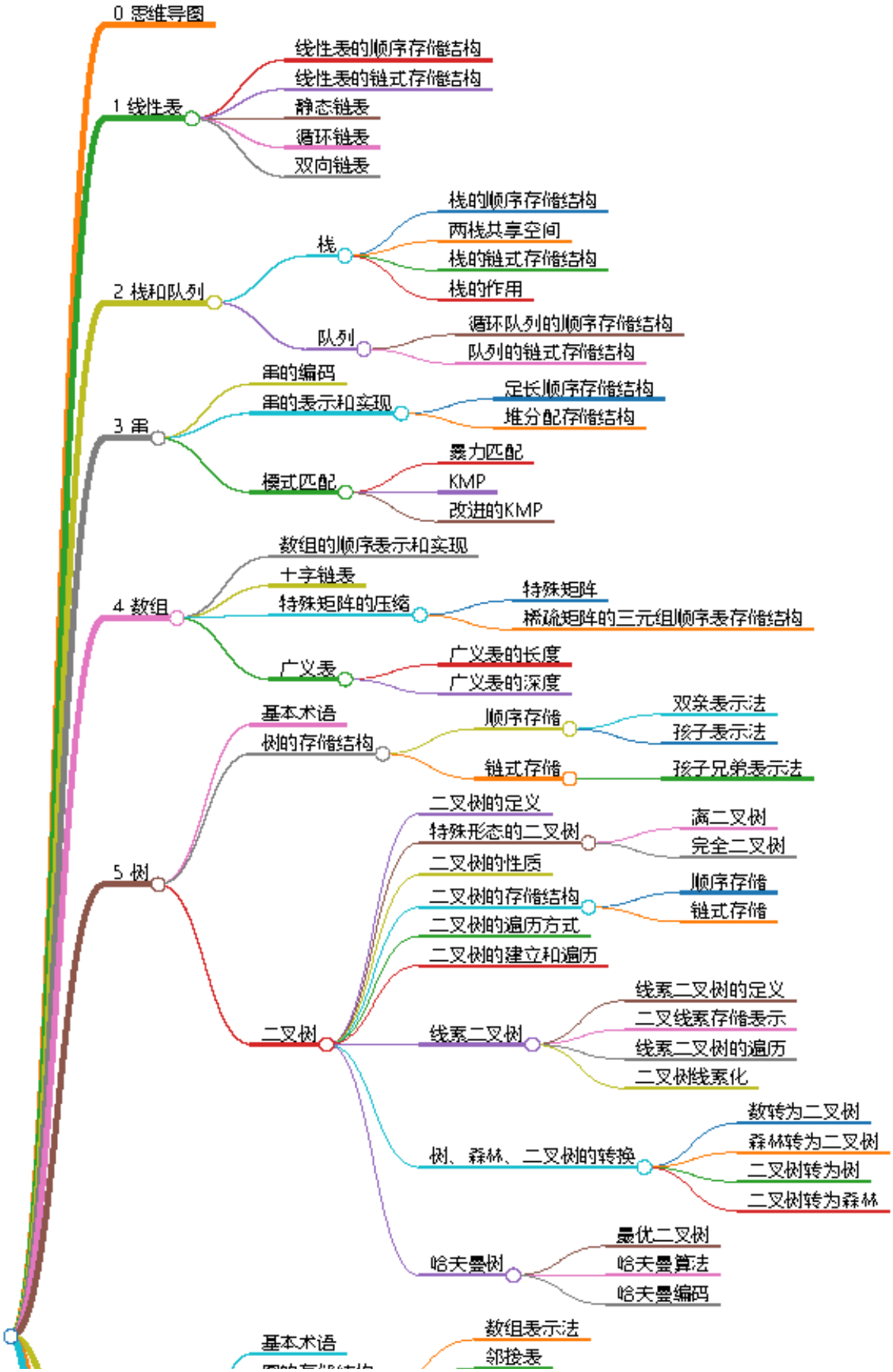
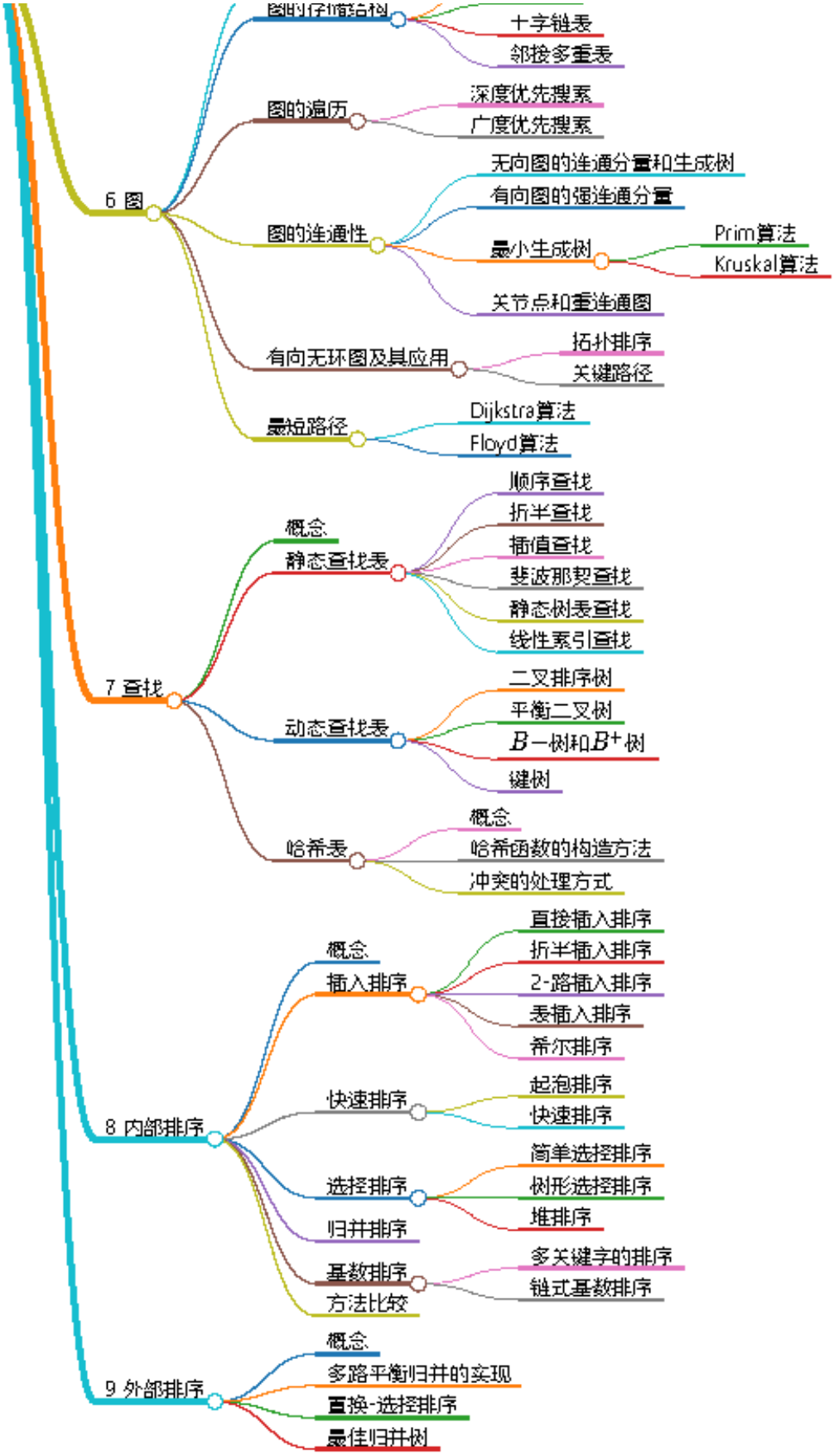
0 思维导图


0 思维导图
1 线性表
线性表的顺序存储结构
线性表的链式存储结构
静态链表
循环链表
双向链表
2 栈和队列
栈
栈的顺序存储结构
两栈共享空间
栈的链式存储结构
栈的作用
队列
循环队列的顺序存储结构
队列的链式存储结构
3 串
串的编码
串的表示和实现
定长顺序存储结构
堆分配存储结构
模式匹配
暴力匹配
KMP
改进的KMP
4 数组
数组的顺序表示和实现
十字链表
特殊矩阵的压缩
特殊矩阵
稀疏矩阵的三元组顺序表存储结构
广义表
广义表的长度
广义表的深度
5 树
基本术语
树的存储结构
顺序存储
双亲表示法
孩子表示法
链式存储
孩子兄弟表示法
二叉树
二叉树的定义
特殊形态的二叉树
满二叉树
完全二叉树
二叉树的性质
二叉树的存储结构
顺序存储
链式存储
二叉树的遍历方式
二叉树的建立和遍历
线索二叉树
线索二叉树的定义
二叉线索存储表示
线索二叉树的遍历
二叉树线索化
树、森林、二叉树的转换
树转为二叉树
森林转为二叉树
二叉树转为树
二叉树转为森林
哈夫曼树
最优二叉树
哈夫曼算法
哈夫曼编码
基本术语
数组表示法
邻接表


6 图
图的存储结构
十字链表
邻接多重表
图的遍历
深度优先搜索
广度优先搜索
图的连通性
无向图的连通分量和生成树
有向图的强连通分量
最小生成树
Prim算法
Kruskal算法
关节点和重连通图
有向无环图及其应用
拓扑排序
关键路径
最短路径
Dijkstra算法
Floyd算法
7 查找
概念
静态查找表
顺序查找
折半查找
插值查找
斐波那契查找
静态树表查找
线性索引查找
动态查找表
二叉排序树
平衡二叉树
$B-$树和B^+树
键树
哈希表
概念
哈希函数的构造方法
冲突的处理方式
8 内部排序
概念
插入排序
直接插入排序
折半插入排序
2-路插入排序
表插入排序
希尔排序
快速排序
起泡排序
快速排序
选择排序
简单选择排序
树形选择排序
堆排序
归并排序
基数排序
多关键字的排序
链式基数排序
方法比较
9 外部排序
概念
多路平衡归并的实现
置换-选择排序
最佳归并树

1 线性表

线性表的顺序存储结构

```c
//线性表的顺序存储结构
# include <stdio.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

# define MAXSIZE 20

typedef int Status;

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

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

//插入操作
//在第i个位置之前插入新的数据元素e，L的长度加1
Status ListInsert(SqList *L, int i, ElemType e) {
  int k;
  if (L->length == MAXSIZE)
    return ERROR;
  if (i < 1 || i > L->length + 1)
    return ERROR;
//  插入的位置不在表尾
  if (i <= L->length) {
//    将要插入位置后数据元素向右移动一位
    for (k = L->length - 1; k >= i - 1; k--) {
      L->data[k + 1] = L->data[k];
    }
  }
  L->data[i - 1] = e;
  L->length++;
  return OK;
}
```

```c
//删除操作
//删除L的第i个数据元素，并用e返回其值，L的长度减1
Status ListDelete(SqList *L, int i, ElemType *e) {
  int k;
  if (L->length == 0)
    return ERROR;
  if (i < 1 || i > L->length)
    return ERROR;
  *e = L->data[i - 1];
  if (i < L->length) {
    for (k = i, k < L->length; k++) {
      L->data[k - 1] = L->data[k];
    }
  }
  L->length--;
  return OK;
}


int main() {

  return 0;
}
```

线性表的链式存储结构

```c
//线性表的链式存储结构
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int ElemType;

typedef struct Node {
  ElemType data;
  struct Node *next;
} Node;

typedef struct Node *LinkList;

//读取第i个元素的值
Status GetElem(LinkList L, int i, ElemType *e) {
```

```
  int j;
  LinkList p;
  p = L->next;
  j = 1;
  while (p && j < i) {
    p = p->next;
    ++j;
  }
  if (!p || j > i)
    return ERROR;
  *e = p->data;
  return OK;
}

//单链表的插入
//在L中第i个结点位置之前插入新的数据元素e
Status ListInsert(LinkList *L, int i, ElemType e) {
  int j;
  LinkList p, s;
  p = *L;
  j = 1;
//  寻找第i-1个结点
  while (p && j < i) {
    p = p->next;
    ++j;
  }
  if (!p || j > i)
    return ERROR;
  s = (LinkList)malloc(sizeof(Node));
  s->data = e;
  s->next = p->next;
  p->next = s;
  return OK;
}

//单链表的删除
//删除L的第i个结点
Status ListDelete(LinkList *L, int i, ElemType *e) {
  int j;
  LinkList p, q;
  p = *L;
  j = 1;
//  寻找第i-1个结点
  while (p->next && j < i) {
    p = p->next;
    ++j;
  }
  if (!(p->next) || j > i)
    return ERROR;
```

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


//单链表的整表创建
//头插法
void CreateListHead(LinkList *L, int n) {
  LinkList p;
  int i;
  srand(time(0));
  *L = (LinkList)malloc(sizeof(Node));
  (*L)->next = NULL;
  for (i = 0; i < n; i++) {
    p = (LinkList)malloc(sizeof(Node));
    p->data = rand() % 100 + 1;
    p->next = (*L)->next;
    (*L)->next = p;
  }
}

//尾插法
void CreateListTail(LinkList *L, int n) {
  LinkList p, r;
  int i;
  srand(time(0));
  *L = (LinkList)malloc(sizeof(Node));
  (*L)->next = NULL;
  r = *L;

  for (i = 0; i < n; i++) {
    p = (LinkList)malloc(sizeof(Node));
    p->data = rand() % 100 + 1;
    r->next = p;
    r = p;
  }
  r->next = NULL;
}

//链表的整表删除
Status ClearList(LinkList *L) {
  LinkList p, q;
  p = (*L)->next;
  while (p) {
    q = q->next;
    free(p);
```

```
    p = q;
  }
  (*L)->next = NULL;
  return OK;
}

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

int main() {
  LinkList p;
    CreateListHead(&p, 100);
    ListTraverse(p);
  return 0;
}
```

静态链表

数组的一个分量表示结点，用游标代替指针指示结点在数组中的相对位置。

```
//静态链表
//用数组描述的链表
//游标实现法
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int ElemType;

# define MAXSIZE 1000
typedef struct {
  ElemType data;
  int cur;
} Component, StaticLinkList[MAXSIZE];


//初始化
Status InitList(StaticLinkList space) {
```

```
  int i;
  for (i = 0; i < MAXSIZE - 1; i++)
    space[i].cur = i + 1;
  space[MAXSIZE - 1].cur = 0;
  return OK;
}

//未被使用的数组元素成为备用链表
//数组中的第一个元素，即下标为0的元素的cur就存放备用链表第一个结点的下标
//数组的最后一个元素的cur则存放第一个有数值的元素的下标

//返回分配的结点下标
int Malloc_SLL(StaticLinkList space) {
  int i = space[0].cur;
  if (space[0].cur)
    space[0].cur = space[i].cur;
  return i;
}

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

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

Status ListInsert(StaticLinkList L, int i, ElemType e) {
  int j, k, l;
//  最后一个元素的下标
  k = MAXSIZE - 1;
  if (i < 1 || i > ListLength(L) + 1)
    return ERROR;
  j = Malloc_SLL(L);
  if (j) {
    L[j].data = e;
    for (l = 1; l <= i - 1; l++)
      k = L[k].cur;
    L[j].cur = L[k].cur;
```

```
    L[k].cur = j;
    return OK;
  }
  return ERROR;
}

//删除第i个数据元素
Status ListDelete(StaticLinkList L, int i) {
  int j, k;
  if (i < 1 || i > ListLength(L))
    return ERROR;
  k = MAXSIZE - 1;
  for (j = 1; j <= i - 1; j++)
    k = L[k].cur;
  j = L[k].cur;
  L[k].cur = L[j].cur;
  Free_SLL(L, j);
  return OK;
}

int main() {
  StaticLinkList s;
  InitList(s);
  ListInsert(s, 1, 11);
  ListDelete(s, 1);
  ListInsert(s, 1, 22);
  ListInsert(s, 1, 33);
  printf("Length = %d\n", ListLength(s));
//  遍历
  for (int i = 1; i <= ListLength(s); i++) {
    printf("%d %d\n", i, s[i].data);
  }
  return 0;
}
```

循环链表

Circular linked list

循环链表和单链表的主要差异就在于循环的判断条件上，原来是判断`p->next`是否为空，现在则是`p->next`不等于头结点，则循环结束。

```
//循环链表
//单循环链表往往设立尾指针而不是头指针
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
```

```c
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
# define OVERFLOW -2

typedef int Status;
typedef int Boolean;
typedef int ElemType;

typedef struct LNode {
  ElemType data;
  LNode *next;
} LNode, *LinkList;

//构造一个空的线性表L
void InitList(LinkList &L) {
  L = (LinkList)malloc(sizeof(LNode));
  if (!L)
    exit(OVERFLOW);
  L->next = L;  //头结点的指针域指向头结点
}

//将L重置为空表
void ClearList(LinkList &L) {
  LinkList p, q;
  L = L->next;
  p = L->next;
//  未到表尾
  while (p != L) {
    q = p->next;
    free(p);
    p = q;
  }
  L->next = L;
}

//销毁线性表L
void DestoryList(LinkList &L) {
  ClearList(L);
  free(L);
  L = NULL;
}

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

```
int ListLength(LinkList L) {
  int i = 0;
  LinkList p = L->next;
  while (p != L) {
    i++;
    p = p->next;
  }
  return i;
}

Status GetElem(LinkList L, int i, ElemType &e) {
  int j = 1;
  LinkList p = L->next->next;
  if (i <= 0 || i > ListLength(L))
    return ERROR;
  while (j < i) {
    j++;
    p = p->next;
  }
  e = p->data;
  return OK;
}

//返回L中第一个与e满足关系compare()的数据元素的位序，不存在则返回0
int LocateElem(LinkList L, ElemType e, Status(* compare)(ElemType, ElemType)) {
  int i = 0;
  LinkList p = L->next->next;
//  p未指向头结点
  while (p != L->next) {
    i++;
    if (compare(p->data, e))
      return i;
    p = p->next;
  }
  return 0;
}

//若cur_e是L中的数据元素，且不是第一个，则用pre_e返回它的前驱，返回OK
//否则操作失败，pre_e无定义，返回ERROR
Status PriorElem(LinkList L, ElemType cur_e, ElemType &pre_e) {
  LinkList q, p = L->next->next;
  q = p->next;
  while (q != L->next) {
    if (q->data == cur_e) {
      pre_e = p->data;
      return OK;
    }
    p = q;
```

```
    q = q->next;
  }
  return ERROR;
}

//若cur_e是L中的数据元素，且不是最后一个，则用next_e返回它的后继，返回OK
//否则操作失败，next_e无定义，返回ERROR
Status NextElem(LinkList L, ElemType cur_e, ElemType &next_e) {
  LinkList p = L->next->next;
  while (p != L->next) {
    if (p->data == cur_e) {
      next_e = p->next->data;
      return OK;
    }
    p = p->next;
  }
  return ERROR;
}

//在L的第i个位置之前插入元素e
Status ListInsert(LinkList &L, int i, ElemType e) {
  LinkList p = L->next, s;
  int j = 0;
  if (i < 0 || i > ListLength(L) + 1)
    return ERROR;
//  寻找第i-1个结点
  while (j < i - 1) {
    j++;
    p = p->next;
  }
  s = (LinkList)malloc(sizeof(LNode));
  s->data = e;
  s->next = p->next;
  p->next = s;
//  如果插在表尾
  if (p == L)
    L = s;
  return OK;
}

Status ListDelete(LinkList &L, int i, ElemType &e) {
  LinkList p = L->next, q;
  int j = 0;
  if (i < 0 || i > ListLength(L))
    return ERROR;
  while (j < i - 1) {
    j++;
    p = p->next;
  }
```

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

//依次对L的每个数据元素调用函数vi()
void ListTraverse(LinkList L, void(*vi)(ElemType)) {
  LinkList p = L->next->next;
  while (p != L->next) {
    vi(p->data);
    p = p->next;
  }
  printf("\n");
}

int main() {

  return 0;
}
```

双向链表

Double linked list

```
//双向链表
//也是循环的，第一个结点的前驱是最后一个结点；最后一个结点的后继是第一个结点
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
# define OVERFLOW -2

typedef int Status;
typedef int Boolean;
typedef int ElemType;

typedef struct DuLNode {
  ElemType data;
  DuLNode *prior, *next;
} DuLNode, *DuLinkList;
```

```
void InitList(DuLinkList &L) {
  L = (DuLinkList)malloc(sizeof(DuLNode));
  if (L)
    L->prior = L->next = L;
  else
    exit(OVERFLOW);
}

void ClearList(DuLinkList L) {
  DuLinkList p = L->next;
  while (p != L) {
    p = p->next;
    free(p->prior);
  }
  L->prior = L->next = L;
}

void DestroyList(DuLinkList &L) {
  ClearList(L);
  free(L);
  L = NULL;
}

Status ListEmpty(DuLinkList L) {
  if (L->next == L && L->prior == L)
    return TRUE;
  else
    return FALSE;
}

int ListLength(DuLinkList L) {
  int i = 0;
  DuLinkList p = L->next;
  while (p != L) {
    i++;
    p = p->next;
  }
  return i;
}

