j<n;j++)
            if (i!=j)
            {
                printf("   %d->%d:",i,j);
                if (A[i][j]==INF)
                {
                    if (i!=j)
                        printf("不存在路径\n");
                }
                else
                {
                    printf("路径长度为:%3d ",A[i][j]);
```

```
                    printf("路径为%d ",i);
                    pre=path[i][j];
                    while (pre!=-1)
                    {
                        printf("%d ",pre);
                        pre=path[pre][j];
                    }
                    printf("%d\n",j);
                }
            }
}

void main()
{
    int cost[6][MAXVEX]={        /*图6.9的代价矩阵*/
        {0,50,10,INF,INF,INF},
        {INF,0,15,50,10,INF},
        {20,INF,0,15,INF,INF},
        {INF,20,INF,0,35,INF},
        {INF,INF,INF,30,0,INF},
        {INF,INF,INF,3,INF,0}};
    Floyed(cost,6);
    printf("\n");
}
```

拓扑排序算法

```c
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#define MAXVEX 100

typedef char VertexType[3];         /*定义VertexType为char数组类型*/

typedef struct vertex
{
    int adjvex;
    VertexType data;
} VType;

typedef struct graph
{
    int n,e;                    /*n为实际顶点数,e为实际边数*/
    VType vexs[MAXVEX];           /*顶点集合*/
    int edges[MAXVEX][MAXVEX];    /*边的集合*/
} AdjMatix;                       /*图的邻接矩阵类型*/

typedef struct edgenode
{
    int adjvex;                  /*邻接点序号*/
    int value;                  /*边的权值*/
    struct edgenode *next;        /*下一条边的顶点*/
} ArcNode;                        /*每个顶点建立的单链表中结点的类型*/

typedef struct vexnode
{
    VertexType data;              /*结点信息*/
    int count;                    /*存放顶点入度,新增用于拓扑排序*/
    ArcNode *firstarc;            /*指向第一条边结点*/
} VHeadNode;                    /*单链表的头结点类型*/

typedef struct
{
    int n,e;                    /*n为实际顶点数,e为实际边数*/
    VHeadNode adjlist[MAXVEX];    /*单链表头结点数组*/
```

```
39. } AdjList;                          /*图的邻接表类型*/
40.
41. void DispAdjList(AdjList *G)    /*显示邻接表(含顶点入度)*/
42. {
43.     int i;
44.     ArcNode *p;
45.     printf("图的邻接表表示如下:\n");
46.     for (i=0;i<G->n;i++)
47.     {
48.         printf("  [%d,%3s:]=>",i,G->adjlist[i].data,G->adjlist[i].count);
49.         p=G->adjlist[i].firstarc;
50.         while (p!=NULL)
51.         {
52.             printf("(%d,%d)->",p->adjvex,p->value);
53.             p=p->next;
54.         }
55.         printf("∧\n");
56.     }
57. }
58.
59. void MatToList(AdjMatix g,AdjList *&G)  /*例6.3算法:将邻接矩阵g转换成邻接表G*/
60. {
61.     int i,j;
62.     ArcNode *p;
63.     G=(AdjList *)malloc(sizeof(AdjList));
64.     for (i=0;i<g.n;i++)                    /*给邻接表中所有头结点的指针域置初值*/
65.     {
66.         G->adjlist[i].firstarc=NULL;
67.         strcpy(G->adjlist[i].data,g.vexs[i].data);
68.     }
69.     for (i=0;i<g.n;i++)                    /*检查邻接矩阵中每个元素*/
70.         for (j=g.n-1;j>=0;j--)
71.             if (g.edges[i][j]!=0)         /*邻接矩阵的当前元素不为0*/
72.             {
73.                 p=(ArcNode *)malloc(sizeof(ArcNode));/*创建一个结点*p*/
74.                 p->value=g.edges[i][j];p->adjvex=j;
75.                 p->next=G->adjlist[i].firstarc;          /*将*p链到链表后*/
76.                 G->adjlist[i].firstarc=p;
77.             }
78.     G->n=g.n;G->e=g.e;
79. }
80.
```

```
81. void TopSort(AdjList *G)
82. {
83.     int i,j;
84.     int St[MAXV],top=-1;                /*栈St的指针为top*/
85.     ArcNode *p;
86.     for (i=0;i<n;i++)
87.         if (adj[i].count==0)            /*入度为0的顶点入栈*/
88.         {
89.             top++;
90.             St[top]=i;
91.         }
92.         while (top>-1)                  /*栈不为空时循环*/
93.         {
94.             i=St[top];top--;            /*出栈*/
95.             printf("%d ",i);            /*输出顶点*/
96.             p=adj[i].firstarc;          /*找第一个相邻顶点*/
97.             while (p!=NULL)
98.             {
99.                 j=p->adjvex;
100.                adj[j].count--;
101.                if (adj[j].count==0)/*入度为0的相邻顶点入栈*/
102.                {
103.                    top++;
104.                    St[top]=j;
105.                }
106.                p=p->nextarc;         /*找下一个相邻顶点*/
107.            }
108.        }
109. }
110.
111. void main()
112. {
113.     int i,j;
114.     AdjMatix g;
115.     AdjList *G;
116.     int a[6][6]={ {0,1,0,10},{1,0,1,0,0},{0,1,0,1,1},{1,0,1,0,1},{0,0,1,1,0} };
117.     char *vname[MAXVEX]={"a","b","c","d","e"};
118.     g.n=5;g.e=12;    /*建立图6.1(a)的无向图,每1条无向边算为2条有向边*/
119.     for (i=0;i<g.n;i++)
120.         strcpy(g.vexs[i].data,vname[i]);
121.     for (i=0;i<g.n;i++)
122.         for (j=0;j<g.n;j++)
```

