数据结构

1. **介绍最小生成树算法（Prim、Kruskal）**

   $Prim$算法：每次总是选出一个**离生成树距离最小的点**去加入生成树，最后实现最小生成树

   $Kruskal$算法：先将**边从小到大排列**，从小到大的添加**不构成「环路」**的边

   详解：

   $prim$算法

   - **MST-Prim-PriQueue(G)**

```
输入: 图G =< V, E, W >
输出: 最小生成树T
新建一维数组color[1..|V|], dist[1..|V|], pred[1..|V|]
新建空优先队列Q
//初始化
for u ∈ V do
    color[u] ← WHITE
    dist[u] ← ∞
    pred[u] ← NULL
end
dist[1] ← 0
Q.Insert(V, dist)
```

   O(|V|)

   - **MST-Prim-PriQueue(G)**

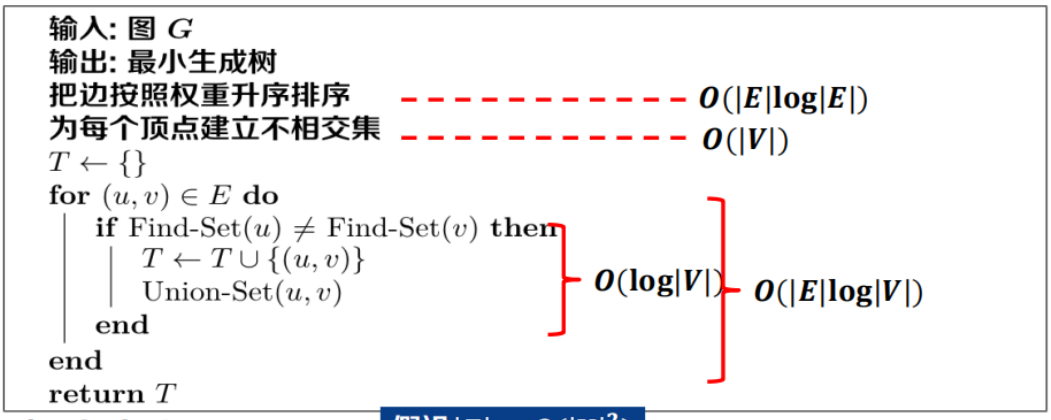
```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v, u) < dist[u] then
            dist[u] ← w(v, u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

   O(|E| · log|V|)

   使用优先队列，时间复杂度为$O(|E|log|V|)$

$kruskal$算法

- **MST-Kruskal(G)**



- **时间复杂度**

假设$|E| = O(|V|^2)$

  - $O(|E|\log|E| + |E|\log|V|) = O(|E|\log|V|^2 + |E|\log|V|) = O(|E|\log|V|)$

使用并查集，时间复杂度为$O(|E|log|V|)$

- **Prim算法和Kruskal算法比较**

	Prim算法	Kruskal算法
核心思想	保持一棵树，不断扩展	子树森林，合并为一棵树
数据结构	优先队列	不相交集合
求解视角	微观视角，基于当前点选边	宏观视角，基于全局顺序选边
算法策略	都是采用贪心策略的图算法	

2. **介绍图的最短路径算法（Dijkstra、Floyd）**
   - Dijkstra算法：

每次取出未访问结点中距离源点最小的，用该结点更新其他结点的距离（松弛操作）。

思想：贪心算法

   - Floyd算法：

     首先初始化，所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。

然后使用三层循环，分别表示从i号顶点到j号顶点只经过前k个点的最短路径。

（循环从外到里是：k, i, j）

思想：动态规划，滚动数组

详解：

$Dijkstra$算法

- **Dijkstra-PriQueue(G, s)**

```
输入: 图G =< V, E, W >, 源点s
输出: 单源最短路径P
新建一维数组color[1..|V|], dist[1..|V|], pred[1..|V|]
新建空优先队列Q
//初始化
for u ∈ V do
    color[u] ← WHITE
    dist[u] ← ∞
    pred[u] ← NULL
end
dist[s] ← 0
Q.Insert(V, dist)
```

O(|V|)

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v, u) < dist[u] then
            dist[u] ← dist[v] + w(v, u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

时间复杂度O(|E| · log|V|)

使用优先队列，时间复杂度为$O(|E|log|V|)$

$Floyd$算法：

- **All-Pairs-Shortest-Paths(G)**