Status GetElem(DuLinkList L, int i, ElemType &e) {
  int j = 1;
  DuLinkList p = L->next;
  while (p != L && j < i) {
    j++;
    p = p->next;
  }
//  第i个元素不存在
  if (p == L || j > i)
```

```
    return ERROR;
  e = p->data;
  return OK;
}

int LocateElem(DuLinkList L, ElemType e, Status(* compare)(ElemType, ElemType)) {
  int i = 0;
  DuLinkList p = L->next;
  while (p != L) {
    i++;
    if (compare(p->data, e))
      return i;
    p = p->next;
  }
  return 0;
}

Status PriorElem(DuLinkList L, ElemType cur_e, ElemType &pre_e) {
  DuLinkList p = L->next->next;
  while (p != L) {
    if (p->data == cur_e) {
      pre_e = p->prior->data;
      return OK;
    }
    p = p->next;
  }
  return ERROR;
}

Status NextElem(DuLinkList L, ElemType cur_e, ElemType &next_e) {
  DuLinkList p = L->next->next;
  while (p != L) {
    if (p->prior->data == cur_e) {
      next_e = p->data;
      return OK;
    }
    p = p->next;
  }
  return ERROR;
}

DuLinkList GetElemP(DuLinkList L, int i) {
  int j;
  DuLinkList p = L;
  if (i < 0 || i > ListLength(L))
    return NULL;
  for (j = 1; j <= i; j++)
    p = p->next;
  return p;
```

```
}

Status ListInsert(DuLinkList &L, int i, ElemType e) {
  DuLinkList p, s;
  if (i < 1 || i > ListLength(L) + 1)
    return ERROR;
//  第i个结点的前驱
  p = GetElemP(L, i - 1);
  if (!p)
    return ERROR;
  s = (DuLinkList)malloc(sizeof(DuLNode));
  if (!s)
    return ERROR;
  s->data = e;
  s->next = p->next;
  s->prior = p;
  p->next->prior = s;
  p->next = s;
  return OK;
}

Status ListDelete(DuLinkList &L, int i, ElemType &e) {
  DuLinkList p;
  if (i < 1)
    return ERROR;
  p = GetElemP(L, i);
  if (!p)
    return ERROR;
  e = p->data;
  p->prior->next = p->next;
  p->next->prior = p->prior;
  free(p);
  return OK;
}

void ListTraverse(DuLinkList L, void(*vi)(ElemType)) {
  DuLinkList p = L->next;
  while (p != L) {
    vi(p->data);
    p = p->next;
  }
  printf("/n");
}

void ListTraverseBack(DuLinkList L, void(*vi)(ElemType)) {
  DuLinkList p = L->prior;
  while (p != L) {
    vi(p->data);
```

```
    p = p->prior;
  }
  printf("/n");
}

int main() {

  return 0;
}
```

2 栈和队列

栈

- 允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)。
- 栈又称后进先出(Last In First Out)的线性表。

栈的顺序存储结构

```
//栈的顺序存储结构
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int SElemType;

# define MAXSIZE 1000

typedef struct {
  SElemType data[MAXSIZE];
  int top;
} SqStack;

Status Push(SqStack *S, SElemType e) {
  if (S->top == MAXSIZE - 1)
    return ERROR;
  S->top++;
  S->data[S->top] = e;
  return OK;
}

//当栈存在一个元素时，top等于0，通常空栈的判断条件为top等于-1
Status Pop(SqStack *S, SElemType *e) {
```

```
  if (S->top == -1)
    return ERROR;
  *e = S->data[S->top];
  S->top--;
  return OK;
}

Status StackTraverse(SqStack S) {
  int i = S.top;
  while (i > -1) {
    printf("%d\n", S.data[i--]);
  }
  return OK;
}

int main() {
  SqStack s;
  s.top = -1;
  Push(&s, 1);
  Push(&s, 2);
  int e;
  Pop(&s, &e);
  printf("%d\n", e);
  StackTraverse(s);

  return 0;
}
```

两栈共享空间

```
//栈的顺序存储结构
//两栈共享空间
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int SElemType;

# define MAXSIZE 1000

typedef struct {
  SElemType data[MAXSIZE];
  int top1;
  int top2;
```

```
} SqDoubleStack;

Status Push(SqDoubleStack *S, SElemType e, int stackNumber) {
  if (S->top1 + 1 == S->top2)
    return ERROR;
  if (stackNumber == 1)
    S->data[++S->top1] = e;
  else if (stackNumber == 2)
    S->data[--S->top2] = e;
  return OK;
}

Status Pop(SqDoubleStack *S, SElemType *e, int stackNumber) {
  if (stackNumber == 1) {
    if (S->top1 == -1)
      return ERROR;
    *e = S->data[S->top1--];
  }
  else if (stackNumber == 2) {
    if (S->top2 == MAXSIZE)
      return ERROR;
    *e = S->data[S->top2++];
  }
  return OK;
}

int main() {
  SqDoubleStack S;
  S.top1 = -1;
    S.top2 = MAXSIZE;
  Push(&S, 1, 1);
  printf("%d", S.data[0]);
  return 0;
}
```

栈的链式存储结构

```
//栈的链式存储结构
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int SElemType;
```

```c
# define MAXSIZE 1000

typedef struct StackNode {
  SElemType data;
  struct StackNode *next;
} StackNode, *LinkStackPtr;

typedef struct LinkStack {
  LinkStackPtr top;
  int count;
} LinkStack;

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

Status Push(LinkStack *S, SElemType e) {
  LinkStackPtr p = (LinkStackPtr)malloc(sizeof(StackNode));
  p->data = e;
  p->next = S->top;
  S->top = p;
  S->count++;
  return OK;
}

Status Pop(LinkStack *S, SElemType *e) {
  LinkStackPtr p;
  if (StackEmpty(*S))
    return ERROR;
  *e = S->top->data;
  p = S->top;
  S->top = S->top->next;
  free(p);
  S->count--;
  return OK;
}

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

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

int main() {
  LinkStack s;
  InitStack(&s);
  Push(&s, 1);
  Push(&s, 2);
  Push(&s, 3);
  int e;
  Pop(&s, &e);
  StackTraverse(s);

  return 0;
}
```

栈的作用

- 递归
- 逆波兰表达式

队列

- 允许插入的一端称为队尾，允许删除的一端称为队头。
- 先进先出(First In First Out)。

循环队列的顺序存储结构

注意判断队列满的条件和计算队列长度的公式。

```
//循环队列的顺序存储结构
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int QElemType;

# define MAXSIZE 1000
```

```c
typedef struct {
  QElemType data[MAXSIZE];
  int front;  //头指针
  int rear; //尾指针
} SqQueue;

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

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

//入队
Status EnQueue(SqQueue *Q, QElemType e) {
//  队列满的判断
  if ((Q->rear + 1) % MAXSIZE == Q->front)
    return ERROR;
  Q->data[Q->rear] = e;
  Q->rear = (Q->rear + 1) % MAXSIZE;
  return OK;
}

//出队
Status DeQueue(SqQueue *Q, QElemType *e) {
//  队列空的判断
  if (Q->front == Q->rear)
    return ERROR;
  *e = Q->data[Q->front];
  Q->front = (Q->front + 1) % MAXSIZE;
  return OK;
}

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


int main() {
```

```
  SqQueue q;
  InitQueue(&q);
  EnQueue(&q, 10);
  EnQueue(&q, 20);
  EnQueue(&q, 30);
  int e;
  DeQueue(&q, &e);
  QueueTraverse(q);
  return 0;
}
```

队列的链式存储结构

```
//队列的链式存储结构
# include <stdio.h>
# include <stdlib.h>

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0

typedef int Status;
typedef int QElemType;

typedef struct QNode {
  QElemType data;
  struct QNode *next;
} QNode, *QueuePtr;

typedef struct {
  QueuePtr front, rear;
} LinkQueue;

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

//入队
Status EnQueue(LinkQueue *Q, QElemType e) {
  QueuePtr s = (QueuePtr)malloc(sizeof(QNode));
  if (!s)
    return ERROR;
  s->data = e;
  s->next = NULL;
```

```
  Q->rear->next = s;
  Q->rear = s;
  return OK;
}

//出队
Status DeQueue(LinkQueue *Q, QElemType *e) {
  QueuePtr p;
  if (Q->front == Q->rear)
    return ERROR;
  p = Q->front->next;
  *e = p->data;
  Q->front->next = p->next;
//  如果队头是队尾
  if (Q->rear == p)
    Q->rear = Q->front;
  free(p);
  return OK;
}

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

int main() {
  LinkQueue q;
  InitQueue(&q);
  EnQueue(&q, 1);
  EnQueue(&q, 2);
  EnQueue(&q, 3);
  int e;
  DeQueue(&q, &e);
  QueueTraverse(q);
  return 0;
}
```

3 串

- 串(string)是由零个或多个字符组成的有限序列，又名叫字符串。
- 零个字符的串成为空串(null string)。
- 空格串：只包含空格的串。
- 子串和主串：串中任意个数的连续字符组成的子序列成为该串的子串，相应的，包含子串的串称为主串。

串的编码

- 标准ASCII编码：由7位二进制数表示一个字符，总共可以表示128个字符。
- 扩展ASCII编码：由位二进制数表示一个字符，总共可以表示256个字符。
- Unicode编码：比较常用的是由16位二进制数表示一个字符。

串的表示和实现

定长顺序存储结构

```
# define MAX_STR_LEN 40
typedef unsigned char SString[MAX_STR_LEN + 1];
```

堆分配存储结构

```
struct HString {
  char *ch;
  int length;
};

void InitString(HString &S) {
  S.length = 0;
  S.ch = NULL;
}

//生成一个其值等于串常量cahrs的串T
void StrAssign(HString &T, char *chars) {
  int i, j;
  if (T.ch)
    free(T.ch);
  i = strlen(chars);
  if (!i)
    InitString(T);
  else {
    T.ch = (char*)malloc(i * sizeof(char));
    if (!T.ch)
      exit(OVERFLOW);
    for (j = 0; j < 1; j++)
      T.ch[j] = chars[j];
    T.length = i;
  }
}
```

模式匹配

暴力匹配

```
//返回子串T在主串S中第pos个字符之后的位置
//字符串第一个位置存放长度
int Index(String s, String T, int pos) {
//  主串S当前的下标，若pos不为1，则从pos位置开始匹配
  int i = pos;
//  子串T的下标
  int j = 1;
  while (i <= S[0] && j <= T[0]) {
    if (S[i] == T[j]) {
      i++;
      j++;
    }
    else {
//      退回到上次匹配首位的下一位
      i = i - j + 2;
      j = 1;
    }
  }
  if (j > T[0])
    return i - T[0];
  else
    return 0;
}
```

KMP

KMP算法仅当模式与主串之间存在许多“部分匹配”的情况下才能体现出它的优势。

求next数组方法：

$next[j] =$

1. 0，当$j = 1$时
2. 最大公共前后缀长度 + 1
3. 1，其他情况

求next数组代码：

```
void get_next(String T, int *next) {
  int i, j;
  i = 1;
  j = 0;
  next[1] = 0;
  while (i < T[0]) {
//    T[i]为后缀的单个字符，T[j]为前缀
    if (j == 0 || T[i] == T[j]) {
```

```
      i++;
      j++;
      next[i] = j;
    }
    else {
//      字符不相等，j回溯
      j = next[j];
    }
  }
}
```

匹配代码：

```
int IndexKMP(String S, String T, int pos) {
  int i = pos;
  int j = 1;
  int next[255];
  get_next(T, next);
  while (i <= S[0] && j <= T[0]) {
    if (j == 0 || S[i] == T[j]) {
      ++i;
      ++j;
    }
    else {
//      j退回合适的位置，i不变
      j = next[j];
    }
  }
  if (j > T[0])
    return i - T[0];
  else
    return 0;
}
```

改进的KMP

求nextval数组方法：

$nextval[j] =$

1. $j = 1$时，0
2. $j > 1$时，
    1. 若$P_j \neq P_{next}[j]$，则$nextval[j] = next[j]$
    2. 若$P_j = P_{next}[j]$，则$nextval[j] = nextval[next[j]]$

求next数组代码：

```
void get_nextval(String T, int *nextval) {
  int i, j;
```

```
  i = 1;
  j = 0;
  nextval[1] = 0;
  while (i < T[0]) {
    if (j == 0 || T[i] == T[j]) {
      ++i;
      ++j;
//      若当前字符与前缀字符不同
      if (T[i] != T[j])
        nextval[i] = j;
      else
        nextval[i] = nextval[j];
    }
    else
      j = nextval[j];
  }
}
```

4 数组

数组是由类型相同的数据元素构成的有序集合，每个元素称为数组元素，每个元素受$n(n \geq 1)$个线性关系的约束，每个元素在n个线性关系中的序号称为该元素的下标，可以通过下标访问该数据元素。

一个二维数组类型可以定义为其分量类型为一维数组类型的一维数组类型，也就是说：

```
typedef Elem Type Array[m][n];
```

等价于

```
typedef ElemType Array1[n];
typedef Array1 Array2[m];
```

数组一旦被定义，它的维数和维界就不再改变。因此，除了结构的初始化和销毁之外，数组只有存取元素和修改元素值的操作。

数组的顺序表示和实现

对二维数组可有两种存储方式：一种是以列序为主序的存储方式，一种是以行序为主序的存储方式。在扩展 Basic 、 Pascal 、 Java 和 C 语言中， 用的都是以行序为主序的存储结构， 而在 FORTRAN 语言中，用的是以列序为主序的存储结构。

对数组数据结构的解释：
(原文链接：https://blog.csdn.net/panglinzhuo/article/details/79397277)

```
*base:

数组元素基址,以二维数组A为例,将数组(按行或者按列)拉成一个向量L所组成的线性结构的首地址.
```

*bounds:

数组维界地址,指向一个一维数组B,它存放了数组A各维度元素的数目.假设数组A是(3,4,5)大小的, 则数组B=[3,4,5]

*constants:

数组映像函数常量基址,指向一个数组C, 它存放了"数组A各个维度上的数字加一时, 元素在线性结构L上所移动的距离".举个栗子吧!

首先二维数组A(3,4):

bounds[] = [3,4];

constants[] = [4,1]:代表第0维上的数字每加一, 元素在线性结构L中的位置就增加了4; 而第一维上的数字每加一, 元素位置也是加一

若要求坐标为(2,2)的元素地址addr.(处在第三行,第三列的位置),则

addr = 数组A首元素地址 + 2*constants[0] + 2*constants[1]

= 数组A首元素地址 + 2*4+2 = 数组A首元素地址 + 10

再看一个三维数组A(3,4,5):

bounds[] = {3,4,5}

constants[] = {4*5, 5, 1}

可以看出:constants[2] = 1; constants[1] = constants[2]*bounds[2]; constants[0] = constants[1]*bounds[1]

即:constants[i] = constans[i+1] * bounds[i+1]

对于一个元素, 其第0维每增加一, 在线性结构L中的位置就增加了4*5 = 20个, 以此类推.

则元素(1,2,3)的元素位置为:

1*constants[0] + 2*constants[1] + 3*constants[2] = 1*20 + 2*5 +3*1 = 33

代码实现:

```
//数组的顺序表示和实现
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <math.h>
```

```
# include <stdarg.h>
# include <iostream>

using namespace std;

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
//# define OVERFLOW -2

typedef int Status;
typedef int Boolean;
typedef int ElemType;

# define MAX_ARRAY_DIM 8

struct Array {
//  数组元素基址
  ElemType *base;
//  数组维数
  int dim;
//  数组维界基址
  int *bounds;
//  数组映像函数常量基址
  int *constants;
};

Status InitArray(Array &A, int dim, ...) {
  int elemtotal = 1;
//  变长参数表
  va_list ap;
  if (dim < 1 || dim > MAX_ARRAY_DIM)
    return ERROR;
  A.dim = dim;
  A.bounds = (int*)malloc(dim * sizeof(int));
  if (!A.bounds)
    exit(OVERFLOW);
  va_start(ap, dim);
  for (int i = 0; i < dim; i++) {
    A.bounds[i] = va_arg(ap, int);
    if (A.bounds[i] < 0)
      return UNDERFLOW;
    elemtotal *= A.bounds[i];
  }
  //  结束提取变长参数
  va_end(ap);
  A.base = (ElemType*)malloc(elemtotal * sizeof(ElemType));
```

```
  if (!A.base)
    exit(OVERFLOW);
  A.constants = (int*)malloc(dim * sizeof(int));
  if (!A.constants)
    exit(OVERFLOW);
//  最后一维的偏移量为1
  A.constants[dim - 1] = 1;
  for (int i = dim - 2; i >= 0; i--)
    A.constants[i] = A.bounds[i + 1] * A.constants[i + 1];
  return OK;
}

void DestoryArray(Array &A) {
  if (A.base)
    free(A.base);
  if (A.bounds)
    free(A.bounds);
  if (A.constants)
    free(A.constants);
  A.base = A.bounds = A.constants = NULL;
  A.dim = 0;
}

//Value(), Assign()调用此函数
//若ap指示的各下标值合法，则求出该元素在A中的相对地址off
Status Locate(Array &A, va_list ap, int &off) {
  int i, ind;
  off = 0;
  for (i = 0; i < A.dim; i++) {
    ind = va_arg(ap, int);
    if (ind < 0 || ind >= A.bounds[i])
      return OVERFLOW;
    off += A.constants[i] * ind;
  }
  return OK;
}

//"..."依次为各维的下标值，若各下标合法，则e被赋值为A的相应的元素值
Status Value(ElemType &e, Array A, ...) {
  va_list ap;
  int off;
//  变长参数从形参A之后开始
  va_start(ap, A);
  if (Locate(A, ap, off) == OVERFLOW)
    return ERROR;
  e = *(A.base + off);
  return OK;
}
```

```
//"..."依次为各维的下标值，若各下标合法，则e的值赋给A的指定的元素
Status Assign(Array A, ElemType e, ...) {
  va_list ap;
  int off;
  va_start(ap, e);
  if (Locate(A, ap, off) == OVERFLOW)
    return ERROR;
  *(A.base + off) = e;
  return OK;
}

int main() {
  Array A;
//  构造A[3][4][2]
  int i, j, k, dim = 3, bound1 = 3, bound2 = 4, bound3 = 2;
  ElemType e;
  InitArray(A, dim, bound1, bound2, bound3);
  cout << "A.bounds = ";
  for (i = 0; i < dim; i++)
    cout << *(A.bounds + i) << " ";
  cout << endl << "A.constants = ";
  for (i = 0; i < dim; i++)
    cout << *(A.constants + i) << " ";

  cout << endl << "Data follows:" << endl;
  for (i = 0; i < bound1; i++) {
    for (j = 0; j < bound2; j++) {
      for (k = 0; k < bound3; k++) {
        Assign(A, i * 100 + j * 10 + k, i, j, k);
        Value(e, A, i, j, k);
        printf("A[%d][%d][%d] = %2d ", i, j, k, e);
      }
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

十字链表

用十字链表存储稀疏矩阵，该存储方式采用的是 "链表+数组" 结构。
使用十字链表压缩存储稀疏矩阵时，矩阵中的各行各列都各用一各链表存储，与此同时，所有行链表的表头存储到一个数组(rhead)，所有列链表的表头存储到另一个数组(chead)中。

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

```c
# include <stdlib.h>

typedef struct OLNode {
  int i, j, e; //矩阵三元组i代表行 j代表列 e代表当前位置的数据
  struct OLNode *right, *down; //指针域 右指针 下指针
} OLNode, *OLink;

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

CrossList CreateMatrix_OL(CrossList M);
void display(CrossList M);
int main() {
  CrossList M;
  M.rhead = NULL;
  M.chead = NULL;
  M = CreateMatrix_OL(M);
  printf("输出矩阵M:\n");
  display(M);
  return 0;
}

CrossList CreateMatrix_OL(CrossList M) {
  int m, n, t;
  int i, j, e;
  OLNode *p, *q;
  printf("输入矩阵的行数、列数和非0元素个数: ");
  scanf("%d%d%d", &m, &n, &t);
  M.mu = m;
  M.nu = n;
  M.tu = t;
  if (!(M.rhead = (OLink*)malloc((m + 1) * sizeof(OLink))) || !(M.chead =
(OLink*)malloc((n + 1) * sizeof(OLink)))) {
    printf("初始化矩阵失败");
    exit(0);
  }
  for (i = 1; i <= m; i++) {
    M.rhead[i] = NULL;
  }
  for (j = 1; j <= n; j++) {
    M.chead[j] = NULL;
  }
  for (scanf("%d%d%d", &i, &j, &e); 0 != i; scanf("%d%d%d", &i, &j, &e)) {
    if (!(p = (OLNode*)malloc(sizeof(OLNode)))) {
      printf("初始化三元组失败");
      exit(0);
    }
```

```
    p->i = i;
    p->j = j;
    p->e = e;
    //链接到行的指定位置
    if (NULL == M.rhead[i] || M.rhead[i]->j > j) {
      p->right = M.rhead[i];
      M.rhead[i] = p;
    } else {
      for (q = M.rhead[i]; (q->right) && q->right->j < j; q = q->right);
      p->right = q->right;
      q->right = p;
    }
    //链接到列的指定位置
    if (NULL == M.chead[j] || M.chead[j]->i > i) {
      p->down = M.chead[j];
      M.chead[j] = p;
    } else {
      for (q = M.chead[j]; (q->down) && q->down->i < i; q = q->down);
      p->down = q->down;
      q->down = p;
    }
  }
  return M;
}

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

特殊矩阵的压缩

特殊矩阵

对称矩阵，对角矩阵的存储。

稀疏矩阵的三元组顺序表存储结构

组中数据分别表示(行标，列标，元素值)。