```
123.             g.edges[i][j]=a[i][j];
124.     MatToList(g,G);        /*生成邻接表*/
125.     DispAdjList(G);        /*输出邻接表*/
126.     for (i=0;i<g.n;i++)    visited[i]=0; /*顶点标识置初值*/
127.     printf("从顶点0的深度优先遍历序列:\n");
128.     printf("  递归算法:");DFS(G,0);printf("\n");
129.     for (i=0;i<g.n;i++)    visited[i]=0; /*顶点标识置初值*/
130.     printf("  非递归算法:");DFS1(G,0);printf("\n");
131. }
```

顺序查找

```c
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;

typedef char ElemType[10];

typedef struct
{
    KeyType key;        /*存放关键字,KeyType为关键字类型*/
    ElemType data;    /*其他数据, ElemType为其他数据的类型*/
} LineList;

int SeqSearch(LineList R[],int n,KeyType k)
{
    int i=0;
    while (i<n && R[i].key!=k) i++;
    if (i>=n)
        return(-1);
    else
        return(i);
}

void main()
{
    KeyType a[]={3,9,1,5,8,10,6,7,2,4},k=6;
    LineList R[MaxSize];
    int n=10,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    i=SeqSearch(R,n,k);
    if (i>=0)
        printf("R[%d].key=%d\n",i,k);
    else
        printf("%d不在a中\n",k);
}
```

二分法查找

```c
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;

typedef char ElemType[10];

typedef struct
{
    KeyType key;        /*存放关键字,KeyType为关键字类型*/
    ElemType data;    /*其他数据, ElemType为其他数据的类型*/
} LineList;

int BinSearch(LineList R[],int n,KeyType k)
{
    int i,low=0,high=n-1,mid;
    int find=0;                 /*find=0表示未找到;find=1表示已找到*/
       while (low<=high && !find)
     {    mid=(low+high)/2;   /*整除取中间值*/
        if (k<R[mid].key)
            high=mid-1;
        else if (k>R[mid].key)
            low=mid+1;
           else
        {    i=mid;
            find=1;
        }
}
     if (find==0)
        return(-1);
else
        return(i);
}

void main()
{
    KeyType a[]={2,4,7,9,10,14,18,26,32,40},k=7;
    LineList R[MaxSize];
```

```
39.     int n=10,i;
40.     for (i=0;i<n;i++)
41.         R[i].key=a[i];
42.     i=BinSearch(R,n,k);
43.     if (i>=0)
44.         printf("R[%d].key=%d\n",i,k);
45.     else
46.         printf("%d不在a中\n",k);
47. }
```

分块查找

```c
#include <stdio.h>
#define MaxSize 100
#define MaxBlk 20

typedef int KeyType;

typedef char ElemType[10];

typedef struct
{
    KeyType key;        /*存放关键字,KeyType为关键字类型*/
    ElemType data;    /*其他数据, ElemType为其他数据的类型*/
} LineList;

typedef struct
{
    KeyType key;
    int low,high;
} IDXType;            /*索引表的类型*/

int BlkSearch(LineList R[],IDXType idx[],int m,KeyType k)
{
    int low=0,high=m-1,mid,i,j,find=0;
    while (low<=high && !find)    /*二分查找索引表*/
    {
        mid=(low+high)/2;
        if (k<idx[mid].key)
            high=mid-1;
        else if (k>idx[mid].key)
            low=mid+1;
        else
        {
            high=mid-1;
            find=1;
        }
    }
    if (low<m)                    /*k小于索引表内最大值*/
    {
```

```
        i=idx[low].low;             /*在索引表中定块起始地址*/
        j=idx[low].high;          /*在索引表中定块终止地址*/
       }
    while (i<j && R[i].key!=k) /*在指定的块内采用顺序方法进行查找*/
        i++;
     if (i>=j)
         return(-1);
    else
        return(i);
}