```
输入: 图G =< V, E, W >
输出: 任意两点最短路径
新建二维数组D[1..|V|, 1..|V|], Rec[1..|V|, 1..|V|]
for i ← 1 to |V| do
    for j ← 1 to |V| do
        Rec[i, j] ← 0
        if i = j then
            D[i, j] ← 0
        end
        else
            D[i, j] ← W[i, j]
        end
    end
end
```

初始化

- **All-Pairs-Shortest-Paths(G)**

```
for k ← 1 to |V| do
    for i ← 1 to |V| do
        for j ← 1 to |V| do
            if D[i, j] > D[i, k] + D[k, j] then
                D[i, j] ← D[i, k] + D[k, j]
                Rec[i, j] ← k
            end
        end
    end
end
return D, Rec
```

松弛操作

算法小结

- **直观思路：使用Dijkstra算法依次求解所有点**

```
输入: 图G
输出: 任意两点最短路径
for i ← 1 to |V| do
    Paths[i] ← Dijkstra − PriQueue(G, i)
end
return Paths
```

$O(|E|\log|V|)$ → $O(|V||E|\log|V|)$

针对稠密图 $|E| = O(|V|^2)$

$O(|V|^3\log|V|)$

- **Floyd-Warshall算法时间复杂度：$O(|V|^3)$**

3. **什么是稳定排序，哪些排序是稳定排序**

定义：能保证两个相等的数，经过排序之后，其在序列的前后位置顺序不变。

（A1=A2，排序前A1在A2前面，排序后A1还在A2前面）

意义：**稳定性本质是维持具有相同属性的数据的插入顺序**

稳定排序有：插入排序、冒泡排序、归并排序、基数排序、桶排序、计数排序

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

基本思想参考：https://www.cnblogs.com/onepixel/articles/7674659.html

4. **C 语言中数组名和数组指针的关系**

数组名代表了一个指向数组首元素的常量指针，其类型与数组元素类型相同。

数组指针指向数组的变量指针，其类型在定义时确定。

指向数组首元素的指针和指向整个数组的数组指针值相同，但类型不同。

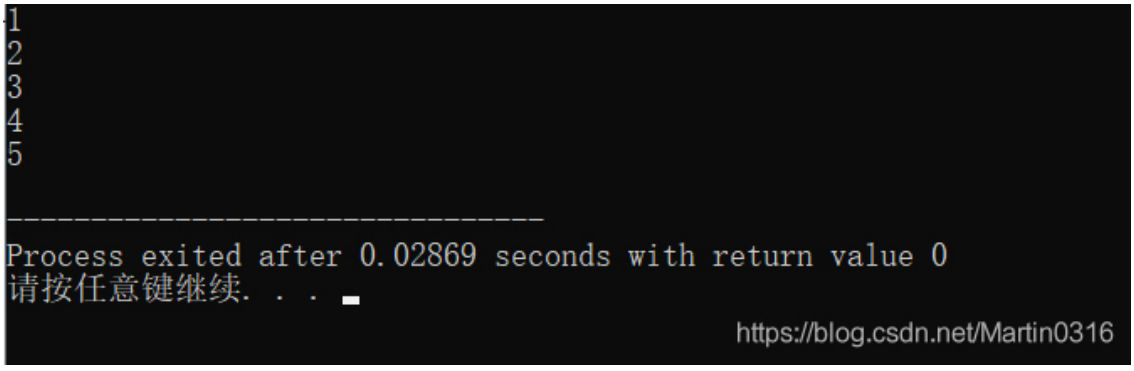
当p为int型指针，p++移动4个字节；当p为int (*p)[5]型指针时，p++移动20个字节

```
#include <stdio.h>
int main()
{
    int temp[5] = {1, 2, 3, 4, 5};
    int (*p)[5] = &temp;
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("%d\n", *(*p + i));
     }
     return 0;
}
```

这是因为虽然temp和&temp的值都是相同的，但它们的意义却是不相同的：

temp指的是这个数组的 **第一个元素** 的首地址
&temp 指的是这 **整个数组** 的首地址