```
# define MAX_SIZE 100
```

```
struct Triple {
//  行下标，列下标
  int i, j;
//  非零元素值
  ElemType e;
};

struct TSMatrix {
  Triple data[MAX_SIZE + 1];
//  矩阵的行数，列数，非零元个数
  int mu, nu, tu;
};
```

广义表

- 广义表中的数据元素可以具有不同的结构：原子或列表。

5.5 广义表的存储结构

由于广义表$(\alpha_1,\alpha_2,\cdots,\alpha_n)$中的数据元素可以具有不同的结构(或是原子,或是列表),因此难以用顺序存储结构表示,通常采用链式存储结构,每个数据元素可用一个结点表示。

如何设定结点的结构?由于列表中的数据元素可能为原子或列表,由此需要两种结构的结点:一种是表结点,用以表示列表;一种是原子结点,用以表示原子。从上节得知:若列表不空,则可分解成表头和表尾;反之,一对确定的表头和表尾可惟一确定列表。由此,一个表结点可由 3 个域组成:标志域、指示表头的指针域和指示表尾的指针域;而原子结点只需两个域:标志域和值域(如图 5.8 所示)。其形式定义说明如下:

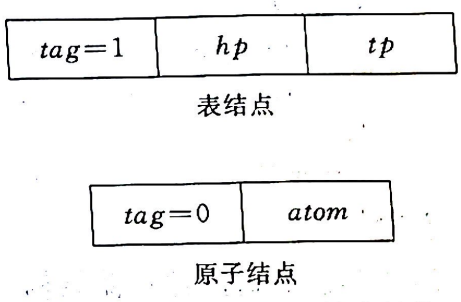


图 5.8 列表的链表结点结构

```
// ------ 广义表的头尾链表存储表示 ------
typedef enum {ATOM, LIST }ElemTag; // ATOM == 0:原子,LIST == 1:子表
typedef struct GLNode {
    ElemTag  tag;              // 公共部分,用于区分原子结点和表结点
    union {                    // 原子结点和表结点的联合部分
      AtomType  atom;          // atom 是原子结点的值域,AtomType 由用户定义
      struct {struct GLNode   * hp, * tp;}ptr;
                               // ptr 是表结点的指针域,ptr.hp 和 ptr.tp 分别指向表头和表尾
  };
} * GList;                     // 广义表类型
```

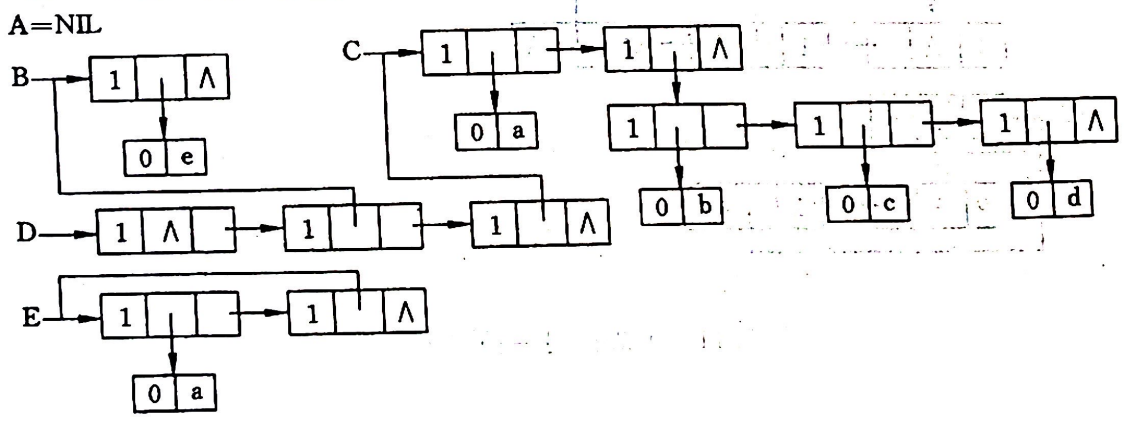


图 5.9 广义表的存储结构示例

广义表的长度

广义表的长度指的是广义表中数据元素的数量。这里需要指明的是，一个广义表中，一个原子算做是一个元素，一个子表也只算做一个元素。

空表的长度为 0，只含有一个原子的广义表长度为 1。

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

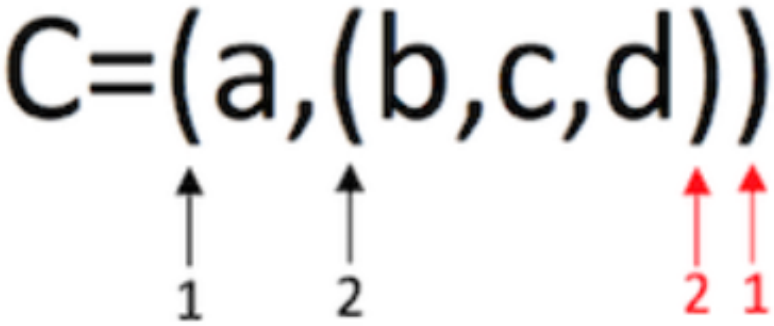
在广义表 {{a,b,c}} 中只有一个子表 {a,b,c}，因此它的长度为 1。

在 $LS = (a_1, a_2, \ldots, a_n)$ 中，a_i表示原子或者子表， LS 的长度为 n。

广义表的深度

广义表的深度，指的是广义表中括号的重数。

例如：$C = (a, (b, c, d))$:



图中，从前往后数左括号的数量就是广义表C的深度，为2；也可以从右往左数右括号的数量(红色)。

5 树

基本术语

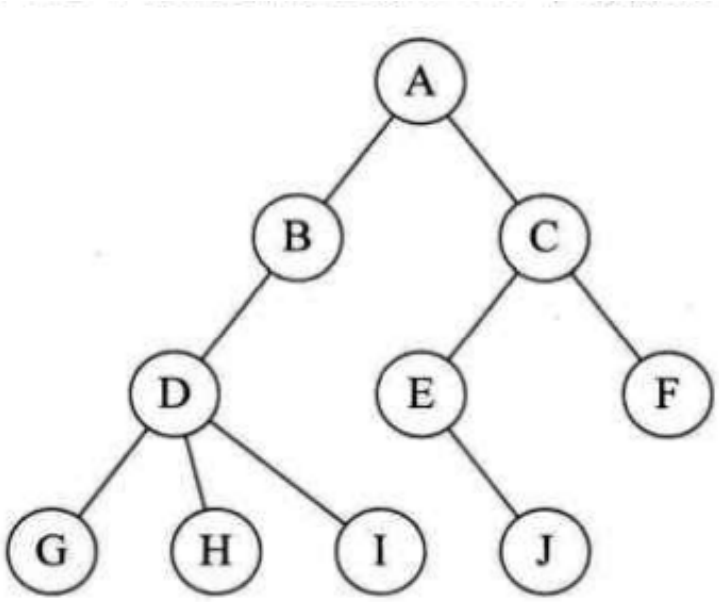
树结构是一种非线性存储结构，存储的是具有“一对多”关系的数据元素的集合。

- 结点拥有的子树数称为结点的**度**(Degree)。
- 度为0的结点称为**叶子**(Leaf)或**终端结点**。
- 度不为0的结点称为**非终端结点**或**分支结点**。除根结点外，分支结点也称为**内部结点**。
- **树的度**是树内各结点的度的最大值。
- 结点的子树的根称为该结点的**孩子**(Child)，相应的，该结点称为孩子的**双亲**(Parent)。
- 统一双亲的孩子之间互称**兄弟**(Sibling)。
- 以某结点为根的子树中的任一结点都称为该结点的**子孙**。

- 结点的**层次**(Level)从根开始定义起，根为第一层。
- 双亲在同一层的结点互称为**堂兄弟**。
- 树中结点的最大层次称为树的**深度**(Depth)或高度。
- 如果将树中结点的各子树看成从左至右是有次序的(即不能互换)，则称该树为**有序树**，否则称为**无序树**。

- **森林**是$m(m \geq 0)$棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。

树的存储结构



顺序存储

双亲表示法

```
# define MAX_TREE_SIZE 100
typedef int TElemType;

//结点结构
typedef struct PTNode {
  TElemType data;
  int parent;
} PTNode;

//树结构
typedef struct {
  PTNode nodes[MAX_TREE_SIZE];
  int r, n;
} PTree;
```

根结点的位置域设置为-1。
此时，很容易找到它的双亲结点，时间复杂度为$O(1)$，但如果找结点的孩子，需要遍历整个结构。

孩子表示法

把每个结点的孩子结点排列起来，以单链表作为存储结构，则n个结点有n个孩子链表，如果是叶子结点则此单链表为空，然后n个头指针又组成一个线性表，采用顺序存储结构，放进一个一维数组中。

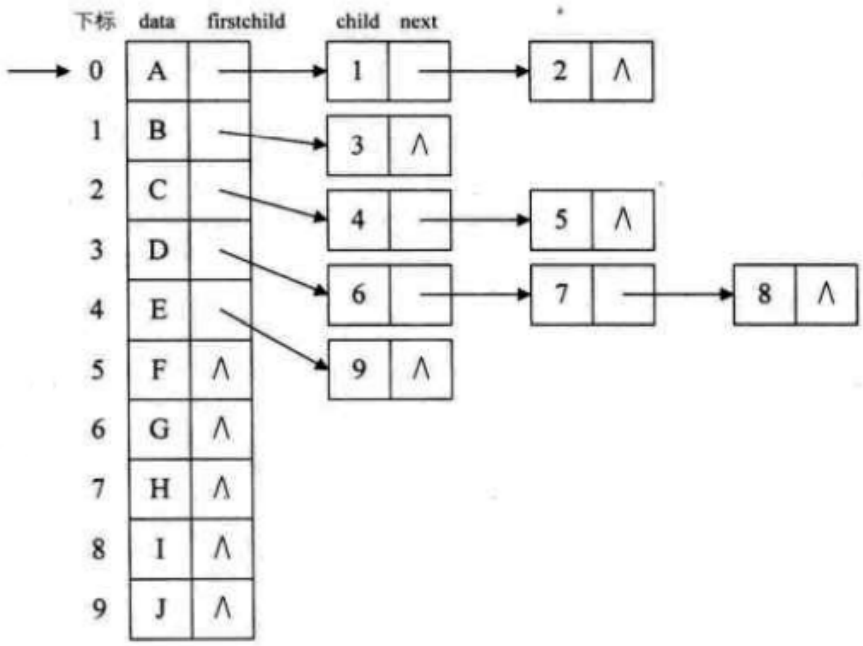


图 6-4-4

为此，设计两种结点结构，一个是孩子链表的孩子结点，如表 6-4-7 所示。

表 6-4-7

child	next

其中 child 是数据域，用来存储某个结点在表头数组中的下标。next 是指针域，用来存储指向某结点的下一个孩子结点的指针。

另一个是表头数组的表头结点，如表 6-4-8 所示。

表 6-4-8

data	firstchild

其中 data 是数据域，存储某结点的数据信息。firstchild 是头指针域，存储该结点的孩子链表的头指针。

```
# define MAX_TREE_SIZE 100
typedef int TElemType;

//孩子结点
typedef struct CTNode {
  int child;
  struct CTNode *next;

} *ChildPtr;

//表头结点
typedef struct {
  TElemType data;
```

```
  ChildPtr firstchild;
} CTBox;

//树结构
typedef struct {
//  结点数组
  CTBox nodes[MAX_TREE_SIZE];
//  根的位置和结点数
  int r, n;
} CTree;
```

此时，查找某个结点的某个孩子，找某个结点的兄弟，只需要查找这个结点的孩子单链表即可，但如果查找某个结点的双亲要遍历整棵树。改进后，将双亲表示法和孩子表示法结合为双亲孩子表示法。

链式存储

孩子兄弟表示法

```
typedef int TElemType;

typedef struct CSNode {
  TElemType data;
  struct CSNode *firstchild, *rightsib;
} CSNode, *CSTree;
```

最大的好处是它把一棵复杂的树变成了一棵二叉树。

二叉树

二叉树的定义

- 每个结点至多只有两棵子树(即二叉树中不存在度大于2的结点)。
- 二叉树的子树有左右之分，其次序不能任意颠倒。

特殊形态的二叉树

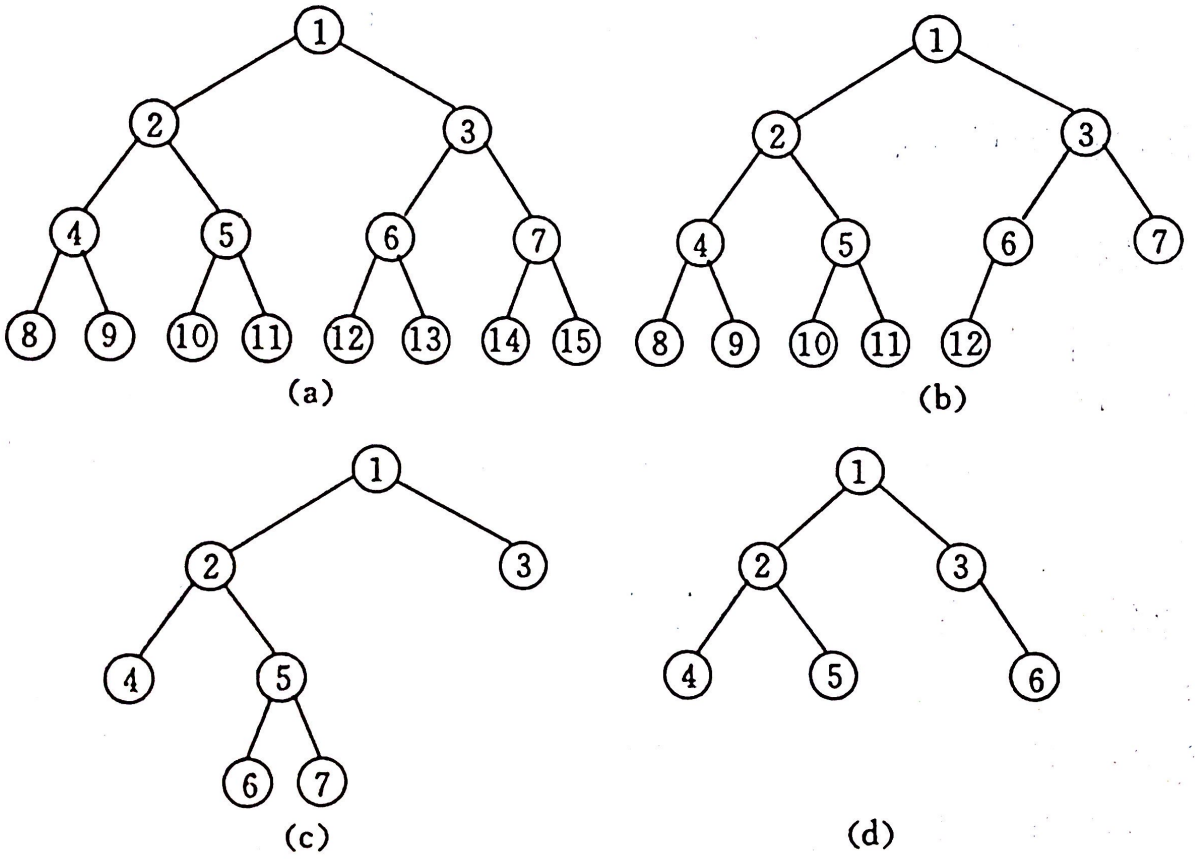


图 6.4　特殊形态的二叉树

(a) 满二叉树；(b) 完全二叉树；(c)和(d)非完全二叉树

满二叉树

一棵深度为k且有2^k-1个结点的二叉树称为**满二叉树**。

完全二叉树

深度为k的，有n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称为**完全二叉树**。

特点：

1. 叶子结点只可能在层次最大的两层上出现。
2. 对任一结点，若其右分支下的子孙的最大层次为l，则其左分支下的子孙的最大层次数必为l或$l+1$。

性质：

1. 具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$。($\lfloor x \rfloor$表示不大于$x$的最大整数)
2. 如果对一棵有n个结点的完全二叉树(其深度为$\lfloor \log_2 n \rfloor + 1$)的结点按层序编号(从第1层到第$\lfloor \log_2 n \rfloor + 1$层，每层从左到右)，则对任一结点$i(1 \le i \le n)$，有
    1. 如果$i=1$，则结点i是二叉树的根，无双亲；如果$i>1$，则其双亲PARENT(i)是结点$\lfloor i/2 \rfloor$。
    2. 如果$2i>n$，则结点i无左孩子(结点i为叶子结点)；否则其左孩子LCHILD(i)是结点$2i$。
    3. 如果$2i+1>n$，则结点i无右孩子；否则其右孩子RCHILD(i)是结点$2i+1$。

二叉树的性质

1. 在二叉树的第i层上至多有2^{i-1}个结点($i \geq 1$)。
2. 深度为k的二叉树至多有$2^k - 1$个结点($k \geq 1$)。
3. 对任意一棵二叉树T，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0 = n_2 + 1$。

二叉树的存储结构

顺序存储

二叉树的顺序存储就是用数组存储二叉树。二叉树的每个结点在顺序存储中都有自己的固定位置。

如果设置根结点存储在[0]，则每个结点在数组中的序号如图 6-1 所示。第 i 层结点的序号从 $2^{i-1}-1$ 到 2^i-2；序号为 j 的结点，其所处的层 i 是 $\lfloor \log_2(j+1) \rfloor + 1$；其在层 i 中的排序 $k=j+2-2^{i-1}$；其双亲序号为 $\lfloor (j-1)/2 \rfloor$；其左右孩子序号分别为 $2j+1$ 和 $2j+2$。

有些算法(如第 9 章的堆排序)设置根结点存储在[1]([0]用做临时存储单元)，则每个结点在数组中的序号如图 6-2 所示。第 i 层结点的序号从 2^{i-1} 到 2^i-1；序号为 j 的结点，其所处的层 i 是 $\lfloor \log_2 j \rfloor + 1$；其在层 i 中的排序 $k=j+1-2^{i-1}$；其双亲序号为 $\lfloor j/2 \rfloor$；其左右孩子序号分别为 $2j$ 和 $2j+1$。

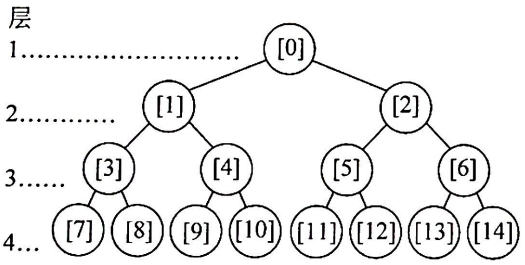


图 6-1　根结点在[0]的二叉树结点序号

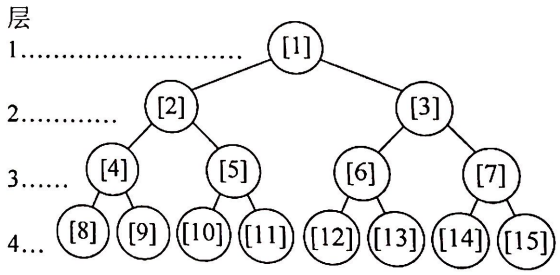


图 6-2　根结点在[1]的二叉树结点序号

i 层的满二叉树，其结点总数为 2^i-1。二叉树每增加一层，结点数就要加倍。所以二叉树的顺序存储结构适合存储完全二叉树。第 9 章的堆排序以及第 10 章的多路平衡归并和置换-选择排序都要建立完全二叉树，故采用顺序存储结构。如果二叉树的形态和完全二叉树的差别很大，采用顺序存储结构就很浪费存储空间。这种情况可以采用链式存储结构。

```
# define MAX_TREE_SIZE 100

typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree bt;
```

例如使用0表示不存在此结点，仅适用于完全二叉树。

链式存储

```
typedef int TElemType;

typedef struct BiTNode {
  TElemType data;
  BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