void main()
{
    KeyType a[]={9,22,12,14,35,42,44,38,48,60,58,47,78,80,77,82},k=48;
    LineList R[MaxSize];
    IDXType I[MaxBlk];
    int n=16,m=4,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    I[0].key=22;I[0].low=0;I[0].high=3;
    I[1].key=44;I[1].low=4;I[1].high=7;
    I[2].key=60;I[2].low=8;I[2].high=11;
    I[3].key=82;I[3].low=12;I[3].high=15;
    i=BlkSearch(R,I,m,k);
    if (i>=0)
        printf("R[%d].key=%d\n",i,k);
    else
        printf("%d不在a中\n",k);
}
```

二叉树排序查找

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

typedef int KeyType;

typedef char ElemType[10];

typedef struct tnode
{
    KeyType key;
    ElemType data;
    struct tnode *lchild,*rchild;
} BSTNode;

BSTNode *BSTSearch(BSTNode *bt,KeyType k)
{
    BSTNode *p=bt;
    while (p!=NULL && p->key!=k)
    {
        if (k<p->key)
            p=p->lchild;  /*沿左子树查找*/
        else
            p=p->rchild;  /*沿右子树查找*/
    }
    return(p);
}

int BSTInsert(BSTNode *&bt,KeyType k)
{
    BSTNode *f,*p=bt;
    while (p!=NULL)
    {
        if (p->key==k)
            return(0);
        f=p;                          /*f指向*p结点的双亲结点*/
        if (p->key>k)
            p=p->lchild;              /*在左子树中查找*/
        else
```

```
            p=p->rchild;            /*在右子树中查找*/
    }
    p=(BSTNode *)malloc(sizeof(BSTNode));    /*建立新结点*/
    p->key=k;
    p->lchild=p->rchild=NULL;
    if (bt==NULL)                    /*原树为空时,*p作为根结点插入*/
        bt=p;
    else if (k<f->key)
        f->lchild=p;                /*插入*p作为*f的左孩子*/
    else
        f->rchild=p;                /*插入*p作为*f的右孩子*/
    return(1);
}

void CreateBST(BSTNode *&bt,KeyType str[],int n)
{
    bt=NULL;                        /*初始时bt为空树*/
    int i=0;
    while (i<n)
    {
        BSTInsert(bt,str[i]); /*将关键字str[i]插入二叉排序树bt中*/
        i++;
    }
}

void DispBST(BSTNode *bt)
{
    if (bt!=NULL)
    {    printf("%d",bt->key);
        if (bt->lchild!=NULL || bt->rchild!=NULL)
        {    printf("(");
            DispBST(bt->lchild);                /*递归处理左子树*/
            if (bt->rchild!=NULL) printf(",");
            DispBST(bt->rchild);                /*递归处理右子树*/
            printf(")");
        }
    }
}

int BSTDelete(BSTNode *&bt,KeyType k)
{
    BSTNode *p=bt,*f,*r,*f1;
```

```
    f=NULL;                          /*p指向待比较的结点,f指向*p的双亲结点*/
    while (p!=NULL && p->key!=k)/*查找值域为x的结点*/
    {    f=p;
         if (p->key>k)
            p=p->lchild;         /*在左子树中查找*/
         else
            p=p->rchild;         /*在右子树中查找*/
    }
    if (p==NULL)                 /*未找到key域为k的结点*/
        return(0);
    else if (p->lchild==NULL)  /**p为被删结点,若它无左子树*/
    {
        if (f==NULL)             /**p是根结点,则用右孩子替换它*/
            bt=p->rchild;
        else if (f->lchild==p)    /**p是双亲结点的左孩子,则用其右孩子替换它*/
        {    f->lchild=p->rchild;
            free(p);
        }
        else if(f->rchild==p)    /**p是双亲结点的右孩子,则用其右孩子替换它*/
        {    f->rchild=p->rchild;
            free(p);
        }
    }
    else if (p->rchild==NULL)    /**p为被删结点,若它无右子树*/
    {
        if (f==NULL)             /**p是根结点,则用左孩子替换它*/
            bt=p->lchild;
        if (f->lchild==p)         /**p是双亲结点的左孩子,则用其左孩子替换它*/
        {    f->lchild=p->lchild;
            free(p);
        }
        else if(f->rchild==p)    /**p是双亲结点的右孩子,则用其左孩子替换它*/
        {    f->rchild=p->lchild;
            free(p);
        }
    }
    else                         /**p为被删结点,若它有左子树和右子树*/
    {
        f1=p;r=p->lchild;         /*查找*p的左子树中的最右下结点*r*/
        while (r->rchild!=NULL)    /**r一定是无右子树的结点,*f1作为r的双亲*/
        {    f1=r;
            r=r->rchild;
```

```
123.         }
124.         if (f1->lchild==r)        /**r是*f1的左孩子,删除*r*/
125.             f1->lchild=r->rchild;
126.         else if (f1->rchild==r)    /**r是*f1的右孩子,删除*r*/
127.             f1->rchild=r->lchild;
128.         r->lchild=p->lchild;    /*以下语句是用*r替代*p*/
129.         r->rchild=p->rchild;
130.         if (f==NULL)            /**f为根结点*/
131.             bt=r;                /*被删结点是根结点*/
132.         else if (f->lchild==p)    /**p是*f的左孩子*/
133.             f->lchild=r;
134.         else                    /**p是*f的右孩子*/
135.             f->rchild=r;
136.         free(p);
137.     }
138.     return(1);
139. }
140.
141. void main()
142. {
143.     BSTNode *bt=NULL,*p;
144.     KeyType a[]={10,6,12,8,3,20,9,25,15},k;
145.     int n=9;
146.     CreateBST(bt,a,n);
147.     printf("BST:");DispBST(bt);printf("\n");
148.     k=9;
149.     printf("查找关键字为%d的结点\n",k);
150.     p=BSTSearch(bt,k);
151.     if (p!=NULL)
152.         printf("存在关键字为%d结点\n",k);
153.     else
154.         printf("不存在关键字为%d结点\n",k);
155.     k=7;
156.     printf("插入关键字为%d的结点\n",k);
157.     BSTInsert(bt,k);
158.     printf("BST:");DispBST(bt);printf("\n");
159.     k=10;
160.     printf("删除关键字为%d的结点\n",k);
161.     BSTDelete(bt,k);
162.     printf("BST:");DispBST(bt);printf("\n");
163. }
```