```
#include <iostream>

using namespace std;

int main() {
    int temp[5] = {1, 2, 3, 4, 5};
    int (*p)[5] = &temp;
    int i;
    cout << "p = " << p << " temp = " << temp << endl;
    cout << "p + 1 = " << p + 1 << " temp + 1 = " << temp
+ 1 << endl;
```

```
    return 0;
}

// 输出可以看到p 和 temp 值是相同的
// p + 1和 temp + 1则不同
console:
p = 0x2bec3ffb80 temp = 0x2bec3ffb80
p + 1 = 0x2bec3ffb94 temp + 1 = 0x2bec3ffb84
```

5. **堆、栈和队列的定义与区别**

   - **堆**

     堆通常是一个可以被看做一棵树的数组对象。堆总是满足下列性质：

     - 堆中某个**结点的值总是不大于或不小于其父结点的值**；
     - 堆总是一棵**完全二叉树。**

     将根结点最大的堆叫做最大堆或大根堆，根结点最小的堆叫做最小堆或小根堆。

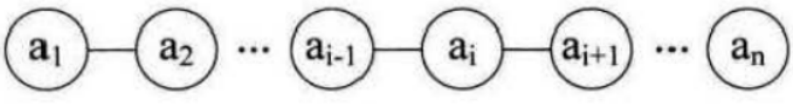
   - **栈**

     - **只允许在一端进行插入或删除操作的线性表。**
     - 栈顶（Top）：线性表允许进行插入和删除的一端。
     - 栈底（Bottom）：固定的，不允许进行插入和删除的另一端。
     - 栈的特点是：后进先出

     (附注：线性表的定义)

     线性表（List）：零个或多个数据元素的有限序列。

     线性表的数据集合为{a1,a2,…,an}，假设每个元素的类型均为DataType。其中，除第一个元素a1外，每一个元素有且只有一个直接前驱元素，除了最后一个元素an外，每一个元素有且只有一个直接后继元素。数据元素之间的关系是一对一的关系。

   - **队列**

     - 队列是一种操作受限的线性表，队列只允许在表的一端进行插入，在表的另一端进行删除。可进行插入的一段称为队尾，可进行删除的一端称为队头。

- 队列的主要特点就是**先进先出**。依照存储结构可分为：顺序队和链式队。

堆、栈、队列之间的区别是?

①堆是在程序运行时，而不是在程序编译时，申请某个大小的内存空间。即动态分配内存，对其访问和对一般内存的访问没有区别。

②栈就是一个桶，后放进去的先拿出来，它下面本来有的东西要等它出来之后才能出来。（后进先出）

③队列只能在队首做删除操作,在队尾做插入操作，而栈只能在栈顶做插入和删除操作。（先进先出）

6. **数组和链表的优缺点**
   - 数组的优点
     - 随机访问性强
     - 查找速度快
   - 数组的缺点
     - 插入和删除效率低
     - 可能浪费内存
     - 内存空间要求高，必须有足够的连续内存空间
     - 数组大小固定，不能动态拓展
   - 链表的优点
     - 插入删除速度快
     - 内存利用率高，不会浪费内存
     - 大小没有固定，拓展灵活
   - 链表的缺点
     - 不能随机查找，必须从第一个开始遍历，查找效率低

	数组	链表
优点	随机访问性强 查找速度快	插入删除速度快 内存利用率高，不会浪费内存 大小没有固定，拓展灵活
缺点	插入和删除效率低 可能浪费内存 内存空间要求高，必须有足够的连续内存空间 大小固定，不能动态拓展	不能随机查找，必须从第一个开始遍历，查找效率低