二叉树的遍历方式

遍历方式：

- 先(根)序遍历，中序遍历，后序遍历。
- 层次遍历。

表达式：

- 先序遍历为前缀表示(波兰式)。
- 中缀表示。
- 后序遍历为后缀表示(逆波兰式)。

二叉树的建立和遍历

```
//二叉链表
# include <stdio.h>
# include <stdlib.h>
# include <iostream>
# include <queue>

using namespace std;

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
# define OVERFLOW -2

typedef int Status;
typedef int Boolean;
typedef int TElemType;

typedef struct BiTNode {
  TElemType data;
  BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

# define Nil NULL
# define ClearBiTree DestoryBiTree
```

```
void InitBiTree(BiTree &T) {
  T = NULL;
}

void DestoryBiTree(BiTree &T) {
  if (T) {
    DestoryBiTree(T->lchild);
    DestoryBiTree(T->rchild);
    free(T);
    T = NULL;
  }
}

void PreOrderTraverse(BiTree T, void(*Visit)(TElemType)) {
  if (T) {
    Visit(T->data);
    PreOrderTraverse(T->lchild, Visit);
    PreOrderTraverse(T->rchild, Visit);
  }
}

void InOrderTraverse(BiTree T, void(*Visit)(TElemType)) {
  if (T) {
    InOrderTraverse(T->lchild, Visit);
    Visit(T->data);
    InOrderTraverse(T->rchild, Visit);
  }
}

void PostOrderTraverse(BiTree T, void(*Visit)(TElemType)) {
  if (T) {
    PostOrderTraverse(T->lchild, Visit);
    PostOrderTraverse(T->rchild, Visit);
    Visit(T->data);
  }
}

void LevelOrderTraverse(BiTree T, void(*Visit)(TElemType)) {
  queue<BiTree> q;
  BiTree a;
  if (T) {
    q.push(T);
    while (!q.empty()) {
      a = q.front();
      q.pop();
      Visit(a->data);
      if (a->lchild != NULL)
        q.push(a->lchild);
```

```
      if (a->rchild != NULL)
        q.push(a->rchild);
    }
    cout << endl;
  }
}


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

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

TElemType Root(BiTree T) {
  if (BiTreeEmpty(T))
//    表示无值
    return Nil;
  else
    return T->data;
}

TElemType Value(BiTree p) {
  return p->data;
}

void Assign(BiTree p, TElemType value) {
  p->data = value;
}

void CreateBiTree(BiTree &T) {
  TElemType ch;
  scanf("%c", &ch);
  if (ch == '#')
    T = NULL;
  else {
    T = (BiTree)malloc(sizeof(BiTNode));
    if (!T)
      exit(OVERFLOW);
```

```
    T->data = ch;
    CreateBiTree(T->lchild);
    CreateBiTree(T->rchild);
  }
}

void Visit(TElemType e) {
  printf("%c ", e);
}

int main() {
  BiTree T;
  InitBiTree(T);
  CreateBiTree(T);
  cout << endl << "PreOrder: " << endl;
  PreOrderTraverse(T, Visit);
  cout << endl << "InOrder: " << endl;
  InOrderTraverse(T, Visit);
  cout << endl << "PostOrder: " << endl;
  PostOrderTraverse(T, Visit);
  cout << endl << "LevelOrder: " << endl;
  LevelOrderTraverse(T, Visit);
  return 0;
}
```

线索二叉树

线索二叉树的定义

当以二叉链表作为存储结构时，只能找到结点的左、右孩子信息，而不能直接得到结点在任一序列中的前驱和后继信息，这种信息只有在遍历的动态过程中才能得到。

故做如下改进：

1. 若结点有左子树，则其`lchild`域指示其左孩子，否则令`lchild`域指示其前驱。
2. 若结点有右子树，则其`rchild`域指示其右孩子，否则令`rchild`域指示其后继。
3. 为了避免混淆，增加两个标志位`LTag`和`RTag`。

以上述结构构成的二叉链表作为树的存储结构，叫做**线索链表**，其中指向结点前驱和后继的指针叫做**线索**。加上线索的二叉树称为**线索二叉树**(Threaded Binary Tree)。

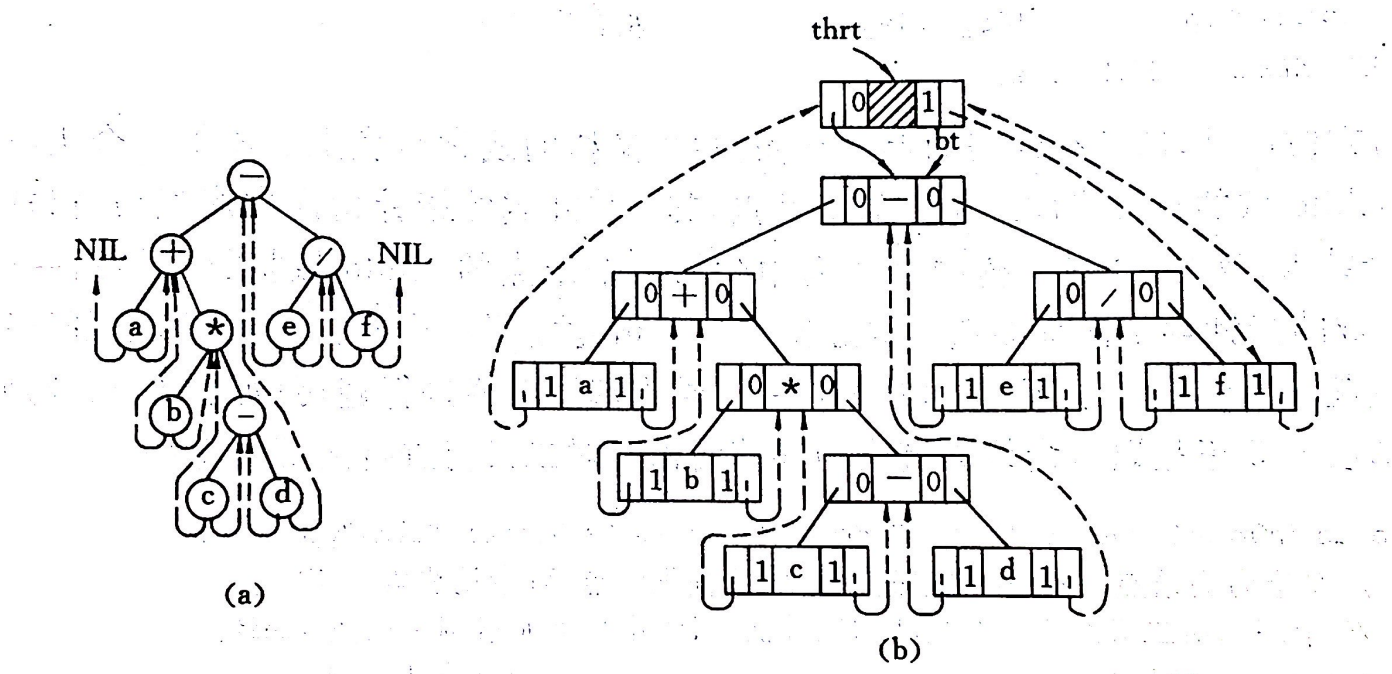


图 6.11　线索二叉树及其存储结构

(a) 中序线索二叉树；(b) 中序线索链表

- 找前驱：在中序遍历中，结点的后继应是遍历其右子树时访问的第一个结点，即右子树中最左下的结点。
- 找后继：若其左标志位为1，则左链为线索，指示其前驱，否则遍历左子树时最后访问的一个结点(左子树中最右下的结点)为其前驱。

二叉线索存储表示

```
typedef int TElemType;

typedef enum PointerTag {
  Link, Thread
};

typedef struct BiThrNode {
  TElemType data;
  struct BiThrNode *lchild, *rchild;
  PointerTag LTag, RTag;
} BiThrNode, *BiThrTree;
```

仿照线性表的存储结构，在二叉树的线索链表上也添加一个头结点，并令其`lchild`域的指针指向二叉树的根结点，其`rchild`域的指针指向中序遍历时访问的最后一个结点。

相当于为二叉树建立了一个双向线索链表，既可以从第一个结点起顺后继进行遍历，也可以从最后一个结点起顺前序进行遍历。

线索二叉树的遍历

```
Status InOrderTraverse_Thr(BiThrTree T, void(*Visit)(TElemType)) {
  BiThrTree p;
//  根结点
  p = T->lchild;
  while (p != T) {
    while (p->LTag == Link)
      p = p->lchild;
    Visit(p->data);
    while (p->RTag == Thread && p->rchild != T) {
      p = p->rchild;
      Visit(p->data);
    }
    p = p->rchild;
  }
  return OK;
}
```

二叉树线索化

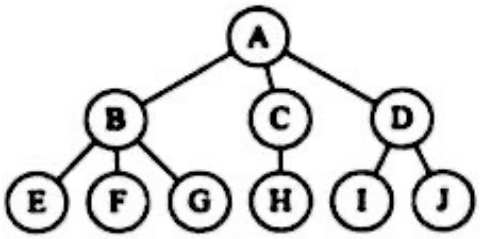
线索化即修改空指针的过程。

```
//始终指向刚访问过的结点
BiThrTree pre;
void InThreading(BiThrTree p) {
  if (p) {
    InThreading(p->lchild);
//    没有左孩子
    if (!p->lchild) {
      p->LTag = Thread;
      p->lchild = pre;
    }
    if (!p->rchild) {
      pre->RTag = Thread;
      pre->rchild = p;
    }
    pre = p;
    InThreading(p->rchild);
  }
}
```

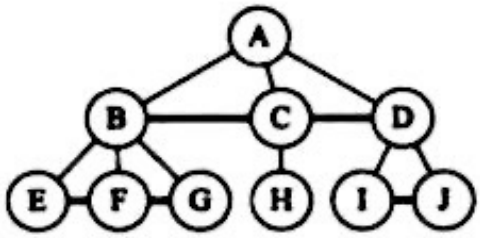
树、森林、二叉树的转换

数转为二叉树

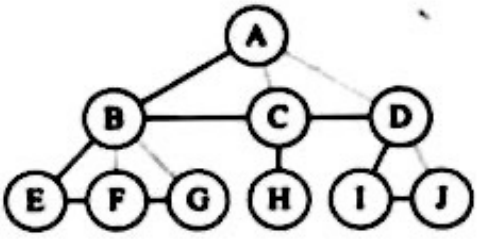
1. 加线。在所有兄弟结点之间加一条连线。
2. 去线。树中的每个结点，只保留它与第一个孩子结点的连线，删除它与其它孩子结点之间的连线。
3. 层次调整。以树的根节点为轴心，将整棵树顺时针旋转一定角度，使之结构层次分明。注意第一个孩子是结点的左孩子，兄弟转换过来的孩子是结点的右孩子。



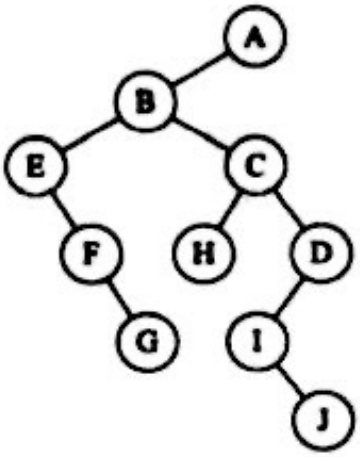
树



步骤1：给兄弟加线



步骤2：给除长子外的孩子去线



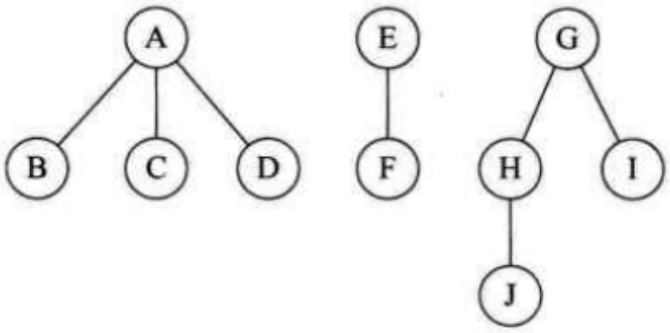
步骤3：层次调整

口诀：兄弟相连，长兄为父，孩子靠左。

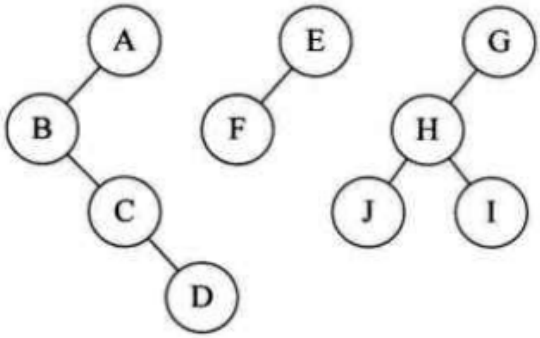
核心：左孩子，右兄弟。

森林转为二叉树

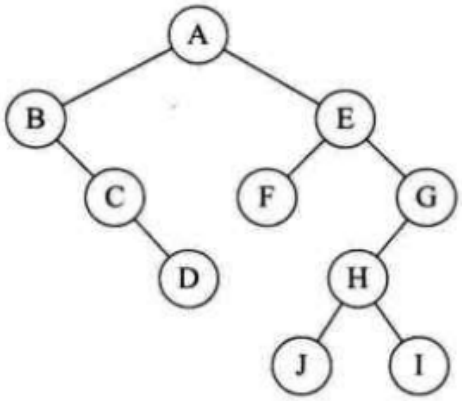
1. 把每棵树转换为二叉树。
2. 第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根节点的右孩子。



拥有三棵树的森林



步骤1:森林中每棵树转换为二叉树

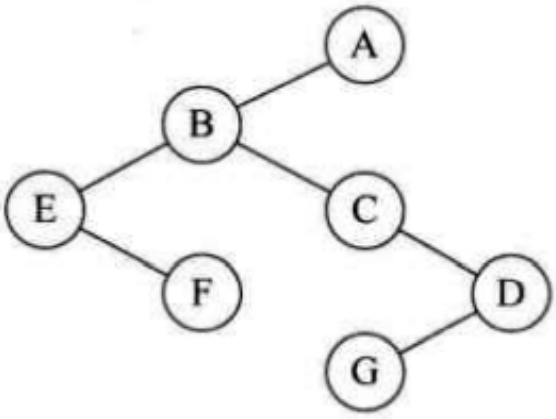


步骤2：将所有二叉树转换为一棵二叉树

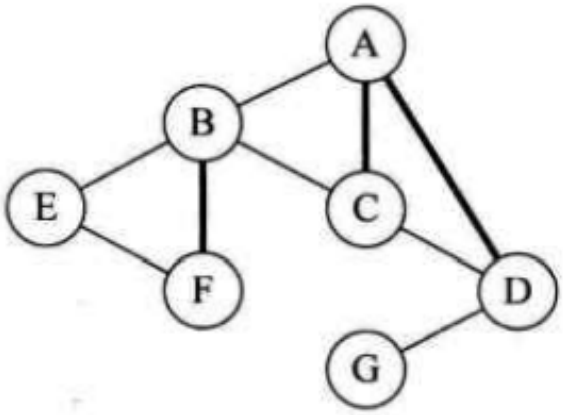
二叉树转为树

是树转换为二叉树的逆过程。还原结点A的孩子，结点A的左孩子开始，一直向右走，这些结点就是结点A的孩子，遇见顺序就是它们作为结点A孩子的顺序。

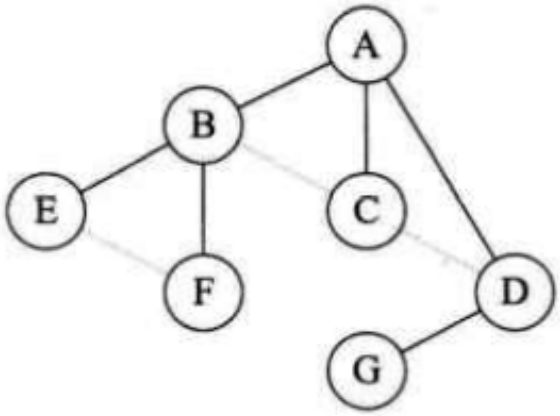
1. 加线。若某结点X的左孩子结点存在，则将这个左孩子的右孩子结点、右孩子的右孩子结点、右孩子的右孩子的右孩子结点...，都作为结点X的孩子。将结点X与这些右孩子结点用线连接起来。
2. 去线。删除原二叉树中所有结点与其右孩子结点的连线。
3. 层次调整。



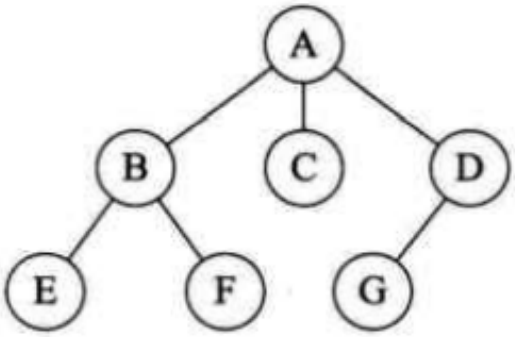
二叉树



步骤1：加线



步骤2：去线

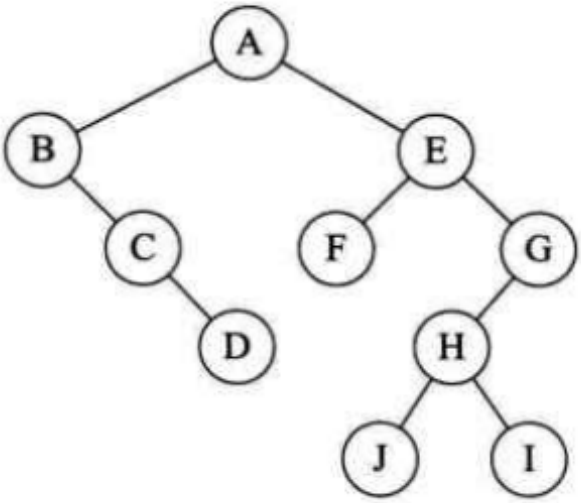


步骤3：层次调整

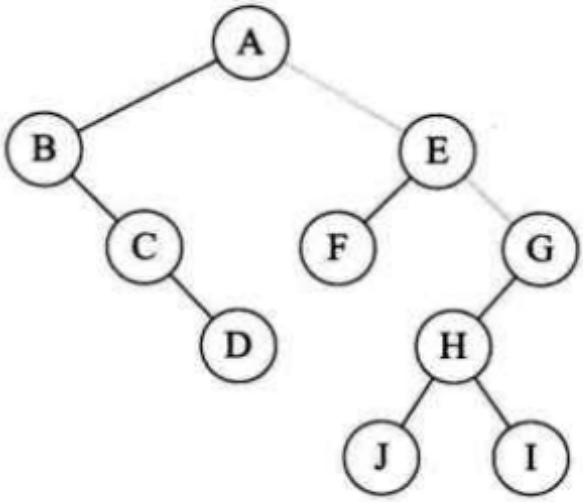
二叉树转为森林

假如一棵二叉树的根结点有右孩子，则这棵二叉树能够转换为森林，否则将转换为一棵树。在二叉树种A有右子树上向右的一连串结点都是A的兄弟，那么就把兄弟分离，A的每个兄弟结点作为森林中树的根结点。

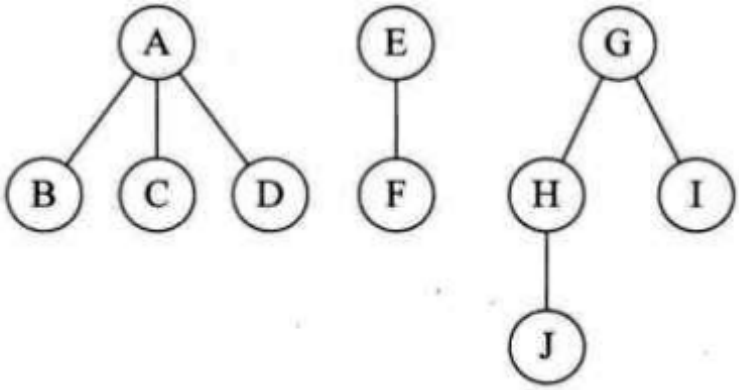
1. 从根结点开始，若右孩子存在，则把与右孩子结点的连线删除。再查看分离后的二叉树，若其根结点的右孩子存在，则连线删除...。直到所有这些根结点与右孩子的连线都删除为止。
2. 将每棵分离后的二叉树转换为树。



二叉树



步骤1：寻找右孩子去线



步骤2：将分离的二叉树转换成树

哈夫曼树

最优二叉树

- 路径：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数目称做**路径长度**。
- 树的路径长度是从树根到每一个结点的路径长度之和。