哈希表查找

By C++

```
#include <stdio.h>
#define MaxSize 100        /*哈希表最大长度*/

typedef int KeyType;

typedef struct
{
    KeyType key;    /*关键字值*/
    int si;            /*探查次数*/
} HashTable;

void CreateHT(HashTable ht[],KeyType a[],int n,int m,int p)    /*构造哈希表*/
{
    int i,d,cnt;
    for (i=0;i<m;i++)  /*置初值*/
    {
        ht[i].key=0;
        ht[i].si=0;
    }
    for (i=0;i<n;i++)
    {
        cnt=1;            /*累计探查次数*/
        d=a[i]%p;
        if (ht[d].key==0)
        {
            ht[d].key=a[i];
            ht[d].si=cnt;
        }
        else
        {
            do            /*处理冲突*/
            {
                d=(d+1)%m;
                cnt++;
            } while (ht[d].key!=0);
            ht[d].key=a[i];
```

```
            ht[d].si=cnt;
        }
    }
}

void DispHT(HashTable ht[],int n,int m)    /*输出哈希表*/
{
    int i;
    double avg;
    printf("i:  ");
    for (i=0;i<m;i++)
        printf("%-3d",i);
    printf("\n");
    printf("key:");
    for (i=0;i<m;i++)
        printf("%-3d",ht[i].key);
    printf("\n");
    printf("si: ");
    for (i=0;i<m;i++)
        printf("%-3d",ht[i].si);
    printf("\n");
    avg=0;
    for (i=0;i<m;i++)
        avg+=ht[i].si;
    avg=avg/n;
    printf("平均查找长度:ASL(%d)=%g\n",n,avg);
}

void main()
{
    HashTable ht[MaxSize];
    KeyType a[]={19,1,23,14,55,20,84,27,68,11,10,77};
    int n=12,m=19,p=13;
    CreateHT(ht,a,n,m,p);
    DispHT(ht,n,m);
}
```

By Golang

```
// Package hashtable creates a ValueHashtable data structure for the Item type
package hashtable
```

```
3.
4.  import (
5.      "fmt"
6.      "sync"
7.  )
8.
9.  // Key the key of the dictionary
10. type Key interface{}
11.
12. // Value the content of the dictionary
13. type Value interface{}
14.
15. // ValueHashtable the set of Items
16. type ValueHashtable struct {
17.     items map[int]Value
18.     lock  sync.RWMutex
19. }
20.
21. // the hash() private function uses the famous Horner's method
22. // to generate a hash of a string with O(n) complexity
23. func hash(k Key) int {
24.     key := fmt.Sprintf("%s", k)
25.     h := 0
26.     for i := 0; i < len(key); i++ {
27.         h = 31*h + int(key[i])
28.     }
29.     return h
30. }
31.
32. // Put item with value v and key k into the hashtable
33. func (ht *ValueHashtable) Put(k Key, v Value) {
34.     ht.lock.Lock()
35.     defer ht.lock.Unlock()
36.     i := hash(k)
37.     if ht.items == nil {
38.         ht.items = make(map[int]Value)
39.     }
40.     ht.items[i] = v
41. }
42.
43. // Remove item with key k from hashtable
44. func (ht *ValueHashtable) Remove(k Key) {
```

```
45.     ht.lock.Lock()
46.     defer ht.lock.Unlock()
47.     i := hash(k)
48.     delete(ht.items, i)
49. }
50.
51. // Get item with key k from the hashtable
52. func (ht *ValueHashtable) Get(k Key) Value {
53.     ht.lock.RLock()
54.     defer ht.lock.RUnlock()
55.     i := hash(k)
56.     return ht.items[i]
57. }
58.
59. // Size returns the number of the hashtable elements
60. func (ht *ValueHashtable) Size() int {
61.     ht.lock.RLock()
62.     defer ht.lock.RUnlock()
63.     return len(ht.items)
64. }
```