7. **C 语言指针和数组实际的类型**

   指针的类型：

   - int*ptr;//指针的类型是int*
   - char*ptr;//指针的类型是char*
   - int**ptr;//指针的类型是int**
   - int(*ptr)[3];//指针的类型是int(*)[3]，指针所指向的的类型是int()[3]
   - int*(*ptr)[4];//指针的类型是int*(*)[4]，指针指向的类型是int*()[4]

     指向Array[4]数组的指针，数组元素的类型是 int*

     等价于：

     ```
     typedef int* PINT;
     typedef PINT PINT4[4]; //四个int*指针
     typedef PINT4* PTR_TYPE;//指向四个int*指针
     PTR_TYPE ptr;
     ```

8. **B 树和 B+树的定义**

   - B树

   B 树，又称多路平衡查找树， B 树中所有结点的孩子个数的最大值称为 B 树的阶，通常用m

   表示。一棵m 阶B 树或为空树，或为满足如下特性的m 叉树：

   1. 树中每个结点至多有m棵子树，即至多含有$m-1$个关键字。

2. 若根结点不是终端结点，则至少有两棵子树。
3. 除根结点外的所有非叶结点至少有$\lceil m/2 \rceil$ 棵子树，即至少含有$\lceil m/2 \rceil - 1$个关键字。
4. 所有的叶结点都出现在同一层次上，并且不带信息（可以视为外部结点或类似千折半查找判定

树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。

B 树是所有结点的平衡因子均等于0 的多路平衡查找树。

- B+树

2.B+树是应数据库所需而出现的一种B 树的变形树。

一棵m 阶的B+树需满足下列条件：
1) 每个分支结点最多有m 棵子树（孩子结点）。
2) 非叶根结点至少有两棵子树，其他每个分支结点至少有「m/2]棵子树。
3) 结点的子树个数与关键字个数相等。
4) 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。
5) 所有分支结点（可视为索引的索引）中仅包含它的各个子结点（即下一级的索引块）中关键字的最大值及指向其子结点的指针。

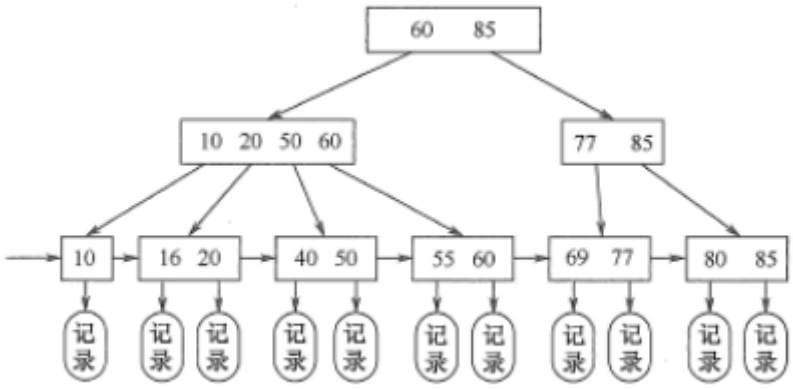


图 7.8　B+树结构示意图

m 阶的B+树与m 阶的B 树的主要差异如下:

1) 在B+树中，具有n 个关键字的结点只含有n 棵子树，即每个关键字对应一棵子树；而在B 树中，具有n 个关键字的结点含有n+1棵子树。

2) 在B+树中，每个结点（非根内部结点）的关键字个数n 的范围是「m/2]<=n<= m (根结点：1<=n<=m); 在B 树中，每个结点（非根内部结点）的关键字个数n 的范围是「m/2]-1<=n<= m-1 。

3) 在B+树中，叶结点包含信息，所有非叶结点仅起索引作用，非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。

4) 在B+树中，叶结点包含了全部关键字，即在非叶结点中出现的关键字也会出现在叶结点中；而在B 树中，叶结点包含的关键字和其他结点包含的关键字是不重复的。

9. 前序、中序、后序遍历的定义

   前序遍历：前序遍历首先访问根结点然后遍历左子树，最后遍历右子树。

   中序遍历：中序遍历首先遍历左子树，然后访问根节点，最后遍历右子树。

   后序遍历：后序遍历首先遍历左子树，然后遍历右子树，最后访问根节点。

10. 介绍二路归并算法

    二路归并排序是用**分治**思想，分治模式在每一层递归上有三个步骤：