树的带权路径长度：$WPL = \sum_{k=1}^{n} w_k l_k$

例如，图 6.22 中的 3 棵二叉树，都有 4 个叶子结点 a、b、c、d，分别带权 7、5、2、4，它们的带权路径长度分别为

(a) $WPL=7\times2+5\times2+2\times2+4\times2=36$

(b) $WPL=7\times3+5\times3+2\times1+4\times2=46$

(c) $WPL=7\times1+5\times2+2\times3+4\times3=35$

其中以(c)树的为最小。可以验证，它恰为赫夫曼树，即其带权路径长度在所有带权为 7、5、2、4 的 4 个叶子结点的二叉树中居最小。

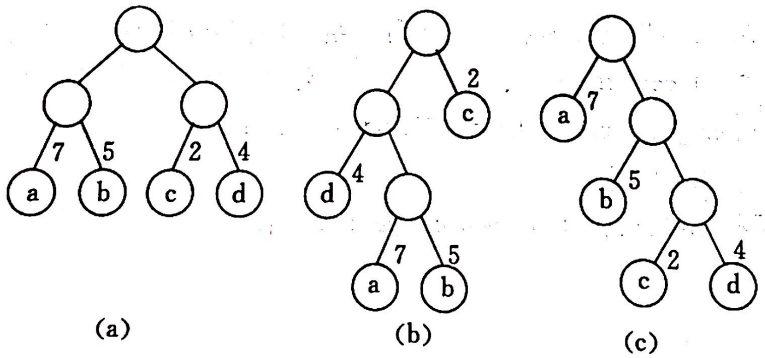


图 6.22　具有不同带权路径长度的二叉树

哈夫曼算法

1. 将每个带有权值的结点作为一棵仅有根结点的二叉树，树的权值为结点的权值。
2. 将其中两棵权值最小的树组成一棵新的二叉树，新树的权值为结点的权值之和。
3. 重复步骤2，直到所有结点都在一棵二叉树上，这棵二叉树就是最优二叉树。

哈夫曼编码

为了编码尽可能的短，故设计长短不等的编码，则必须是任一字符的编码都不是另一个字符的编码的前缀，这种编码称做**前缀编码**。

```
//哈夫曼编码
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <iostream>

using namespace std;

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
# define OVERFLOW -2

typedef int Status;
typedef int Boolean;
```

```
typedef int TElemType;

//静态三叉链表二叉树结构
//双亲值为0的是根结点
//左右孩子值均为0的是叶子结点
typedef struct {
//  结点的权值
  unsigned int weight;
  unsigned int parent, lchild, rchild;
} HTNode, *HuffmanTree;

//动态分配数组存储哈夫曼编码表
typedef char **HuffmanCode;

# define Order

//返回哈夫曼树t的前i个结点中权值最小的树的根结点序号
//select()调用
int min(HuffmanTree t, int i) {
  int j, m;
//  k存最小权值，初值取为不小于可能的值(无符号整型最大值)
  unsigned int k = UINT_MAX;
//  前i个结点
  for (j = 1; j <= i; j++)
//    权值小于k，又是树的根结点
    if (t[j].weight < k && t[j].parent == 0) {
      k = t[j].weight;
      m = j;
    }
//  给选中的根结点的双亲赋非0值，避免第二次查找该结点
  t[m].parent = 1;
  return m;
}

//在前i个结点中选择两个权值最小的树的根结点序号，s1是其中权值较小的
void select(HuffmanTree t, int i, int &s1, int &s2) {
# ifdef Order
  int j;
# endif

  s1 = min(t, i);
  s2 = min(t, i);

# ifdef Order
  if (s1 > s2) {
    j = s1;
    s1 = s2;
    s2 = j;
  }
```

```
# endif
}

void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n) {
  int start;
  unsigned f;

  int m, i, s1, s2;
  unsigned c;
  HuffmanTree p;
  char *cd;
  if (n <= 1)
    return;
//  n个叶子结点的哈夫曼树共有m个结点
  m = 2 * n - 1;
//  0号单元未使用
  HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode));
//  从1号单元开始到n号单元，给叶子结点赋值
  for (p = HT + 1, i = 1; i <= n; i++, p++, w++) {
    p->weight = *w;
    p->parent = 0;
    p->lchild = 0;
    p->rchild = 0;
  }
//  i从i+1到m
  for (; i <= m; i++, p++)
    p->parent = 0;
//  建立哈夫曼树
  for (i = n + 1; i <= m; i++) {
//    在HT[1~i-1]中选择parent为0且weight最小的两个结点
    select(HT, i - 1, s1, s2);
    HT[s1].parent = HT[s2].parent = i;
    HT[i].lchild = s1;
    HT[i].rchild = s2;
    HT[i].weight = HT[s1].weight + HT[s2].weight;
  }

//  叶子到根逆向求每个字符的哈夫曼编码
  HC = (HuffmanCode)malloc((n + 1) * sizeof(char*));
//  分配n个字符编码的头指针向量
  cd = (char*)malloc(n * sizeof(char));
  cd[n - 1] = '\0';
  for (i = 1; i <= n; i++) {
//    逐个字符求哈夫曼编码
    start = n - 1;  // 编码结束位置
//    叶子从根逆向求编码
    for (c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {
      if (HT[f].lchild == c)  // c是双亲的左孩子
        cd[--start] = '0';
```

```
      else  // c是右孩子
        cd[--start] = '1';
    }
//    为第i个字符编码分配空间
    HC[i] = (char*)malloc((n - start) * sizeof(char));
    strcpy(HC[i], &cd[start]);  // 从cd复制到HC
  }
  free(cd);
}

int main() {
  HuffmanTree HT;
  HuffmanCode HC;
  int *w, n, i;
  cout << "Input number of weight(>1): ";
  cin >> n;
//  存放n个权值
  w = (int*)malloc(n * sizeof(int));
  cout << "Input weight(int): " << endl;
  for (i = 0; i < n; i++)
    cin >> *(w + i);
  HuffmanCoding(HT, HC, w, n);
  for (i = 1; i <= n; i++)
    cout << HC[i] << " ";
  return 0;
}
```

6 图

基本术语

数据对象 V：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系 R：

R = {VR}

VR = {<v,w> | v,w∈V 且 P(v,w)，<v,w>表示从 v 到 w 的弧，谓词 P(v,w)定义了弧<v,w>的意义或信息 }

- 在图中的数据元素通常称做**顶点**(Vertex)，V是顶点的有穷非空集合，VR是两个顶点之间的关系的集合。
- 若$< v,w >\in VR$，则$< v,w >$表示从v到w的一条**弧**(Arc)，且v称为**弧尾**(Tail)或初始点(Initial node)，称w为**弧头**(Head)或终端点(Terminal node)，此时的图称为**有向图**(Digragh)。
- 若$< v,w >\in VR$必有$< w,v >\in VR$，即VR是对称的，则以无序对(v,w)代替这两个有序对，表示v和w之间的一条**边**(Edge)，此时的图称为**无向图**(Undigraph)。

用n表示图中顶点的数目，用e表示边或弧的数目，不考虑顶点到其自身的弧或边：

- 对于无向图，e的取值范围是0到$\frac{1}{2}n(n-1)$。
- 有$\frac{1}{2}n(n-1)$条边的无向图称为**完全图**(Completed graph)。
- 有很少条边或弧(如$e < n\log n$)的图称为**稀疏图**(Sparse graph)，反之称为**稠密图**(Dense graph)。
- 与图的边或弧相关的数叫做**权**(Weight)。这些权可以表示从一个顶点到另一个顶点的距离或耗费。这种带权的图通常称为**网**(Netword)。
- **子图**(Subgraph)：子图G'中所有的顶点和边均包含于原图G。

- 对于无向图$G=(V,\{E\})$，如果边$(v,v')\in E$，则称顶点v和v'互为**邻接点**(Adjacent)。边(v,v')**依附**(Incident)于顶点v和v'，或者说(v,v')和顶点v和v'**相关联**。
- 顶点v的**度**(Degree)是和v相关联的边的数目，记为$TD(V)$。
- 对于有向图$G=(V,\{A\})$，如果弧$<v,v'>\in A$，则称顶点v**邻接到**顶点v'，顶点v'**邻接自**顶点v。弧$<v,v'>$和顶点v，v'相关联。

- 以顶点v为头的弧的数目称为v的**入度**(InDegree)，记作$ID(v)$。
- 以顶点v为尾的弧的数目称为v的**出度**(OutDegree)，记作$OD(v)$。
- 顶点v的度为$TD(v)=ID(v)+OD(v)$。
- 一般地，如果顶点v_i的度记为$TD(v_i)$，那么一个有n个顶点，e条边或弧的图，满足$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$。

- 无向图$G=(V,\{E\})$中从顶点v到顶点v'的**路径**(Path)是一个顶点序列，如果G是有向图，则路径也是有向的。
- **路径的长度**是路径上边或弧的数目。
- 第一个顶点和最后一个顶点相同的路径称为**回路**或**环**(Cycle)。
- 序列中顶点不重复出现的路径称为**简单路径**。
- 除第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为**简单回路**或**简单环**。

- 在无向图G中，如果从顶点v到顶点v'有路径，则称v和v'是**连通**的。
- 如果图中任意两个顶点都是连通的，则称G是**连通图**(Connected Graph)。
- **连通分量**(Connected Component)指的是无向图中的极大连通子图。
- 在有向图G中，如果对于每一对顶点，互相都存在路径，则称G是**强连通图**。
- 有向图中的极大强连通子图称做有向图的**强连通分量**。

- 对连通图进行遍历，过程中所经过的边和顶点的组合可看做是一棵普通树，通常称为**生成树**。
- 连通图中，由于任意两顶点之间可能含有多条通路，遍历连通图的方式有多种，往往一张连通图可能有多种不同的生成树与之对应。
- 一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的$n-1$条边。
- 如果在一棵生成树上添加一条边，必定构成一个环。
- 一棵有n个顶点的生成树有且仅有$n-1$条边。如果一个图有n个顶点和小于$n-1$条边，则是非连通图。如果它多于$n-1$条边，则一定有环。但是，有$n-1$条边的图不一定是生成树。

如果一个有向图恰有一个顶点的入度为 0，其余顶点的入度均为 1，则是一棵有向树。一个有向图的**生成森林**由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。图 7.6 所示为其一例。

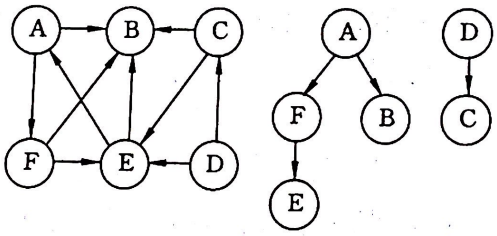


图 7.6　一个有向图及其生成森林

- 生成树是对应连通图来说，而生成森林是对应非连通图来说的。

图的存储结构

应根据具体的图和需要进行的操作，设计恰当的结点结构和表结构。

数组表示法

用两个数组分别存储数据元素(顶点)的信息和数据元素之间的关系(边或弧)的信息。

```
//图的数组表示法
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <iostream>

using namespace std;

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
# define OVERFLOW -2

typedef int Status;
typedef int Boolean;

# define INFINITY INT_MAX
//顶点关系类型，与INFINITY类型一致
typedef int VRType;
//最大顶点个数
# define MAX_VERTEX_NUM 26

//存储弧的相关信息，一般是结构体类型，没有时设为空
typedef char* InfoType;
```

```
//有向图，有向网，无向图，无向网
enum GraphKind {
    DG, DN, UDG, UDN
};

//VertexType存储顶点的一切信息，如名称、坐标等
# define MAX_NAME 9
struct VertexType {
  char name[MAX_NAME];
};

void Visit(VertexType ver) {
  cout << ver.name;
}

void Input(VertexType &ver) {
  cin >> ver.name;
}

//弧信息结构
typedef struct {
//  顶点关系类型
//  对于无权图，用1、0表示相邻否，对带权图，则为权值，也可以定义为浮点型
  VRType adj;
//  该弧相关信息的指针，可省略
  InfoType *info;
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

//图的结构
struct MGraph {
//  顶点向量
  VertexType vexs[MAX_VERTEX_NUM];
//  邻接矩阵
  AdjMatrix arcs;
//  图的当前顶点数和弧数
  int vexnum, arcnum;
//  图的种类
  GraphKind kind;
};

//若存在顶点u则返回在图中的序号
int LocateVex(MGraph G, VertexType u) {
  int i;
  for (i = 0; i < G.vexnum; i++) {
    if (strcmp(u.name, G.vexs[i].name) == 0)
      return i;
  }
  return -1;
}
```

```
//邻接矩阵构造有向图
void CreateDG(MGraph &G) {
  int i, j, k, IncInfo;
  VertexType v1, v2;
  cout << "输入有向图G的顶点数、弧数、弧是否含相关信息(0/1): " << endl;
//  录入顶点数和弧数
  cin >> G.vexnum >> G.arcnum >> IncInfo;
  printf("输入%d个顶点的值: \n", G.vexnum);
//  录入顶点信息
  for (i = 0; i < G.vexnum; i++)
    Input(G.vexs[i]);
  for (i = 0; i < G.vexnum; i++) {
    for (j = 0; j < G.vexnum; j++) {
//      不相邻
      G.arcs[i][j].adj = 0;
      G.arcs[i][j].info = NULL;
    }
  }
  printf("输入%d个条弧的弧尾 弧头: \n", G.arcnum);
  for (k = 0; k < G.arcnum; k++) {
    cin >> v1.name >> v2.name;
    i = LocateVex(G, v1); //弧尾的序号
    j = LocateVex(G, v2); //弧头的序号
//    录入弧信息
    G.arcs[i][j].adj = 1;
    if (IncInfo)
      ;
  }
  G.kind = DG;
}

//邻接矩阵构造有向网G
void CreateDN(MGraph &G) {
  int i, j, k, IncInfo;
  VRType w;
  VertexType v1, v2;
  cout << "输入有向网G的顶点数、弧数、弧是否含相关信息(0/1): " << endl;
  cin >> G.vexnum >> G.arcnum >> IncInfo;
  printf("输入%d个顶点的值: \n", G.vexnum);
  for (i = 0; i < G.vexnum; i++)
    Input(G.vexs[i]);
  for (i = 0; i < G.vexnum; i++) {
    for (j = 0; j < G.vexnum; j++) {
//      网, 不相邻
      G.arcs[i][j].adj = INFINITY;
      G.arcs[i][j].info = NULL;
    }
  }
```

```
  printf("输入%d个条弧的弧尾 弧头 权值: \n", G.arcnum);
  for (k = 0; k < G.arcnum; k++) {
    cin >> v1.name >> v2.name >> w;
    i = LocateVex(G, v1); //弧尾的序号
    j = LocateVex(G, v2); //弧头的序号
    G.arcs[i][j].adj = w;
    if (IncInfo)
      ;
  }
  G.kind = DN;
}

//邻接矩阵构造无向图G
void CreateUDG(MGraph &G) {
  int i, j, k, IncInfo;
  VertexType v1, v2;
  cout << "输入无向图G的顶点数、边数、边是否含相关信息(0/1): " << endl;
  cin >> G.vexnum >> G.arcnum >> IncInfo;
  printf("输入%d个顶点的值: \n", G.vexnum);
  for (i = 0; i < G.vexnum; i++)
    Input(G.vexs[i]);
  for (i = 0; i < G.vexnum; i++) {
    for (j = 0; j < G.vexnum; j++) {
      G.arcs[i][j].adj = 0;
      G.arcs[i][j].info = NULL;
    }
  }
  printf("输入%d个条边的顶点1 顶点2: \n", G.arcnum);
  for (k = 0; k < G.arcnum; k++) {
    cin >> v1.name >> v2.name;
    i = LocateVex(G, v1); //弧尾的序号
    j = LocateVex(G, v2); //弧头的序号
    G.arcs[i][j].adj = 1;
    if (IncInfo)
      ;
    G.arcs[j][i] = G.arcs[i][j];
  }
  G.kind = UDG;
}

//邻接矩阵构造无向网G
void CreateUDN(MGraph &G) {
  int i, j, k, IncInfo;
  VRType w;
  VertexType v1, v2;
  cout << "输入无向网G的顶点数、弧数、弧是否含相关信息(0/1): " << endl;
  cin >> G.vexnum >> G.arcnum >> IncInfo;
  printf("输入%d个顶点的值: \n", G.vexnum);
  for (i = 0; i < G.vexnum; i++)
```

```
    Input(G.vexs[i]);
  for (i = 0; i < G.vexnum; i++) {
    for (j = 0; j < G.vexnum; j++) {
//      网，不相邻
      G.arcs[i][j].adj = INFINITY;
      G.arcs[i][j].info = NULL;
    }
  }
  printf("输入%d个条弧的弧尾 弧头 权值：\n", G.arcnum);
  for (k = 0; k < G.arcnum; k++) {
    cin >> v1.name >> v2.name >> w;
    i = LocateVex(G, v1); //弧尾的序号
    j = LocateVex(G, v2); //弧头的序号
    G.arcs[i][j].adj = w;
    if (IncInfo)
      ;
    G.arcs[j][i] = G.arcs[i][j];
  }
  G.kind = UDN;
}

void CreateGraph(MGraph &G) {
  cout << "输入图G的类型(有向图：0；有向网：1；无向图：2；无向网：3)：";
  scanf("%d", &G.kind);
  switch(G.kind) {
    case DG:
      CreateDG(G);
      break;
    case DN:
      CreateDN(G);
      break;
    case UDG:
      CreateUDG(G);
      break;
    case UDN:
      CreateUDN(G);
      break;
  }
}

//返回序号为v的顶点信息
VertexType GetVex(MGraph G, int v) {
  if (v < 0 || v >= G.vexnum)
    exit(OVERFLOW);
  return G.vexs[v];
}

//v是G中的某个顶点，对v赋新值value
Status PutVex(MGraph &G, VertexType v, VertexType value) {
```

```
  int k = LocateVex(G, v);
  if (k < 0)
    return ERROR;
  G.vexs[k] = value;
  return OK;
}

//v是顶点序号，返回v的第一个邻接顶点的序号
int FirstAdjVex(MGraph G, int v) {
  int i;
  VRType j = 0;
//  网
  if (G.kind % 2)
    j = INFINITY;
  for (i = 0; i < G.vexnum; i++) {
    if (G.arcs[v][i].adj != j)
      return i;
  }
  return -1;
}

//在图G中增添新顶点v，不增添相关的弧，交给InsertArc()
void InsertVex(MGraph &G, VertexType v) {
  int i;
  VRType j = 0;
  if (G.kind % 2)
    j = INFINITY;
  G.vexs[G.vexnum] = v;
  for (i = 0; i <= G.vexnum; i++) {
//    初始化新的行和列
    G.arcs[G.vexnum][i].adj = G.arcs[i][G.vexnum].adj = j;
    G.arcs[G.vexnum][i].info = G.arcs[i][G.vexnum].info = NULL;
  }
  G.vexnum++;
}


//在G中添加<v, w>，若为无向图，再添加<w, v>
Status InsertArc(MGraph &G, VertexType v, VertexType w) {
  int i, v1, w1;
  v1 = LocateVex(G, v);
  w1 = LocateVex(G, w);
  if (v1 < 0 || w1 < 0)
    return ERROR;
  G.arcnum++;
//  网
  if (G.kind % 2) {
    cout << "输入此弧的权值：";
    cin >> G.arcs[v1][w1].adj;
```

```
  }
//  图
  else {
    G.arcs[v1][w1].adj = 1;
  }
  cout << "是否有相关信息? ";
  cin >> i;
  if (i)
    ;
//  无向
  if (G.kind > 1)
    G.arcs[w1][v1] = G.arcs[v1][w1];
  return OK;
}

//在G中删除弧<v, w>, 若为无向图, 再删除<w, v>
Status DeleteArc(MGraph &G, VertexType v, VertexType w) {
  int v1, w1;
  VRType j = 0;
//  网
  if (G.kind % 2)
    j = INFINITY;
  v1 = LocateVex(G, v);
  w1 = LocateVex(G, w);
  if (v1 < 0 || w1 < 0)
    return ERROR;
//  存在弧<v, w>
  if (G.arcs[v1][w1].adj != j) {
//    删除弧
    G.arcs[v1][w1].adj = j;
    if (G.arcs[v1][w1].info) {
      free(G.arcs[v1][w1].info);
      G.arcs[v1][w1].info = NULL;
    }
//    如果无向
    if (G.kind >= 2)
      G.arcs[w1][v1] = G.arcs[v1][w1];
    G.arcnum--;
  }
  return OK;
}

//删除G中顶点v及其相关的弧
Status DeleteVex(MGraph &G, VertexType v) {
  int i, j, k;
  k = LocateVex(G, v);
  if (k < 0)
    return ERROR;
//  删除由顶点v发出的所有弧
```

```
  for (i = 0; i < G.vexnum; i++)
    DeleteArc(G, v, G.vexs[i]);
//  有向图
  if (G.kind < 2) {
//    删除发向顶点v的所有弧
    for (i = 0; i < G.vexnum; i++)
      DeleteArc(G, G.vexs[i], v);
  }
//  序号k后面的顶点向量依次前移
  for (j = k + 1; j < G.vexnum; j++)
    G.vexs[j - 1] = G.vexs[j];

  for (i = 0; i < G.vexnum; i++) {
    for (j = k + 1; j < G.vexnum; j++) {
//      移动待删除顶点之右的矩阵元素
      G.arcs[i][j - 1] = G.arcs[i][j];
    }
  }

  for (i = 0; i < G.vexnum; i++) {
    for (j = k + 1; j < G.vexnum; j++) {
//      移动待删除顶点之下的矩阵元素
      G.arcs[j - 1][i] = G.arcs[j][i];
    }
  }
  G.vexnum--;
  return OK;
}

void DestroyGraph(MGraph &G) {
  int i;
  for (i = G.vexnum - 1; i >= 0; i--)
    DeleteVex(G, G.vexs[i]);
}

void Display(MGraph G) {
  // 输出邻接矩阵存储表示的图G
  int i,j;
  char s[7];
  switch(G.kind) {
    case DG:
      strcpy(s,"有向图");
      break;
    case DN:
      strcpy(s,"有向网");
      break;
    case UDG:
      strcpy(s,"无向图");
      break;
```

```
    case UDN:
      strcpy(s,"无向网");
  }
  printf("%d个顶点%d条边或弧的%s。顶点依次是：",G.vexnum,G.arcnum,s);
  for(i=0; i<G.vexnum; ++i) // 输出G.vexs
    printf("%s ",G.vexs[i]);
  printf("\nG.arcs.adj:\n"); // 输出G.arcs.adj
  for(i=0; i<G.vexnum; i++) {
    for(j=0; j<G.vexnum; j++)
      printf("%11d",G.arcs[i][j].adj);
    printf("\n");
  }
  printf("G.arcs.info:\n"); // 输出G.arcs.info
  printf("顶点1(弧尾) 顶点2(弧头) 该边或弧的信息：\n");
  for(i=0; i<G.vexnum; i++)
    if(G.kind<2) { // 有向
      for(j=0; j<G.vexnum; j++)
        if(G.arcs[i][j].info)
          printf("%5s %11s     %s\n",G.vexs[i],G.vexs[j],G.arcs[i][j].info);
    } // 加括号为避免if-else对配错
    else // 无向,输出上三角
      for(j=i+1; j<G.vexnum; j++)
        if(G.arcs[i][j].info)
          printf("%5s %11s     %s\n",G.vexs[i],G.vexs[j],G.arcs[i][j].info);
}

int main() {
  int i,j,k,n;
  MGraph g;
  VertexType v1,v2;
  printf("请顺序选择有向图,有向网,无向图,无向网\n");
  for(i=0; i<4; i++) { // 验证4种情况
    CreateGraph(g); // 构造图g
    Display(g); // 输出图g
    printf("插入新顶点，请输入顶点的值：");
    scanf("%s",v1);
    InsertVex(g,v1);
    printf("插入与新顶点有关的弧或边，请输入弧或边数：");
    scanf("%d",&n);
    for(k=0; k<n; k++) {
      printf("请输入另一顶点的值：");
      scanf("%s",v2);
      if(g.kind<=1) { // 有向
        printf("对于有向图或网，请输入另一顶点的方向(0:弧头 1:弧尾)：");
        scanf("%d",&j);
        if(j) // v2是弧尾
          InsertArc(g,v2,v1);
        else // v2是弧头
          InsertArc(g,v1,v2);
```

```
      } else // 无向
        InsertArc(g,v1,v2);
    }
    Display(g); // 输出图g
    printf("删除顶点及相关的弧或边，请输入顶点的值：");
    scanf("%s",v1);
    DeleteVex(g,v1);
    Display(g); // 输出图g
  }
  DestroyGraph(g); // 销毁图g
  return 0;
}
```