哈希查找

```c
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100        /*哈希表最大长度*/

typedef int KeyType;

typedef struct node
{
    KeyType key;    /*关键字值*/
    int si;            /*探查次数*/
    struct node *next;
} Node;        /*数据结点类型*/

typedef struct
{
    Node *link;
} HNode;    /*头结点类型*/

void CreateHT(HNode *ht[],KeyType a[],int n,int p)    /*构造哈希表*/
{
    int i,d,cnt;
    Node *s,*q;
    for (i=0;i<p;i++)    /*所有头结点的link域置空*/
    {
        ht[i]=(HNode *)malloc(sizeof(HNode));
        ht[i]->link=NULL;
    }
    for (i=0;i<n;i++)
    {
        cnt=1;
        s=(Node *)malloc(sizeof(Node));    /*构造一个数据结点*/
        s->key=a[i];s->next=NULL;
        d=a[i]%p;                          /*求其哈希地址*/
        if (ht[d]->link==NULL)
        {
            ht[d]->link=s;
            s->si=cnt;
        }
```

```
        else
        {
            q=ht[d]->link;
            while (q->next!=NULL)
            {
                q=q->next;cnt++;
            }
            cnt++;
            s->si=cnt;q->next=s;
        }
    }
}

void DispHT(HNode *ht[],int n,int p)    /*输出哈希表*/
{
    int i,sum=0;
    Node *q;
    printf("哈希表:\n");
    for (i=0;i<p;i++)
    {
        q=ht[i]->link;
        printf("%d:link->",i);
        while (q!=NULL)
        {
            sum+=q->si;
            printf("[%d,%d]->",q->key,q->si);
            q=q->next;
        }
        printf("∧\n");
    }
    printf("\n平均查找长度:ASL=%g\n",1.0*sum/n);
}

void main()
{
    HNode *ht[MaxSize];
    KeyType a[]={87,25,310,8,27,132,68,95,187,123,70,63,47};
    int n=13,p=13;
    CreateHT(ht,a,n,p);
    DispHT(ht,n,p);
}
```

排序方法比较

排序方法	时间复杂度			空间 复杂度	稳定性	复杂性
	最坏情况	平均情况	最好情况			
直接插入排序	O(n^2)	O(n^2)	O(n)	O(1)	稳定	简单
希尔排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)	不稳定	较复杂
选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定	简单
堆排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)	不稳定	较复杂
冒泡排序	O(n^2)	O(n^2)	O(n)	O(1)	稳定	简单
快速排序	O(nlog2n)	O(n^2)	O(nlog2n)	O(nlog2n)	不稳定	较复杂
归并排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(n)	稳定	较复杂
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(n+r)	稳定	较复杂

- 直接插入排序
- 希尔排序

插入排序

```c
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;               /*关键字域*/
    ElemType data;             /*其他数据域*/
} LineList;                    /*线性表元素类型*/

void InsertSort(LineList R[],int n)
{
    int i,j;
    LineList tmp;
    for (i=1;i<n;i++)
    {
        tmp=R[i];
        j=i-1;
        while (j>=0 && tmp.key<R[j].key)/*元素后移,以便腾出一个位置插入tmp*/
        {
            R[j+1]=R[j];
            j--;
        }
        R[j+1]=tmp;        /*在j+1位置处插入tmp*/
    }
}

void main()
{
    LineList R[MaxSize];
    KeyType a[]={75,87,68,92,88,61,77,96,80,72};
    int n=10,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    printf("排序前:");
```

```
39.     for (i=0;i<n;i++)
40.         printf("%3d",R[i].key);
41.     printf("\n");
42.     InsertSort(R,n);
43.     printf("排序后:");
44.     for (i=0;i<n;i++)
45.         printf("%3d",R[i].key);
46.     printf("\n");
47. }
```

希尔排序

```c
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;               /*关键字域*/
    ElemType data;             /*其他数据域*/
} LineList;                    /*线性表元素类型*/

void ShellSort(LineList R[],int n)
{
    int i,j,gap;
    LineList tmp;
    gap=n/2;                    /*增量置初值*/
    while (gap>0)
    {
        for (i=gap;i<n;i++)      /*对所有相隔gap位置的所有元素组进行排序*/
        {
            tmp=R[i];
            j=i-gap;
            while (j>=0 && tmp.key<R[j].key)/*对相隔gap位置的元素组进行排序*/
            {
                R[j+gap]=R[j];
                j=j-gap;                    /*移到本组中的前一个元素*/
            }
            R[j+gap]=tmp;
            j=j-gap;
        }
        gap=gap/2;    /*减小增量*/
    }
}

void main()
{
```

```
39.     LineList R[MaxSize];
40.     KeyType a[]={75,87,68,92,88,61,77,96,80,72};
41.     int n=10,i;
42.     for (i=0;i<n;i++)
43.         R[i].key=a[i];
44.     printf("排序前:");
45.     for (i=0;i<n;i++)
46.         printf("%3d",R[i].key);
47.     printf("\n");
48.     ShellSort(R,n);
49.     printf("排序后:");
50.     for (i=0;i<n;i++)
51.         printf("%3d",R[i].key);
52.     printf("\n");
53. }
```

- 直接选择排序
- 堆排序

选择排序

基本思想

- 每次从未排序的序列中选择最小的数放置在该未排序序列的第一个
- 不断循环，直到排序完成

步骤

- 第一层循环: `i=0; i < len(list)-1; i++`
- 第二层循环: `j := i + 1; j < len(list); j++`
- 寻找未排序序列中最小的数: `if list[j] < list[minIndex] { minIndex = j }`
- 将最小的数与放置到当前未排序序列的第一个: `list[i], list[minIndex] = list[minIndex], list[i]`

By C++

```
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;              /*关键字域*/
    ElemType data;            /*其他数据域*/
} LineList;                   /*线性表元素类型*/

void SelectSort(LineList R[],int n)
{
    int i,j,k;
    LineList tmp;
    for (i=0;i<n-1;i++)
    {
        k=i;
        for (j=i+1;j<n;j++)
```

```
            if (R[j].key<R[k].key)
                k=j;        /*用k指出每趟在无序区段的最小元素*/
        tmp=R[i];            /*将R[k]与R[i]交换*/
        R[i]=R[k];
        R[k]=tmp;
    }
}

void main()
{
    LineList R[MaxSize];
    KeyType a[]={75,87,68,92,88,61,77,96,80,72};
    int n=10,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    printf("排序前:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
    SelectSort(R,n);
    printf("排序后:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
}
```