邻接表

邻接表(Adjacency List)是图的一种链式存储结构。在邻接表中，对图中每个顶点建立一个单链表，第i个单链表中的结点表示依附于顶点v_i的边(对有向图是以顶点v_i为尾的弧)。

表结点：

邻接点域	链域	数据域
adjvex	nextarc	info
指示与顶点v_i邻接的点在图中的位置	下一条边或弧的结点	相关信息

头结点：

数据域	链域
data	firstarc
指向链表中第一个结点	相关信息

若无向图中有n个顶点、e条边，则它的邻接表需要n个头结点和$2e$个表结点。
在边稀疏($e \ll \frac{n(n-1)}{2}$)的情况下，用邻接表表示图比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

逆邻接表：为了便于确定顶点的入度或以顶点v_i为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点v_i建立一个链接以v_i为头的弧的表。

```
# define MAX_NAME 9
struct VertexType {
  char name[MAX_NAME];
};

# define MAX_VERTEX_NUM 26
typedef char* InfoType;
```

```
typedef struct ArcNode {
//  该弧所指向的顶点的位置
  int adjvex;
//  指向下一条弧的指针
  struct ArcNode *nextarc;
  InfoType *info;
} ArcNode;

typedef struct VNode {
//  顶点信息
  VertexType data;
//  指向第一条依附该顶点的弧的指针
  ArcNode *firstarc;
} VNode, AdjList[MAX_VERTEX_NUM];

typedef struct {
  AdjList vertices;
  int vexnum, arcnum;
  int kind;
} ALGraph;
```

十字链表

十字链表(Orthogonal List)是*有向图*的另一种链式存储结构。可以看成是将有向图的邻接表和逆邻接表结合起来得到的一种链表。

弧结点：

尾域	头域	链域(指向弧头相同的下一条弧)	链域(指向弧尾相同的下一条弧)	相关信息
tailvex	headvex	hlink	tlink	info

顶点结点：

相关信息	以该顶点为弧头的第一个弧结点	以该顶点为弧尾的第一个弧结点
data	firstin	firstout

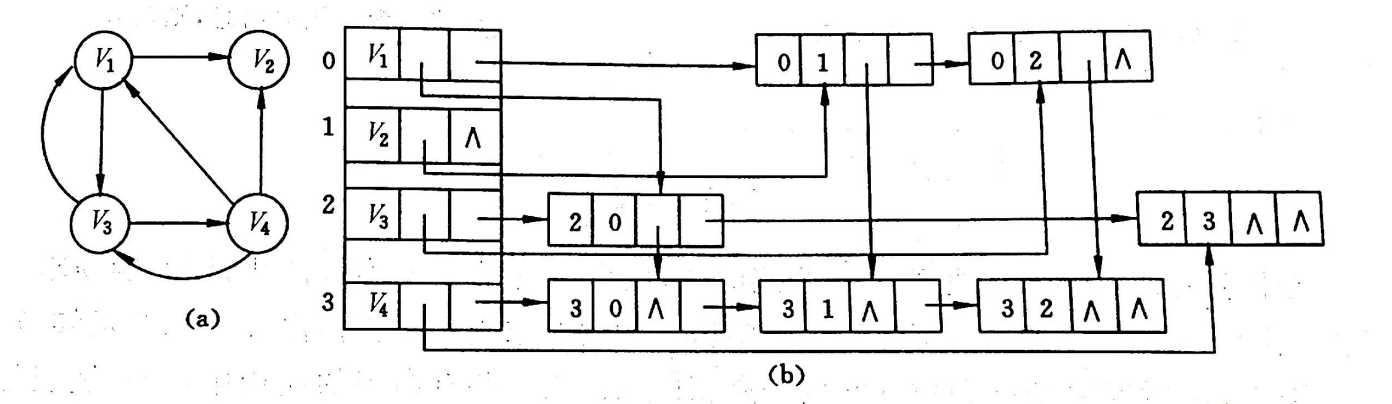


图 7.11　有向图的十字链表

```
# define MAX_VERTEX_NUM 20
typedef struct ArcBox {
//  该弧的尾和头顶点的位置
  int tailvex, headvex;
  struct ArcBox *hlink, *tlink;
  InfoType *info;
} ArcBox;

typedef struct VexNode {
  VertexType data;
  ArcBox *firstin, *firstout;
} VexNode;

typedef struct {
  VexNode xlist[MAX_VERTEX_NUM];
  int vexnum, arcnum;
} OLGraph;
```

邻接多重表

邻接多重表是*无向图*的另一种链式存储结构。在邻接表中每一条边有两个结点，分别在第i个和第j个链表中，给一些操作带来不便。

边结点：

mark	ivex	ilink	jvex	jlink	info

其中`mark`为标志域，可以标记该条边是否被搜索过，`ivex`和`jvex`为该边依附的两个顶点在图中的位置；`ilink`指向下一条依附于顶点`ivex`的边，`jlink`指向下一条依附于顶点`jvex`的边；`info`为相关信息。

顶点结点：

相关信息	指向第一条依附于该顶点的边
data	firstedge

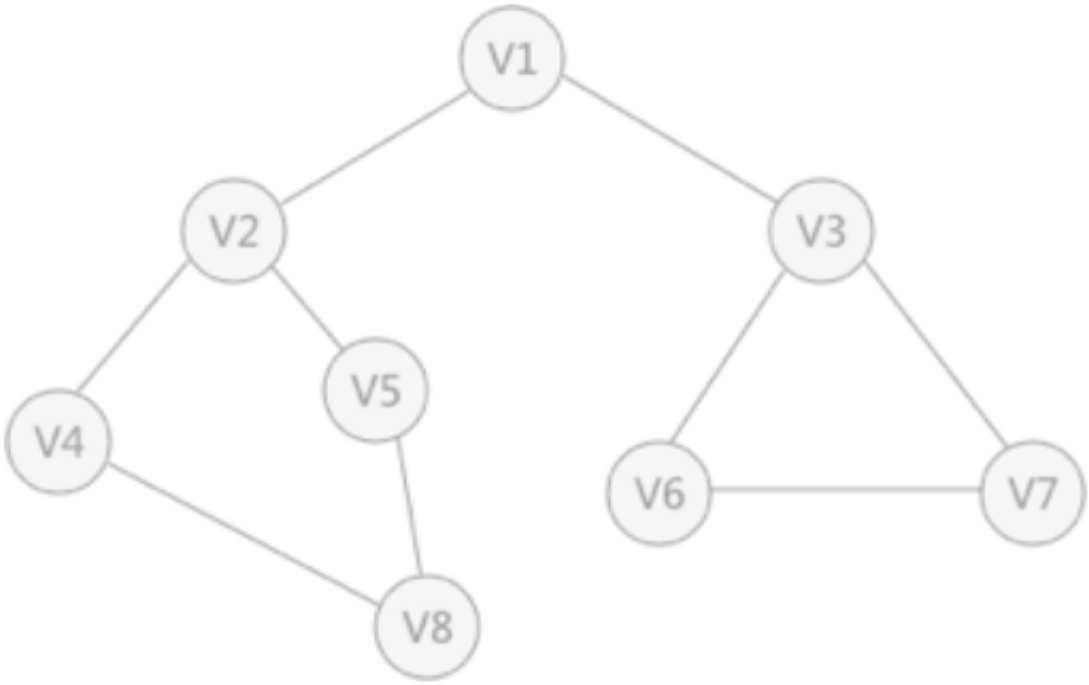
```
# define MAX_VERTEX_NUM 20
typedef enum {
    unvisited, visited
} VisitIf;
typedef struct EBox {
  VisitIf mark;
  int ivex, jvex;
  struct EBox *ilink, *jlink;
  InfoType *info;
} EBox;

typedef struct VexBox {
  VertexType data;
  EBox *firstedge;
} VexBox;

typedef struct {
  VexBox adjmulist[MAX_VERTEX_NUM];
  int vexnum, edgenum;
} AMLGraph;
```

图的遍历

为了避免同一顶点被访问多次，设置辅助数组`visited[0...n-1]`。

深度优先搜索

深度优先搜索(Depth First Search)遍历类似于树的先根遍历，是树的先根遍历的推广。



深度优先搜索的过程类似于树的先序遍历。例如上图是一个无向图，采用深度优先算法遍历这个图的过程为：

1. 首先任意找一个未被遍历过的顶点，例如从 V1 开始，由于 V1 率先访问过了，所以，需要标记 V1 的状态为访问过；
2. 然后遍历 V1 的邻接点，例如访问 V2 ，并做标记，然后访问 V2 的邻接点，例如 V4 (做标记)，然后 V8 ，然后 V5 ；
3. 当继续遍历 V5 的邻接点时，根据之前做的标记显示，所有邻接点都被访问过了。此时，从 V5 回退到 V8 ，看 V8 是否有未被访问过的邻接点，如果没有，继续回退到 V4 ， V2 ， V1 ；
4. 通过查看 V1 ，找到一个未被访问过的顶点 V3 ，继续遍历，然后访问 V3 邻接点 V6 ，然后 V7 ；
5. 由于 V7 没有未被访问的邻接点，所有回退到 V6 ，继续回退至 V3 ，最后到达 V1 ，发现没有未被访问的；
6. 最后一步需要判断是否所有顶点都被访问，如果还有没被访问的，以未被访问的顶点为第一个顶点，继续依照上边的方式进行遍历。

根据上边的过程，可以得到图 1 通过深度优先搜索获得的顶点的遍历次序为：

```
V1 -> V2 -> V4 -> V8 -> V5 -> V3 -> V6 -> V7
```

邻接矩阵的DFS：

```
Boolean visited[MAX_VERTEX_NUM];

void DFS(MGraph G, int i) {
  int j;
  visited[i] = TRUE;
  cout << G.vexs[i].name << " ";
  for (j = 0; j < G.vexnum; j++) {
    if (G.arcs[i][j].adj == 1 && !visited[j])
      DFS(G, j);
  }
}

void DFSTraverse(MGraph G) {
  int i;
//  初始化所有顶点未访问状态
  for (i = 0; i < G.vexnum; i++)
    visited[i] = FALSE;
//  for (i = 0; i < G.vexnum; i++) {
////    对未访问的顶点调用DFS
//    if (!visited[i])
//      DFS(G, i);
//  }
  DFS(G, 0);
  cout << endl;
}
```

广度优先搜索

广度优先搜索(Broadth First Search)遍历类似于树的层次遍历的过程。

广度优先搜索类似于树的层次遍历。从图中的某一顶点出发，遍历每一个顶点时，依次遍历其所有的邻接点，然后再从这些邻接点出发，同样依次访问它们的邻接点。按照此过程，直到图中所有被访问过的顶点的邻接点都被访问到。

最后还需要做的操作就是查看图中是否存在尚未被访问的顶点，若有，则以该顶点为起始点，重复上述遍历的过程。

如上图，假设 V1 作为起始点，遍历其所有的邻接点 V2 和 V3 ，以 V2 为起始点，访问邻接点 V4 和 V5 ，以 V3 为起始点，访问邻接点 V6 、 V7 ，以 V4 为起始点访问 V8 ，以 V5 为起始点，由于 V5 所有的起始点已经全部被访问，所有直接略过， V6 和 V7 也是如此。
以 V1 为起始点的遍历过程结束后，判断图中是否还有未被访问的点，由于图 1 中没有了，所以整个图遍历结束。
遍历顶点的顺序为：

```
V1 -> V2 -> v3 -> V4 -> V5 -> V6 -> V7 -> V8
```

邻接矩阵的BFS：

```
Boolean visited[MAX_VERTEX_NUM];

void BFSTraverse(MGraph G) {
  int i, j;
  queue<int> Q;
  for (i = 0; i < G.vexnum; i++)
    visited[i] = FALSE;
  for (i = 0; i < G.vexnum; i++) {
    if (!visited[i]) {
      visited[i] = TRUE;
      cout << G.vexs[i].name << " ";
      Q.push(i);
      while (!Q.empty()) {
        i = Q.front();
        Q.pop();
        for (j = 0; j < G.vexnum; j++) {
          if (G.arcs[i][j].adj == 1 && !visited[j]) {
            visited[j] = TRUE;
            cout << G.vexs[j].name << " ";
            Q.push(j);
          }
        }
      }
    }
  }
}
```

图的连通性

无向图的连通分量和生成树

在对无向图进行遍历时，对于连通图，仅需从图中任一顶点触发，进行DFS或BFS，便可访问到图中的所有顶点。对于非连通图，则需从多个顶点触发进行搜索，而每一次从一个新的起始点触发进行搜索过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。

深度优先生成树(森林)
广度优先生成树(森林)

对于非连通图，生成森林。

有向图的强连通分量

深度优先搜索是求有向图的强连通分量的一个新的有效方法。

最小生成树

构造连通网的最小代价生成树(Minimum Cost Spaning Tree)，简称最小生成树。
一棵生成树的代价就是树上各边的代价之和。

Prim算法

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

对于给定的连通网，起始状态全部顶点都归为 B 类。在找最小生成树时，选定任意一个顶点作为起始点，并将之从 B 类移至 A 类；然后找出 B 类中到 A 类中的顶点之间权值最小的顶点，将之从 B 类移至 A 类，如此重复，直到 B 类中没有顶点为止。所走过的顶点和边就是该连通图的最小生成树。

```
typedef struct closedge {
    // 记录从顶点集U到V-U的代价最小的边的辅助数组定义
    VertexType adjvex;
    VRType lowcost;
} minside[MAX_VERTEX_NUM];

int minimum(minside SZ, MGraph G) {
    // 求SZ.lowcost的最小正值，并返回其在SZ中的序号
    int i = 0, j, k, min;
    while (!SZ[i].lowcost)
        i++;
    min = SZ[i].lowcost; // 第一个不为0的值
    k = i;
    for (j = i + 1; j < G.vexnum; j++)
        if (SZ[j].lowcost > 0 && min > SZ[j].lowcost) { // 找到新的大于0的最小值
            min = SZ[j].lowcost;
            k = j;
        }
    return k;
}
```

```
void MiniSpanTree_PRIM(MGraph G, VertexType u) {
    // 用普里姆算法从第u个顶点出发构造网G的最小生成树T，输出T的各条边。算法7.9
    int i, j, k;
    minside closedge;
    k = LocateVex(G, u);
    for (j = 0; j < G.vexnum; ++j) { // 辅助数组初始化
        strcpy(closedge[j].adjvex, u);
        closedge[j].lowcost = G.arcs[k][j].adj;
    }
    closedge[k].lowcost = 0; // 初始,U={u}
    printf("最小代价生成树的各条边为:\n");
    for (i = 1; i < G.vexnum; ++i) {
        // 选择其余G.vexnum-1个顶点
        k = minimum(closedge, G); // 求出T的下一个结点：第k顶点
        printf("(%s-%s)\n", closedge[k].adjvex, G.vexs[k]); // 输出生成树的边
        closedge[k].lowcost = 0; // 第k顶点并入U集
        for (j = 0; j < G.vexnum; ++j)
            if (G.arcs[k][j].adj < closedge[j].lowcost) {
                // 新顶点并入U集后重新选择最小边
                strcpy(closedge[j].adjvex, G.vexs[k]);
                closedge[j].lowcost = G.arcs[k][j].adj;
            }
    }
}
```

Kruskal算法

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

由于最小生成树本身是一棵生成树，所以需要时刻满足以下两点：

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

> 连接n个顶点在不产生回路的情况下，只需要$n-1$条边。

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

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

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

```
void kruskal(MGraph G) {
    int set[MAX_VERTEX_NUM], i, j;
```

```
    int k = 0, a = 0, b = 0, min = G.arcs[a][b].adj;
    for (i = 0; i < G.vexnum; i++)
        set[i] = i; // 初态，各顶点分别属于各个集合
    printf("最小代价生成树的各条边为:\n");
    while (k < G.vexnum - 1) // 最小生成树的边数小于顶点数-1
    { // 寻找最小权值的边
        for (i = 0; i < G.vexnum; ++i)
            for (j = i + 1; j < G.vexnum; ++j) // 无向网，只在上三角查找
                if (G.arcs[i][j].adj < min) {
                    min = G.arcs[i][j].adj; // 最小权值
                    a = i; // 边的一个顶点
                    b = j; // 边的另一个顶点
                }
        min = G.arcs[a][b].adj = INFINITY; // 删除上三角中该边，下次不再查找
        if (set[a] != set[b]) // 边的两顶点不属于同一集合
        {
            printf("%s-%s\n", G.vexs[a], G.vexs[b]); // 输出该边
            k++; // 边数+1
            for (i = 0; i < G.vexnum; i++)
                if (set[i] == set[b]) // 将顶点b所在集合并入顶点a集合中
                    set[i] = set[a];
        }
    }
}
```

关节点和重连通图

在一个无向图中，如果删除某个顶点及其相关联的边后，原来的图被分割为两个及以上的连通分量，则称该顶点为无向图中的一个**关节点**(articulation point)。

一个没有关节点的连通图称为**重连通图**(biconnected graph)。

在无向图中，如果任意两个顶点之间含有不止一条通路，这个图就被称为重连通图。在重连通图中，在删除某个顶点及该顶点相关的边后，图中各顶点之间的连通性也不会被破坏。

有向无环图及其应用

一个无环的有向图称做**有向无环图**(directed acycline graph)，简称DAG图。

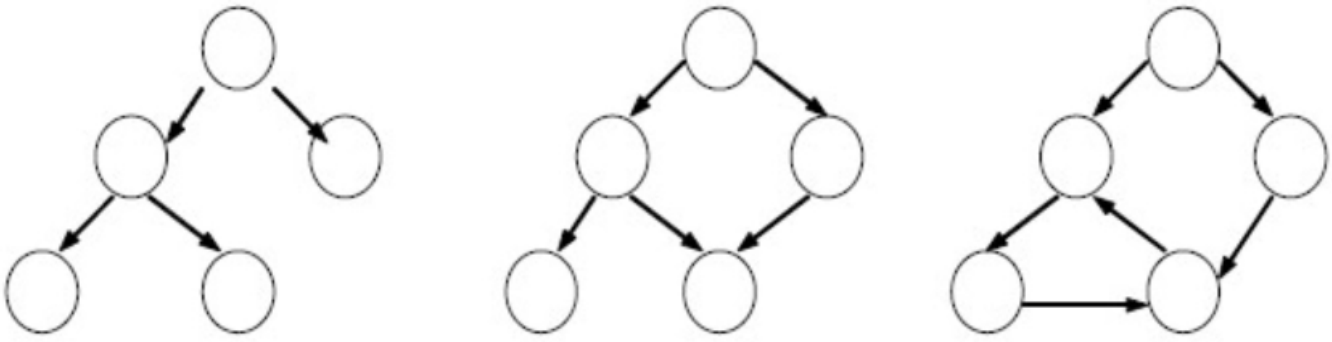


图 8.30　有向树、DAG 图和有向图示意

拓扑排序

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，这样的有向图为顶点表示活动的网，称为**AOV网**(Activity On Vertex NetWork)。AOV网中的弧表示活动之间存在的某种制约关系。

在AOV网中，若不存在回路，则所有活动可排列成一个线性序列，使得每个活动的所有前驱活动都排在该活动的前面，我们把此序列叫做**拓扑序列**(Topological order)。

拓扑排序就是对一个有向图构造拓扑序列的过程。

构造时会有两个结果，如果此网的全部顶点都被输出，则说明它是不存在环(回路)的AOV网；如果输出顶点少了，哪怕是少了一个，也说明这个网存在环(回路)，不是AOV网。

对有向无环图进行拓扑排序，只需要遵循两个原则：

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

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

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

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

typedef struct {
    AdjList vertices;//图中顶点及各邻接点数组
    int vexnum, arcnum;//记录图中顶点数和边或弧数
} ALGraph;

//找到顶点对应在邻接表数组中的位置下标
int LocateVex(ALGraph G, VertexType u) {
    for (int i = 0; i < G.vexnum; i++) {
        if (G.vertices[i].data == u) {
            return i;
        }
    }
    return -1;
}
```

```c
//创建AOV网，构建邻接表
void CreateAOV(ALGraph **G) {
    *G = (ALGraph *) malloc(sizeof(ALGraph));

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

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

        int locate = LocateVex(*(*G), initial);
        p->nextarc = (*G)->vertices[locate].firstarc;
        (*G)->vertices[locate].firstarc = p;
    }
}

//结构体定义栈结构
typedef struct stack {
    VertexType data;
    struct stack *next;
} stack;

//初始化栈结构
void initStack(stack **S) {
    (*S) = (stack *) malloc(sizeof(stack));
    (*S)->next = NULL;
}

//判断链表是否为空
bool StackEmpty(stack S) {
    if (S.next == NULL) {
        return true;
    }
    return false;
}

//进栈，以头插法将新结点插入到链表中
void push(stack *S, VertexType u) {
    stack *p = (stack *) malloc(sizeof(stack));
    p->data = u;
    p->next = NULL;
```

```
    p->next = S->next;
    S->next = p;
}

//弹栈函数，删除链表首元结点的同时，释放该空间，并将该结点中的数据域通过地址传值给变量i;
void pop(stack *S, VertexType *i) {
    stack *p = S->next;
    *i = p->data;
    S->next = S->next->next;
    free(p);
}

//统计各顶点的入度
void FindInDegree(ALGraph G, int indegree[]) {
    //初始化数组，默认初始值全部为0
    for (int i = 0; i < G.vexnum; i++) {
        indegree[i] = 0;
    }
    //遍历邻接表，根据各链表中结点的数据域存储的各顶点位置下标，在indegree数组相应位置+1
    for (int i = 0; i < G.vexnum; i++) {
        ArcNode *p = G.vertices[i].firstarc;
        while (p) {
            indegree[p->adjvex]++;
            p = p->nextarc;
        }
    }
}

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

```
        for (ArcNode *p = G.vertices[index].firstarc; p; p = p->nextarc) {
            VertexType k = p->adjvex;
            if (!(--indegree[k])) {
                //顶点入度为0，入栈
                push(S, k);
            }
        }
    }
    //如果count值小于顶点数量，表明该有向图有环
    if (count < G.vexnum) {
        printf("该图有回路");
        return;
    }
}

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

关键路径

AOE网(Activity On Edge)是在AOV网的基础上，其中每一个边都具有各自的权值，是一个带权有向无环网。通常，其中权值表示活动持续的时间。

由于在AOE网中有些活动可以并行地进行，所以完成工程的最短时间是从开始点到完成点的最长路径的长度(指路径上各活动持续时间之和，不是路径上弧的数目)。路径长度最长的路径叫做**关键路径**(Critical Path)。

```
// algo7-5.cpp 求关键路径。实现算法7.13、7.14的程序
#include"c1.h"

#define MAX_NAME 5 // 顶点字符串的最大长度+1
typedef int InfoType;
typedef char VertexType[MAX_NAME]; // 字符串类型
#include"c7-21.h"
#include"bo7-2.cpp"
#include"func7-1.cpp"

int ve[MAX_VERTEX_NUM]; // 事件最早发生时间，全局变量(用于算法7.13和算法7.14)

typedef int SElemType; // 栈元素类型
#include"c3-1.h" // 顺序栈的存储结构
#include"bo3-1.cpp" // 顺序栈的基本操作

Status TopologicalOrder(ALGraph G, SqStack &T) { // 算法7.13 有向网G采用邻接表存储结构，求各顶点事件的最早发生时间ve(全局变量)。T为拓扑序列
    // 顶点栈,S为零入度顶点栈。若G无回路，则用栈T返回G的一个拓扑序列,且函数值为OK,否则为ERROR
```

```
    int i, k, count = 0; // 已入栈顶点数，初值为0
    int indegree[MAX_VERTEX_NUM]; // 入度数组，存放各顶点当前入度数
    SqStack S;
    ArcNode *p;
    FindInDegree(G, indegree); // 对各顶点求入度indegree[]，在func7-1.cpp中
    InitStack(S); // 初始化零入度顶点栈S
    printf("拓扑序列：");
    for (i = 0; i < G.vexnum; ++i) // 对所有顶点i
        if (!indegree[i]) // 若其入度为0
            Push(S, i); // 将i入零入度顶点栈S
    InitStack(T); // 初始化拓扑序列顶点栈
    for (i = 0; i < G.vexnum; ++i) // 初始化ve[]=0(最小值，先假定每个事件都不受其他事件约束)
        ve[i] = 0;
    while (!StackEmpty(S)) // 当零入度顶点栈S不空
    {
        Pop(S, i); // 从栈S将已拓扑排序的顶点j弹出
        printf("%s ", G.vertices[i].data);
        Push(T, i); // j号顶点入逆拓扑排序栈T(栈底元素为拓扑排序的第1个元素)
        ++count; // 对入栈T的顶点计数
        for (p = G.vertices[i].firstarc; p; p = p->nextarc) { // 对i号顶点的每个邻接点
            k = p->data.adjvex; // 其序号为k
            if (--indegree[k] == 0) // k的入度减1，若减为0，则将k入栈S
                Push(S, k);
            if (ve[i] + *(p->data.info) > ve[k]) // *(p->data.info)是<i,k>的权值
                ve[k] = ve[i] + *(p->data.info); // 顶点k事件的最早发生时间要受其直接前驱顶点i
事件的
        }                   // 最早发生时间和<i,k>的权值约束。由于i已拓扑有序，故ve[i]不再改变
    }
    if (count < G.vexnum) {
        printf("此有向网有回路\n");
        return ERROR;
    } else
        return OK;
}

Status CriticalPath(ALGraph G) { // 算法7.14 G为有向网，输出G的各项关键活动
    int vl[MAX_VERTEX_NUM]; // 事件最迟发生时间
    SqStack T;
    int i, j, k, ee, el, dut;
    ArcNode *p;
    if (!TopologicalOrder(G, T)) // 产生有向环
        return ERROR;
    j = ve[0]; // j的初值
    for (i = 1; i < G.vexnum; i++)
        if (ve[i] > j)
            j = ve[i]; // j=Max(ve[]) 完成点的最早发生时间
    for (i = 0; i < G.vexnum; i++) // 初始化顶点事件的最迟发生时间
        vl[i] = j; // 为完成点的最早发生时间(最大值)
    while (!StackEmpty(T)) // 按拓扑逆序求各顶点的vl值
```

```
        for (Pop(T,
                 j), p = G.vertices[j].firstarc; p; p = p->nextarc) { // 弹出栈T的元素,赋给j,p指向j的后继事件k,事件k的最迟发生时间已确定(因为是逆拓扑排序)
            k = p->data.adjvex;
            dut = *(p->data.info); // dut=<j,k>的权值
            if (vl[k] - dut < vl[j])
                vl[j] = vl[k] - dut; // 事件j的最迟发生时间要受其直接后继事件k的最迟发生时间
        }                       // 和<j,k>的权值约束。由于k已逆拓扑有序，故vl[k]不再改变
    printf("\ni ve[i] vl[i]\n");
    for (i = 0; i < G.vexnum; i++) // 初始化顶点事件的最迟发生时间
    {
        printf("%d   %d     %d", i, ve[i], vl[i]);
        if (ve[i] == vl[i])
            printf(" 关键路径经过的顶点");
        printf("\n");
    }
    printf("j   k  权值  ee  el\n");
    for (j = 0; j < G.vexnum; ++j) // 求ee，el和关键活动
        for (p = G.vertices[j].firstarc; p; p = p->nextarc) {
            k = p->data.adjvex;
            dut = *(p->data.info); // dut=<j,k>的权值
            ee = ve[j]; // ee=活动<j,k>的最早开始时间(在j点)
            el = vl[k] - dut; // el=活动<j,k>的最迟开始时间(在j点)
            printf("%s→%s %3d %3d %3d ", G.vertices[j].data, G.vertices[k].data, dut, ee, el);
            // 输出各边的参数
            if (ee == el) // 是关键活动
                printf("关键活动");
            printf("\n");
        }
    return OK;
}

void main() {
    ALGraph h;
    printf("请选择有向网\n");
    CreateGraph(h); // 构造有向网h，在bo7-2.cpp中
    Display(h); // 输出有向网h，在bo7-2.cpp中
    CriticalPath(h); // 求h的关键路径
}
```

最短路径

路径上第一个顶点为**源点**(Sourse)，最后一个顶点为**终点**(Destination)。
迪杰斯特拉算法计算的是有向网中的某个顶点到其余所有顶点的最短路径；弗洛伊德算法计算的是任意两顶点之间的最短路径。
最短路径算法既适用于有向网，也同样适用于无向网。

Dijkstra算法

迪杰斯特拉算法计算的是从网中一个顶点到其它顶点之间的最短路径问题。

```
// algo7-6.cpp 实现算法7.15的程序。迪杰斯特拉算法的实现
#include"c1.h"

#define MAX_NAME 5 // 顶点字符串的最大长度+1
#define MAX_INFO 20 // 相关信息字符串的最大长度+1
typedef int VRType;
typedef char InfoType;
typedef char VertexType[MAX_NAME];

#include"c7-1.h" // 邻接矩阵存储表示
#include"bo7-1.cpp" // 邻接矩阵存储表示的基本操作

typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 路径矩阵，二维数组
typedef int ShortPathTable[MAX_VERTEX_NUM]; // 最短距离表，一维数组

void ShortestPath_DIJ(MGraph G, int v0, PathMatrix P, ShortPathTable D) { // 用Dijkstra
算法求有向网G的v0顶点到其余顶点v的最短路径P[v]及带权长度
    // D[v]。若P[v][w]为TRUE，则w是从v0到v当前求得最短路径上的顶点。
    // final[v]为TRUE当且仅当v∈S，即已经求得从v0到v的最短路径 算法7.15
    int v, w, i, j, min;
    Status final[MAX_VERTEX_NUM]; // 辅助矩阵，为真表示该顶点到v0的最短距离已求出，初值为假
    for (v = 0; v < G.vexnum; ++v) {
        final[v] = FALSE; // 设初值
        D[v] = G.arcs[v0][v].adj; // D[]存放v0到v的最短距离，初值为v0到v的直接距离
        for (w = 0; w < G.vexnum; ++w)
            P[v][w] = FALSE; // 设P[][]初值为FALSE，没有路径
        if (D[v] < INFINITY) // v0到v有直接路径
            P[v][v0] = P[v][v] = TRUE; // 一维数组p[v][]表示源点v0到v最短路径通过的顶点
    }
    D[v0] = 0; // v0到v0距离为0
    final[v0] = TRUE; // v0顶点并入S集
    for (i = 1; i < G.vexnum; ++i) // 其余G.vexnum-1个顶点
    { // 开始主循环，每次求得v0到某个顶点v的最短路径，并将v并入S集
        min = INFINITY; // 当前所知离v0顶点的最近距离，设初值为∞
        for (w = 0; w < G.vexnum; ++w) // 对所有顶点检查
            if (!final[w] && D[w] < min) //在S集之外的顶点中找离v0最近的顶点，并将其赋给v，距离
赋给min
            {
                v = w;
                min = D[w];
            }
        final[v] = TRUE; // 将v并入S集
        for (w = 0; w < G.vexnum; ++w) // 根据新并入的顶点，更新不在S集的顶点到v0的距离和路径数
组
            if (!final[w] && min < INFINITY && G.arcs[v][w].adj < INFINITY &&
```

```
                (min + G.arcs[v][w].adj < D[w])) { // w不属于S集且v0→v→w的距离＜目前v0→w的距离
                D[w] = min + G.arcs[v][w].adj; // 更新D[w]
                for (j = 0; j < G.vexnum; ++j) // 修改P[w]，v0到w经过的顶点包括v0到v经过的顶点再加上顶点w
                    P[w][j] = P[v][j];
                P[w][w] = TRUE;
            }
    }
}

void main() {
    int i, j;
    MGraph g;
    PathMatrix p; // 二维数组，路径矩阵
    ShortPathTable d; // 一维数组，最短距离表
    CreateDN(g); // 构造有向网g
    Display(g); // 输出有向网g
    ShortestPath_DIJ(g, 0, p, d);//以g中位置为0的顶点为源点，球其到其余各顶点的最短距离。存于d中
    printf("最短路径数组p[i][j]如下:\n");
    for (i = 0; i < g.vexnum; ++i) {
        for (j = 0; j < g.vexnum; ++j)
            printf("%2d", p[i][j]);
        printf("\n");
    }
    printf("%s到各顶点的最短路径长度为：\n", g.vexs[0]);
    for (i = 0; i < g.vexnum; ++i)
        if (i != 0)
            printf("%s-%s:%d\n", g.vexs[0], g.vexs[i], d[i]);
}
```

Floyd算法

每一对顶点之间的最短路径。

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

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

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

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

```
Dis(A, K)+ Dis(K, B)< Dis(A, B)
```

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

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

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

```
// func7-2.cpp 算法7.16,algo7-7.cpp和algo7-9.cpp用到
void ShortestPath_FLOYD(MGraph G, PathMatrix P, DistancMatrix D) { // 用Floyd算法求有向网G中各对顶点v和w之间的最短路径P[v][w]及其带权长度D[v][w]。
    // 若P[v][w][u]为TRUE，则u是从v到w当前求得最短路径上的顶点。算法7.16
    int u, v, w, i;
    for (v = 0; v < G.vexnum; v++) // 各对结点之间初始已知路径及距离
        for (w = 0; w < G.vexnum; w++) {
            D[v][w] = G.arcs[v][w].adj; // 顶点v到顶点w的直接距离
            for (u = 0; u < G.vexnum; u++)
                P[v][w][u] = FALSE; // 路径矩阵初值
            if (D[v][w] < INFINITY) // 从v到w有直接路径
                P[v][w][v] = P[v][w][w] = TRUE; // 由v到w的路径经过v和w两点
        }
    for (u = 0; u < G.vexnum; u++)
        for (v = 0; v < G.vexnum; v++)
            for (w = 0; w < G.vexnum; w++)
                if (D[v][u] < INFINITY && D[u][w] < INFINITY && D[v][u] + D[u][w] < D[v][w]) { // 从v经u到w的一条路径更短
                    D[v][w] = D[v][u] + D[u][w]; // 更新最短距离
                    for (i = 0; i < G.vexnum; i++)
                        P[v][w][i] = P[v][u][i] || P[u][w][i]; // 从v到w的路径经过从v到u和从u到w的所有路径
                }
}
```

7 查找

概念

- **查找表(Search Table)**：由同一类型的数据元素或记录构成的集合。

对查找表的操作：

1. 查询某个“特定的”数据元素是否在查找表中。
2. 检索某个“特定的”数据元素的各种属性。
3. 在查找表中插入一个数据元素。
4. 从查找表中删去某个数据元素。

前两种统称**静态查找表**，包含后两种称为**动态查找表**。

- 关键字(Key)：数据元素或记录中某个数据项的值，用它可以标识一个数据元素。
- 若可唯一标识，称为主关键字(Primary Key)，反之称为次关键字(Second Key)。

静态查找表

顺序查找

```
int Sequential_Search(int *a, int n, int key) {
  int i;
  for (i = 1; i <= n; i++) {
    if (a[i] == key)
      return i;
  }
  return 0;
}
```

设置哨兵进行优化：

```
int Sequential_Search(int *a, int n, int key) {
  int i;
  a[0] = key;
  i = n;
  while (a[i] != key)
    i--;
  return i;
}
```

折半查找

```
int Search_bin(int *a, int n, int k) {
  int low, high, mid;
  low = 1;
  high = n;
  while (low <= high) {
    mid = (low + high) / 2;
    if (key < a[mid])
      high = mid - 1;
    else if (key > a[mid])
      low = mid + 1;
    else
      return mid;
  }
  return 0;
}
```

插值查找

根据要查找的关键字key与查找表中最大最小记录的关键字比较后的查找方法。

适用于关键字均匀分布的表。

修改折半查找中的条件即可。

$mid = \frac{low+high}{2} = low + \frac{1}{2}(high - low)$

斐波那契查找

```
int Fibonacci_Search(int *a, int n, int key) {
  int low, high, mid, i, k;
  low = 1;
  high = n;
  k = 0;
//  计算n位于斐波那契数列的位置
  while (n > F[k] - 1)
    k++;
//  将不满的数值补全
  for (i = n; i < F[k] - 1; i++)
    a[i] = a[n];
  while (low <= high) {
    mid = low + F[k - 1] - 1;
    if (key < a[mid]) {
      high = mid - 1;
      k = k - 1;
    }
    else if (key > a[mid]) {
      low = mid + 1;
      k = k - 2;
    }
    else {
      if (mid <= n)
        return mid;
      else
        return n;
    }
  }
  return 0;
}
```

只进行加减运算。

静态树表查找

如果只考虑查找成功的情况，则使查找性能达最佳的判定树是其带权内路径长度之和PH值[①]

$$PH = \sum_{i=1}^{n} w_i h_i \tag{9-7}$$

取最小值的二叉树。其中：n为二叉树上结点的个数(即有序表的长度)；h_i为第i个结点在二叉树上的层次数；结点的权$w_i = cp_i (i=1,2,\cdots,n)$，其中$p_i$为结点的查找概率，$c$为某个常量。称$PH$值取最小的二叉树为**静态最优查找树**(Static Optimal Search Tree)。

线性索引查找

1. 稠密索引
2. 分块索引
3. 倒排索引

动态查找表

表结构本身是在查找过程中动态生成的，对于给定的key，若查找成功则返回，否则插入关键字等于key的记录。

二叉排序树

二叉排序树要么是空二叉树，要么具有如下特点：

- 二叉排序树中，如果其根结点有左子树，那么左子树上所有结点的值都小于根结点的值；
- 二叉排序树中，如果其根结点有右子树，那么右子树上所有结点的值都大小根结点的值；
- 二叉排序树的左右子树也要求都是二叉排序树；

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

# define OK 1
# define ERROR 0
# define TRUE 1
# define FALSE 0
# define INFEASIBLE -1
# define OVERFLOW -2

typedef int Status;
typedef int Boolean;

typedef struct BiTNode {
  int data;
  struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

//指针f指向T的双亲，初始为NULL
//如果查找失败，p指向查找路径上访问的最后一个结点
```

```
//p是为了查找成功后可以得到查找到的结点位置
Status SearchBST(BiTree T, int key, BiTree f, BiTree *p) {
//  查找不成功
  if (!T) {
    *p = f;
    return FALSE;
  }
  else if (key == T->data) {
    *p = T;
    return TRUE;
  }
  else if (key < T->data)
    return SearchBST(T->lchild, key, T, p);
  else
    return SearchBST(T->rchild, key, T, p);
}

//插入
Status InsertBST(BiTree *T, int key) {
  BiTree p, s;
  if (!SearchBST(*T, key, NULL, &p)) {
    s = (BiTree)malloc(sizeof(BiTNode));
    s->data = key;
    s->lchild = s->rchild = NULL;
    if (!p)
      *T = s;
    else if (key < p->data)
      p->lchild = s;
    else
      p->rchild = s;
    return TRUE;
  }
  else
    return FALSE;
}

//从二叉树中删除结点p，并重接左或右子树
Status Delete(BiTree *p) {
  BiTree q, s;
  if ((*p)->rchild == NULL) {
    q = *p;
    *p = (*p)->lchild;
    free(q);
  }
  else if ((*p)->lchild == NULL) {
    q = *p;
    *p = (*p)->rchild;
    free(q);
  }
```

```
//  左右子树均不空
  else {
    q = *p;
    *p = (*p)->lchild;
    while (s->rchild) {
      q = s;
      s = s->rchild;
    }
    (*p)->data = s->data;
    if (q != *p)
      q->rchild = s->lchild;
    else
      q->lchild = s->lchild;
    free(s);
  }
  return TRUE;
}


//若存在则删除key
Status DeleteBST(BiTree *T, int key) {
  if (!*T)
    return FALSE;
  else {
    if (key == (*T)->data)
      return Delete(T);
    else if (key < (*T)->data)
      return DeleteBST(&(*T)->lchild, key);
    else
      return DeleteBST(&(*T)->rchild, key);
  }
}


int main() {

  return 0;
}
```