By Golang

```
package main

import (
    "fmt"
)

// 选择排序
func selectSort(list []int) []int {
    var minIndex int
    for i := 0; i < len(list)-1; i++ {
        minIndex = i
        for j := i + 1; j < len(list); j++ {
            if list[j] < list[minIndex] { // 寻找未排序序列中最小的数
```

```
                minIndex = j
            }
        }
        list[i], list[minIndex] = list[minIndex], list[i] // 将当前数与最小的数交换位置
        fmt.Println("第", i+1, "趟：", list)                 // 打印每趟的序列
    }
    return list
}

func main() {
    list := []int{75, 87, 68, 92, 88, 61, 77, 96, 80, 72}
    fmt.Println("初始序列:", list)
    result := selectSort(list)
    fmt.Println("最终结果:", result)
}
```

排序过程

```
初始序列: [75 87 68 92 88 61 77 96 80 72]

第 1 趟： [61 87 68 92 88 75 77 96 80 72]
第 2 趟： [61 68 87 92 88 75 77 96 80 72]
第 3 趟： [61 68 72 92 88 75 77 96 80 87]
第 4 趟： [61 68 72 75 88 92 77 96 80 87]
第 5 趟： [61 68 72 75 77 92 88 96 80 87]
第 6 趟： [61 68 72 75 77 80 88 96 92 87]
第 7 趟： [61 68 72 75 77 80 87 96 92 88]
第 8 趟： [61 68 72 75 77 80 87 88 92 96]
第 9 趟： [61 68 72 75 77 80 87 88 92 96]

最终结果: [61 68 72 75 77 80 87 88 92 96]
```

堆排序

基本思想

步骤

By C++

```
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;              /*关键字域*/
    ElemType data;            /*其他数据域*/
} LineList;                   /*线性表元素类型*/

void Sift(LineList R[],int low,int high)
{
    int i=low,j=2*i;            /*R[j]是R[i]的左孩子*/
    LineList tmp=R[i];
    while (j<=high)
    {    if (j<high && R[j].key<R[j+1].key)     /*若右孩子较大,把j指向右孩子*/
            j++;                    /*j变为2i+1,指向右孩子结点*/
        if (tmp.key<R[j].key)
        {    R[i]=R[j];           /*将R[j]调整到双亲结点位置上*/
            i=j;                  /*修改i和j值,以便继续向下筛选*/
            j=2*i;
        }
        else break;               /*筛选结束*/
    }
    R[i]=tmp;                     /*被筛选结点的值放入最终位置*/
}

```

```
void HeapSort(LineList R[],int n)
{
    int i;
    LineList tmp;
    for (i=n/2;i>=1;i--)      /*循环建立初始堆*/
        Sift(R,i,n);
    for (i=n;i>=2;i--)       /*进行n-1次循环,完成堆排序*/
    {      tmp=R[1];            /*将第一个元素同当前区间内R[1]对换*/
        R[1]=R[i];
        R[i]=tmp;
        Sift(R,1,i-1);         /*筛选R[1]结点,得到i-1个结点的堆*/
    }
}

void main()
{
    LineList R[MaxSize];
    KeyType a[]={0,75,87,68,92,88,61,77,96,80,72};    /*有效数据从a[1]开始*/
    int n=10,i;
    for (i=0;i<=n;i++)
        R[i].key=a[i];
    printf("排序前:");
    for (i=1;i<=n;i++)
        printf("%3d",R[i].key);
    printf("\n");
    HeapSort(R,n);
    printf("排序后:");
    for (i=1;i<=n;i++)
        printf("%3d",R[i].key);
    printf("\n");
}
```

By Golang

`go`

- 冒泡排序
- 快速排序

冒泡排序

基本思想

- 比较相邻两个元素，如果第一个比第二个大，就交换两者的顺序
- 对每一对相邻的元素做同样的操作，从最后一对到第一对，每一趟结束最小的元素会排在第一个（即冒泡）
- 从未排序的元素开始循环做以上操作，直到排序完成

步骤

1. 第一层循环：i=0; i<(len-1); i++
2. 第二层循环：j=len-1; j>i; j—
3. 比较相邻两个元素：list[j-1]和list[j]
4. 如果前者比后者大，就交换顺序：list[j-1], list[j] = list[j], list[j-1]

By C++

```
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;              /*关键字域*/
    ElemType data;            /*其他数据域*/
} LineList;                   /*线性表元素类型*/

void BubbleSort(LineList R[],int n)
{
    int i,j,exchange;
    LineList tmp;
    for (i=0;i<n-1;i++)
    {    exchange=0;
        for (j=n-1;j>i;j--)    /*比较,找出最小关键字的记录*/
```

```
            if (R[j].key<R[j-1].key)
            {    tmp=R[j];  /*R[j]与R[j-1]进行交换,将最小关键字记录前移*/
                R[j]=R[j-1];
                R[j-1]=tmp;
                exchange=1;
            }
        if (exchange==0)     /*本趟未发生交换时结束算法*/
            return;
    }
}

void main()
{
    LineList R[MaxSize];
    KeyType a[]={75,87,68,92,88,61,77,96,80,72};
    int n=10,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    printf("排序前:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
    BubbleSort(R,n);
    printf("排序后:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
}
```