对于二叉排序树的查找，走的是从根结点到要查找的结点的路径，其比较次数等于给定值的结点在二叉排序树的层数。

平衡二叉树

平衡二叉树，又称为AVL树。实际上就是遵循以下两个特点的二叉树：

- 每棵子树中的左子树和右子树的深度差不能超过 1；
- 二叉树中每棵子树都要求是平衡二叉树；

```
//定义二叉排序树
typedef struct BSTNode{
    ElemType data;
    int bf;//balance flag
    struct BSTNode *lchild,*rchild;
}*BSTree,BSTNode;
```

$B-$树和B^+树

一颗 m 阶的 B-树，或者本身是空树，或为满足以下特性：

- 树中每个结点至多有 m 棵子树；
- 若根结点不是叶子结点，则至少有两棵子树；
- 除根之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；
- 所有的非终端结点中包含下列信息数据：(n，A0，K1，A1，K2，A2，...，Kn，An)；

  其中：$K_i(i=1,\cdots,n)$为关键字，且$K_i<K_{i+1}(i=1,\cdots,n-1)$；$A_i(i=0,\cdots,n)$为指向子树根结点的指针，且指针$A_{i-1}$所指子树中所有结点的关键字均小于$K_i(i=1,\cdots,n)$，$A_n$所指子树中所有结点的关键字均大于$K_n$，$n(\lceil m/2 \rceil-1\leqslant n\leqslant m-1)$为关键字的个数(或$n+1$为子树个数)。
- 所有的叶子结点都出现在同一层次，并且不带信息。实际上这些结点都不存在，指向这些结点的指针都为 NULL；

一颗 m 阶的 B+树和 m 阶的 B-树的差异在于：

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

键树

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

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

哈希表

概念

根据设定的哈希函数$H(key)$和处理冲突的方法将一组关键字映像到一个有限的连续的地址集(区间)上，并以关键字在地址集中的“像”作为记录在表中的存储位置，这种表便成为**哈希表**，这一映像过程称为哈希造表或**散列**，所得存储位置称**哈希地址**或**散列地址**。

对不同的关键字可能得到同一哈希地址，即$key1 \neq key2$，而$f(key1) = f(key2)$，这种现象称**冲突**(collision)。

若对于关键字集合中的任一关键字，经哈希函数映像到地址集合中任何一个地址的概率是相等的，则称此类哈希函数为**均匀的**(Uniform)哈希函数。

哈希函数的构造方法

- 直接定址法：取关键字或关键字的某个线性函数值为哈希地址。
- 数字分析法：通过对数据的分析，发现数据中冲突较少的部分，并构造散列地址。例如同学们的学号，通常同一届学生的学号，其中前面的部分差别不太大，所以用后面的部分来构造散列地址。
- 平方取中法：当无法确定关键字里哪几位的分布相对比较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为散列地址。这是因为：计算平方之后的中间几位和关键字中的每一位都相关，所以不同的关键字会以较高的概率产生不同的散列地址。
- 取随机数法：使用一个随机函数，取关键字的随机值作为散列地址，这种方式通常用于关键字长度不同的场合。
- 除留取余法：取关键字被某个不大于散列表的表长 n 的数 m 除后所得的余数 p 为散列地址。这种方式也可以在用过其他方法后再使用。该函数对 m 的选择很重要，一般取素数或者直接用 n。

冲突的处理方式

- 开放地址法(也叫开放寻址法)：实际上就是当需要存储值时，对Key哈希之后，发现这个地址已经有值了，这时该怎么办？不能放在这个地址，不然之前的映射会被覆盖。这时对计算出来的地址进行一个探测再哈希，比如往后移动一个地址，如果没人占用，就用这个地址。如果超过最大长度，则可以对总长度取余。这里移动的地址是产生冲突时的增列序量。
- 再哈希法：在产生冲突之后，使用关键字的其他部分继续计算地址，如果还是有冲突，则继续使用其他部分再计算地址。这种方式的缺点是时间增加了。
- 链地址法：链地址法其实就是对Key通过哈希之后落在同一个地址上的值，做一个链表。其实在很多高级语言的实现当中，也是使用这种方式处理冲突的，我们会在后面着重学习这种方式。
- 建立一个公共溢出区：这种方式是建立一个公共溢出区，当地址存在冲突时，把新的地址放在公共溢出区里。

8 内部排序

概念

- **排序**(Sorting)是将一个数据元素或记录的任意序列重新排列成一个按关键字有序的序列。
- 假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种**排序算法是稳定的**。

- 内部排序：指的是待排序记录存放在计算机随机存储器中进行的排序过程。
- 外部排序：指的是待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

插入排序

直接插入排序

直接插入排序(straight insertion sort)是一种最简单的排序方法，它的基本操作是将一个记录插入到已排好序的有序表中，从而得到一个新的、记录数增1的有序表。

```
void InsertSort(SqList *L) {
  int i, j;
  for (i = 2; i <= L->length; i++) {
    // 需要将L->R[i]插入有序子表
    if(L->r[i] < L->r[i - 1]) {
      L->r[0] = L->r[i];  // 复制为哨兵
      for (j = i - 1; L->r[j] > L->r[0]; j--)
//          记录右移
        L->r[j + 1] = L->r[j];
      L->r[j + 1] = L->r[0];
    }
  }
}
```

折半插入排序

查找操作利用折半查找来实现。

2-路插入排序

2-路插入排序是在折半插入排序的基础上再改进之，其目的是减少排序过程中移动记录的次数，但为此需要 n 个记录的辅助空间。具体做法是：另设一个和 L.r 同类型的数组 d，首先将 L.r[1]赋值给 d[1]，并将 d[1]看成是在排好序的序列中处于中间位置的记录，然后从 L.r 中第 2 个记录起依次插入到 d[1]之前或之后的有序序列中。先将待插记录的关键字和 d[1]的关键字进行比较，若 L.r[i].key＜d[1].key，则将 L.r[i]插入到 d[1]之前的有序表中。反之，则将 L.r[i]插入到 d[1]之后的有序表中。在实现算法时，可将 d 看成是一个循环向量，并设两个指针 first 和 final 分别指示排序过程中得到的有序序列中的第一个记录和最后一个记录在 d 中的位置。具体算法留作习题由读者自己写出。

仍以式(10-4)中的关键字为例，进行 2-路插入排序的过程如图 10.2 所示。

在 2-路插入排序中，移动记录的次数约为 $n^2/8$。因此，2-路插入排序只能减少移动记录的次数，而不能绝对避免移动记录。并且，当 L.r[1]是待排序记录中关键字最小或最大的记录时，2-路插入排序就完全失去它的优越性。因此，若希望在排序过程中不移动记录，只有改变存储结构，进行表插入排序。

表插入排序

表插入排序，即使用链表的存储结构对数据进行插入排序。在对记录按照其关键字进行排序的过程中，不需要移动记录的存储位置，只需要更改结点间指针的指向。

```
#define SIZE 100
typedef struct {
    RcdType rc; //记录项
    int next; //指针项
}SLNode;

typedef struct {
    SLNode r[SIZE]; //0号单元为表头结点
    int length;   //链表当前长度
}SLinkListType;
```

希尔排序

希尔排序(Shell's Sort)，又称“缩小增量排序”(Diminishing Increnment Sort)，也是插入排序的一种。

它的基本思想是：先将整个待排记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

希尔排序的一个特点是：子序列的构成不是简单地“逐段分割”,而是将相隔某个“增量”的记录组成一个子序列。

快速排序

起泡排序

Bubble Sort

起泡排序的过程很简单。首先将第一个记录的关键字和第二个记录的关键字进行比较，若为逆序(即 L. r[1]. key>L. r[2]. key)，则将两个记录交换之，然后比较第二个记录和第三个记录的关键字。依次类推，直至第 $n-1$ 个记录和第 n 个记录的关键字进行过比较为止。上述过程称做第一趟起泡排序，其结果使得关键字最大的记录被安置到最后一个记录的位置上。然后进行第二趟起泡排序，对前 $n-1$ 个记录进行同样操作，其结果是使关键字次大的记录被安置到第 $n-1$ 个记录的位置上。一般地，第 i 趟起泡排序是从 L. r[1]到 L. r[n−i+1]依次比较相邻两个记录的关键字，并在“逆序”时交换相邻记录，其结果是这 $n-i+1$ 个记录中关键字最大的记录被交换到第 $n-i+1$ 的位置上。整个排序过程需进行 $k(1\leqslant k<n)$ 趟起泡排序，显然，判别起泡排序结束的条件应该是“在一趟排序过程中没有进行过交换记录的操作”。

```
# define SIZE 10

void swap(int &a, int &b)
{
    int temp = a;
```

```
    a = b;
    b = temp;
}

//n为数组元素个数
void BubbleSort(int *data, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if (data[j] < data[j + 1]) {
        swap(data[j], data[j + 1]);
      }
    }
  }
}

int main() {
  int data[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  BubbleSort(data, SIZE);
  for (int i = 0; i < SIZE; i++)
    printf("%d ", data[i]);
  return 0;
}
```

快速排序

快速排序(Quick Sort)是对起泡排序的一种改进。

基本思想：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

首先从表中选取一个记录的关键字作为分割点(称为“**枢轴**”或者支点，一般选择第一个关键字)

选择排序

选择排序(Selection Sort)的基本思想是：每一趟在$n-i+1(i=1,2,\ldots,n-1)$个记录中选取关键字最小的记录作为有序序列中第i个记录。

简单选择排序

一趟简单选择排序的操作为：通过 $n-i$ 次关键字间的比较，从 $n-i+1$ 个记录中选出关键字最小的记录，并和第 $i(1\leqslant i\leqslant n)$ 个记录交换之。

显然，对 L.r[1..n]中记录进行简单选择排序的算法为：令 i 从 1 至 $n-1$，进行 $n-1$ 趟选择操作，如算法 10.9 所示。容易看出，简单选择排序过程中，所需进行记录移动的操作次数较少，其最小值为“0”，最大值为 $3(n-1)$。然而，无论记录的初始排列如何，所需进行的关键字间的比较次数相同，均为 $n(n-1)/2$。因此，总的时间复杂度也是$O(n^2)$。

```
void SelectSort (SqList &L) {
  // 对顺序表 L 作简单选择排序。
  for (i = 1; i<L.length; ++ i) {          // 选择第 i 小的记录，并交换到位
      j = SelectMinKey(L, i);              // 在 L.r[i..L.length]中选择 key 最小的记录
      if (i!= j) L.r[i]←→L.r[j];           // 与第 i 个记录交换
  }
}// SelectSort
```

树形选择排序

树形选择排序(Tree Selection Sort)，又称锦标赛排序(Tournament Sort)，是一种按照锦标赛的思想进行选择排序的方法。首先对 n 个记录的关键字进行两两比较，然后在其中$\lceil \frac{n}{2} \rceil$个较小者之间再进行两两比较，如此重复，直至选出最小关键字的记录为止。

堆排序

堆的定义如下：n 个元素的序列$\{k_1,k_2,\cdots,k_n\}$当且仅当满足下关系时，称之为堆。

$$\begin{cases} k_i \leqslant k_{2i} \\ k_i \leqslant k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geqslant k_{2i} \\ k_i \geqslant k_{2i+1} \end{cases}$$

$$\left(i=1,2,\cdots,\lfloor \frac{n}{2} \rfloor\right)$$

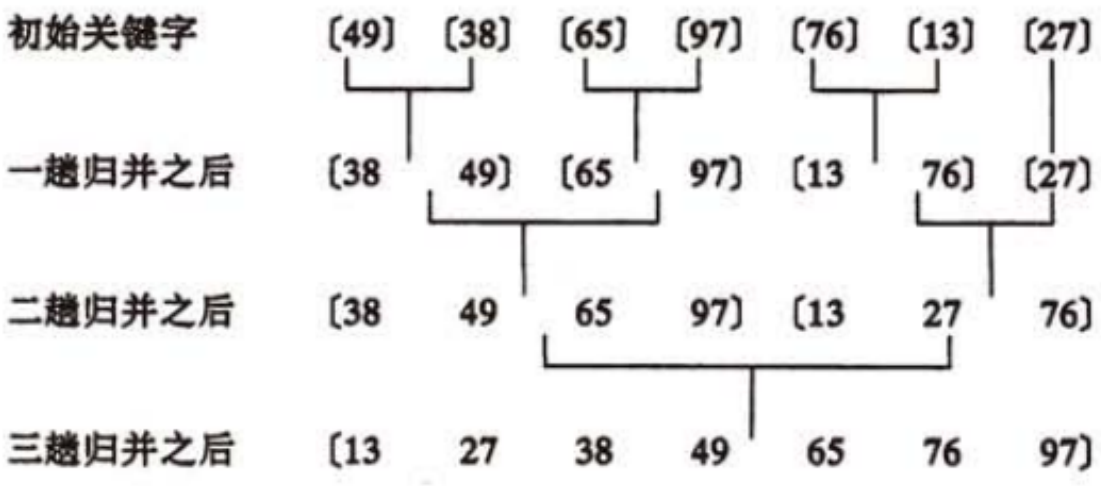
对于堆的定义也可以使用完全二叉树来解释，因为在完全二叉树中第 i 个结点的左孩子恰好是第 2i 个结点，右孩子恰好是 2i+1 个结点。如果该序列可以被称为堆，则使用该序列构建的完全二叉树中，每个根结点的值都必须不小于(或者不大于)左右孩子结点的值。

通过将无序表转化为堆，可以直接找到表中最大值或者最小值，然后将其提取出来，令剩余的记录再重建一个堆，取出次大值或者次小值，如此反复执行就可以得到一个有序序列，此过程为**堆排序**。

归并排序

归并排序(Merging Sort)的实现思想是先将所有的记录完全分开，然后两两合并，在合并的过程中将其排好序，最终能够得到一个完整的有序表。

2-路归并排序：



基数排序

基数排序(Radix Sorting)不同于之前所介绍的各类排序，前边介绍到的排序方法是通过比较和移动记录来实现排序，而基数排序的实现不需要进行对关键字的比较。

基数排序是借助“分配”和“收集”两种操作对单逻辑关键字进行排序的一种内部排序方法。

多关键字的排序

一般情况下，假设有 n 个记录的序列

$$\{R_1, R_2, \cdots, R_n\} \tag{10-10}$$

且每个记录 R_i 中含有 d 个关键字 $(K_i^0, K_i^1, \cdots, K_i^{d-1})$，则称序列(10-10)对关键字 $(K^0, K^1, \cdots, K^{d-1})$ 有序是指：对于序列中任意两个记录 R_i 和 R_j $(1 \leqslant i < j \leqslant n)$ 都满足下列有序关系[①]：

$$(K_i^0, K_i^1, \cdots, K_i^{d-1}) < (K_j^0, K_j^1, \cdots, K_j^{d-1})$$

其中 K^0 称为最主位关键字，K^{d-1} 称为最次位关键字。为实现多关键字排序，通常有两种方法：第一种方法是：先对最主位关键字 K^0 进行排序，将序列分成若干子序列，每个子序列中的记录都具有相同的 K^0 值，然后分别就每个子序列对关键字 K^1 进行排序，按 K^1 值不同再分成若干更小的子序列，依次重复，直至对 K^{d-2} 进行排序之后得到的每一子序列中的记录都具有相同的关键字 $(K^0, K^1, \cdots, K^{d-2})$，而后分别每个子序列对 K^{d-1} 进行排序，最后将所有子序列依次联接在一起成为一个有序序列，这种方法称之为**最高位优先**(Most Significant Digit first)法，简称 MSD 法；第二种方法是从最次位关键字 K^{d-1} 起进行排序。然后再对高一位的关键字 K^{d-2} 进行排序，依次重复，直至对 K^0 进行排序后便成为一个有序序列。这种方法称之为**最低位优先**(Least Significant Digit first)法，简称 LSD 法。

MSD 和 LSD 只约定按什么样的“关键字次序”来进行排序，而未规定对每个关键字进行排序时所用的方法。但从上面所述可以看出这两种排序方法的不同特点：若按 MSD 进行排序，必须将序列逐层分割成若干子序列，然后对各子序列分别进行排序；而按 LSD 进行排序时，不必分成子序列，对每个关键字都是整个序列参加排序，但对 K^i $(0 \leqslant i \leqslant d-2)$ 进行排序时，只能用稳定的排序方法。另一方面，按 LSD 进行排序时，在一定的条件下(即对前一个关键字 K^i $(0 \leqslant i \leqslant d-2)$ 的不同值，后一个关键字 K^{i+1} 均取相同值)，也可以不利用前几节所述各种通过关键字间的比较来实现排序的方法，而是通过若干次“分配”和“收集”来实现排序，如上述第二种整理扑克牌的方法那样。

---

① $(a^0, a^1, \cdots, a^{d-1}) < (b^0, b^1, \cdots, b^{d-1})$ 是指必定存在 l，使得：当 $s=0, \cdots, l-1$ 时，$a^s = b^s$，而 $a^l < b^l$。

链式基数排序

使用链表存储数据，通过调整指针实现LSD(低关键字优先)算法进行排序。

方法比较

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n)\sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)\sim O(n)$	不稳定

9 外部排序

概念

外部排序指的是大文件的排序，即待排序的记录存储在外存储器上，在排序的过程中需进行多次的内、外存之间的交换。

1. 按可用内存大小，将外存上含n个记录的文件分成若干长度为l的子文件或**段**(segment)，依次读入内存，使用适当的内部排序算法对其进行排序，排好序的子文件统称为**归并段**或者**顺串**(run)，将排好序的归并段重新写入外存。
2. 对这些归并段进行逐趟归并，使得归并段(有序的子文件)逐渐由小至大，直到得到整个有序文件为止。

可见，对同一文件而言，进行外排时所需读/写外存的次数和归并的趟数s成正比。
一般情况下，对m个初始归并段进行**k-平衡归并**时，归并的趟数
$s = \lfloor \log_k m \rfloor$
可见，若增加k或减少m便能减少s。

多路平衡归并的实现

如果毫无限度地增加k值，虽然会减少读写外存数据的次数，但会增加内部归并的时间。

为了避免在增加k值的过程中影响内部归并的效率，在进行 k-路归并时可以使用“**败者树**”(Tree of Loser)来实现，该方法在增加 k 值时不会影响其内部归并的效率。

败者树是树形选择排序的一种变形，本身是一棵完全二叉树。

置换-选择排序

归并的趟数不仅和k成反比，也和m成正比，因此，减少m是减少s的另一种途径。
置换-选择排序(Replacement-Selection Sorting)是在树形选择排序的基础上得来的，它的特点是：在整个排序(得到所有初始归并段)的过程中，选择最小(或最大)关键字和输入、输出交叉或平行进行。

最佳归并树

无论是通过等分还是置换-选择排序得到的归并段，如何设置它们的归并顺序，可以使得对外存的访问次数降到最低。

通过以构建赫夫曼树的方式构建归并树，使其对读写外存的次数降至最低(k-路平衡归并，需要选取合适的k值，构建赫夫曼树作为归并树)。所以称此归并树为**最佳归并树**。

参考文献与链接：

- 程杰. 大话数据结构[M]. 清华大学出版社, 2011.
- 严蔚敏, 吴伟民. 数据结构: C 语言版[M]. 清华大学出版社, 2018.
- 高一凡. 数据结构算法解析[M]. 清华大学出版社, 2015.
- https://blog.csdn.net/panglinzhuo/article/details/79397277
- http://data.biancheng.net
- https://www.cnblogs.com/wkfvawl/p/10066666.html
- https://www.bilibili.com/video/BV1Eb41177d1