By Golang

```
package main

import (
    "fmt"
)

// 冒泡排序, 从小到大
func bubbleSort(list []int) []int {
    for i := 0; i < len(list)-1; i++ {
        for j := len(list) - 1; j > i; j-- {  // 从最后两个元素开始比较
```

```
            if list[j-1] > list[j] { // 相邻元素对比
                list[j-1], list[j] = list[j], list[j-1] // 如果后者比前者小，就交互
位置
            }
        }
        fmt.Println("第", i+1, "趟：", list)
    }
    return list
}

func main() {
    list := []int{75, 87, 68, 92, 88, 61, 77, 96, 80, 72}
    fmt.Println("初始序列:", list)
    result := bubbleSort(list)
    fmt.Println("最终结果:", result)
}
```

排序过程

```
初始序列: [75 87 68 92 88 61 77 96 80 72]

第 1 趟： [61 75 87 68 92 88 72 77 96 80]
第 2 趟： [61 68 75 87 72 92 88 77 80 96]
第 3 趟： [61 68 72 75 87 77 92 88 80 96]
第 4 趟： [61 68 72 75 77 87 80 92 88 96]
第 5 趟： [61 68 72 75 77 80 87 88 92 96]
第 6 趟： [61 68 72 75 77 80 87 88 92 96]
第 7 趟： [61 68 72 75 77 80 87 88 92 96]
第 8 趟： [61 68 72 75 77 80 87 88 92 96]
第 9 趟： [61 68 72 75 77 80 87 88 92 96]

最终结果: [61 68 72 75 77 80 87 88 92 96]
```

快速排序

基本思想

- 分治：在未排序序列中选择一个基准数（一般为第一个），将小于基准数的放在其左边，大于基准数的放在其右边，即分成两个区间
- 递归：将上述两个区间，每个区间内执行上述操作，以此类推，直到排序完成

步骤

- 选择一个基准数，一般为第一个：`pivot := list[left]`
- 将小于基准数的放在其左边
- 将大于基准数的放在其右边
- 对每个分区执行递归操作：`quickSort(list, left, i-1); quickSort(list, i+1, right)`

By C++

```
#include <stdio.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;               /*关键字域*/
    ElemType data;             /*其他数据域*/
} LineList;                    /*线性表元素类型*/

void QuickSort(LineList R[],int s,int t) /*对R[s]至R[t]的元素进行快速排序*/
{
    int i=s,j=t;
    LineList tmp;
    if (s<t)                  /*区间内至少存在一个元素的情况*/
    {    tmp=R[s];             /*用区间的第1个记录作为基准*/
        while (i!=j)          /*从区间两端交替向中间扫描,直至i=j为止*/
```

```
        {    while (j>i && R[j].key>tmp.key)
                j--;           /*从右向左扫描,找第1个关键字小于tmp.key的R[j]*/
            R[i]=R[j];         /*找到这样的R[j],则R[i]和R[j]交换*/
            while (i<j && R[i].key<tmp.key)
                i++;         /*从左向右扫描,找第1个关键字大于tmp.key的R[i]*/
            R[j]=R[i];         /*找到这样的R[i],则R[i]和R[j]交换*/
        }
        R[i]=tmp;
        QuickSort(R,s,i-1);    /*对左区间递归排序*/
        QuickSort(R,i+1,t);    /*对右区间递归排序*/
    }
}

void main()
{
    LineList R[MaxSize];
    KeyType a[]={75,87,68,92,88,61,77,96,80,72};
    int n=10,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    printf("排序前:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
    QuickSort(R,0,n-1);
    printf("排序后:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
}
```

By Golang

```
package main

import (
    "fmt"
)

func quickSort(list []int, left, right int) []int {
    if left < right {
```

```
        i, j := left, right
        pivot := list[left] // 以左边第一个数作为基准数，将第一个数暂存在pivot变量中

        // 分治
        for i < j {
            // 从右向左找出第一个小于基准数的，将list[j]赋值给list[i]
            for j > i && list[j] > pivot {
                j--
            }
            list[i] = list[j]

            // 从左向右找出第一个大于基准数的，将list[i]赋值给list[j]
            for i < j && list[i] < pivot {
                i++
            }
            list[j] = list[i]
        }
        list[i] = pivot
        fmt.Println(list) // 打印该趟的序列

        // 递归
        quickSort(list, left, i-1)
        quickSort(list, i+1, right)
    }
    return list
}

func main() {
    list := []int{75, 87, 68, 92, 88, 61, 77, 96, 80, 72}
    fmt.Println("初始序列:", list)
    result := quickSort(list, 0, len(list)-1)
    fmt.Println("最终结果:", result)
}
```

排序过程

```
初始序列: [75 87 68 92 88 61 77 96 80 72]

第 1 趟：[72 61 68 75 88 92 77 96 80 87]
第 2 趟：[68 61 72 75 88 92 77 96 80 87]
第 3 趟：[61 68 72 75 88 92 77 96 80 87]
```

```
第 4 趟：[61 68 72 75 87 80 77 88 96 92]
第 5 趟：[61 68 72 75 77 80 87 88 96 92]
第 6 趟：[61 68 72 75 77 80 87 88 96 92]
第 7 趟：[61 68 72 75 77 80 87 88 92 96]

最终结果: [61 68 72 75 77 80 87 88 92 96]
```

归并排序

```c
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100

typedef int KeyType;        /*关键字类型*/

typedef char ElemType[10];    /*其他数据项类型*/

typedef struct
{
    KeyType key;              /*关键字域*/
    ElemType data;            /*其他数据域*/
} LineList;                   /*线性表元素类型*/

void Merge(LineList R[],int low,int mid,int high)
{
    LineList *R1;
    int i=low,j=mid+1,k=0; /*k是R1的下标,i、j分别为第1、2段的下标*/
    R1=(LineList *)malloc((high-low+1)*sizeof(LineList));  /*动态分配空间*/
    while (i<=mid && j<=high)    /*在第1段和第2段均未扫描完时循环*/
        if (R[i].key<=R[j].key)  /*将第1段中的记录放入R1中*/
        {    R1[k]=R[i];
            i++;k++;
        }
        else                            /*将第2段中的记录放入R1中*/
        {    R1[k]=R[j];
            j++;k++;
        }
        while (i<=mid)              /*将第1段余下部分复制到R1*/
        {
            R1[k]=R[i];
            i++;k++;
        }
        while (j<=high)           /*将第2段余下部分复制到R1*/
        {    R1[k]=R[j];
            j++;k++;
        }
        for (k=0,i=low;i<=high;k++,i++) /*将R1复制回R中*/
```

```
            R[i]=R1[k];
}

void MergePass(LineList R[],int length,int n)
{
    int i;
    for (i=0;i+2*length-1<n;i=i+2*length)     /*归并length长的两相邻子表*/
        Merge(R,i,i+length-1,i+2*length-1);
    if (i+length-1<n)                         /*余下两个子表,后者长度小于length*/
        Merge(R,i,i+length-1,n-1);     /*归并这两个子表*/
}

void MergeSort(LineList R[],int n)    /*二路归并算法*/
{
    int length;
    for (length=1;length<n;length=2*length)
        MergePass(R,length,n);
}

void main()
{
    LineList R[MaxSize];
    KeyType a[]={75,87,68,92,88,61,77,96,80,72};
    int n=10,i;
    for (i=0;i<n;i++)
        R[i].key=a[i];
    printf("排序前:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
    MergeSort(R,n);
    printf("排序后:");
    for (i=0;i<n;i++)
        printf("%3d",R[i].key);
    printf("\n");
}
```

基数排序

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#define MAXE 20            /*线性表中最多元素个数*/
#define MAXR 10            /*基数的最大取值*/
#define MAXD 8            /*关键字位数的最大取值*/

typedef struct node
{
    char data[MAXD];    /*记录的关键字定义的字符串*/
       struct node *next;
} RecType;

void RadixSort(RecType *&p,int r,int d)
/*p为待排序序列链表指针,r为基数,d为关键字位数*/
{
    RecType *head[MAXR],*tail[MAXR],*t;/*定义各链队的首尾指针*/
    int i,j,k;
    for (i=d-1;i>=0;i--)                    /*从低位到高位做d趟排序*/
    {    for (j=0;j<r;j++)                  /*初始化各链队首、尾指针*/
            head[j]=tail[j]=NULL;
        while (p!=NULL)                    /*对于原链表中每个结点循环*/
        {    k=p->data[i]-'0';             /*找第k个链队*/
            if (head[k]==NULL)             /*进行分配,即采用尾插法建立单链表*/
            {
                head[k]=p;
                tail[k]=p;
            }
              else
            {
                tail[k]->next=p;
                tail[k]=p;
            }
            p=p->next;                     /*取下一个待排序的元素*/
        }
        p=NULL;
           for (j=0;j<r;j++)                /*对于每一个链队循环*/
            if (head[j]!=NULL)             /*进行收集*/
```

```
            {    if (p==NULL)
                 {    p=head[j];
                      t=tail[j];
                 }
                 else
                 {    t->next=head[j];
                      t=tail[j];
                 }
            }
        t->next=NULL;                    /*最后一个结点的next域置NULL*/
    }
}

void main()
{
    RecType *h=NULL,*p,*t;
    char *A[]={"75","87","68","92","88","61","77","96","80","72"};
    int i,n=10;
    for (i=0;i<n;i++)          /*构造不带头结点的单链表h*/
    {
        p=(RecType *)malloc(sizeof(RecType));
        strcpy(p->data,A[i]);
        if (h==NULL)
        {
            h=p;
            t=h;
        }
        else
        {
            t->next=p;
            t=p;
        }
    }
    t->next=NULL;
    printf("排序前:");
    for (i=0;i<n;i++)
        printf("%3s",A[i]);
    printf("\n");
    RadixSort(h,10,2);
    printf("排序后:");
    p=h;
    while (p!=NULL)
```

```
81.     {
82.         printf("%3s",p->data);
83.         p=p->next;
84.     }
85.     printf("\n");
86. }
```