    - **分解（Divide）**：将n个元素分成个含n/2个元素的子序列。
    - **解决（Conquer）**：用合并排序法对两个子序列递归的排序。
    - **合并（Combine）**：合并两个已排序的子序列已得到排序结果。

11. 介绍哈夫曼树

1．哈夫曼树的定义

在许多应用中，树中结点常常被赋予一个表示某种意义的数值，称为该结点的权。从树的根到任意结点的路径长度（经过的边数）与该结点上权值的乘积，称为该结点的带权路径长度。树中所有叶结点的带权路径长度之和称为该树的带权路径长度，记为

$$\mathrm{WPL}=\sum_{i=1}^{n} w_i l_i$$

式中，w_i是第 i 个叶结点所带的权值，l_i是该叶结点到根结点的路径长度。

在含有 n 个带权叶结点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为哈夫曼树，也称最优二叉树。例如，图 5.34 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

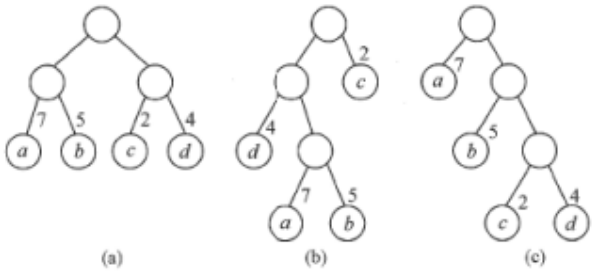


图 5.34　具有不同带权长度的二叉树

(a) WPL = 7×2 + 5×2 + 2×2 + 4×2 = 36。

(b) WPL = 4×2 + 7×3 + 5×3 + 2×1 = 46。

(c) WPL = 7×1 + 5×2 + 2×3 + 4×3 = 35。

其中，图 5.34(c)树的 WPL 最小。可以验证，它恰好为哈夫曼树。

2.哈夫曼树的构造

给定n 个权值分别为W1,W2…， Wn 的结点，构造哈夫曼树的算法描述如下:
1) 将这n 个结点分别作为n 棵仅含一个结点的二叉树，构成森林F 。
2) 构造一个新结点，从F 中选取两棵根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根结点的权值之和。
3) 从F 中删除刚才选出的两棵树，同时将新得到的树加入F 中。
4) 重复步骤2) 和3)'直至F 中只剩下一棵树为止。

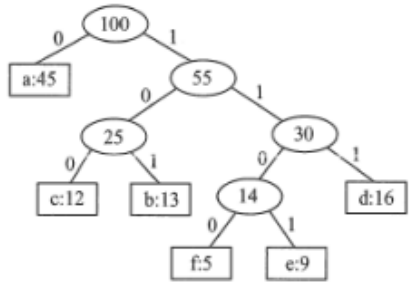
从上述构造过程中可以看出哈夫曼树具有如下特点:
1) 每个初始结点最终都成为叶结点，且权值越小的结点到根结点的路径长度越大。
2) 构造过程中共新建了n — 1 个结点（双分支结点），因此哈夫曼树的结点总数为2n -1 。
3) 每次构造都选择2 棵树作为新结点的孩子，因此哈夫曼树中不存在度为1 的结点。

3.哈夫曼编码:

在数据通信中，若对每个字符用相等长度的二进制位表示，称这种编码方式为固定长度编码。若允许对不同字符用不等长的二进制位表示，则这种编码方式称为可变长度编码。可变长度编码比固定长度编码要好得多，其特点是对频率高的字符赋以短编码，而对频率较低的字符则赋以较长一些的编码，从而可以使字符的平均编码长度减短，起到压缩数据的效果。哈夫曼编码是一种被广泛应用而且非常有效的数据压缩编码。若没有一个编码是另一个编码的前缀，则称这样的编码为**前缀编码**。

由哈夫曼树得到哈夫曼编码是很自然的过程。首先，将每个出现的字符当作一个独立的结点，其权值为它出现的频度（或次数），构造出对应的哈夫曼树。显然，所有字符结点都出现在叶结点中。我们可将字符的编码解释为从根至该字符的路径上边标记的序列，其中边标记为0 表示"转向左孩子"，标记为1 表示"转向右孩子"。

各字符编码为
a:0
b:101
c:100
d:111
e:1101
f:1100



12. 什么是 hash，hash 遇到碰撞怎么办

32.哈希表的概念、哈希函数的构造方法、哈希冲突的解决办法?

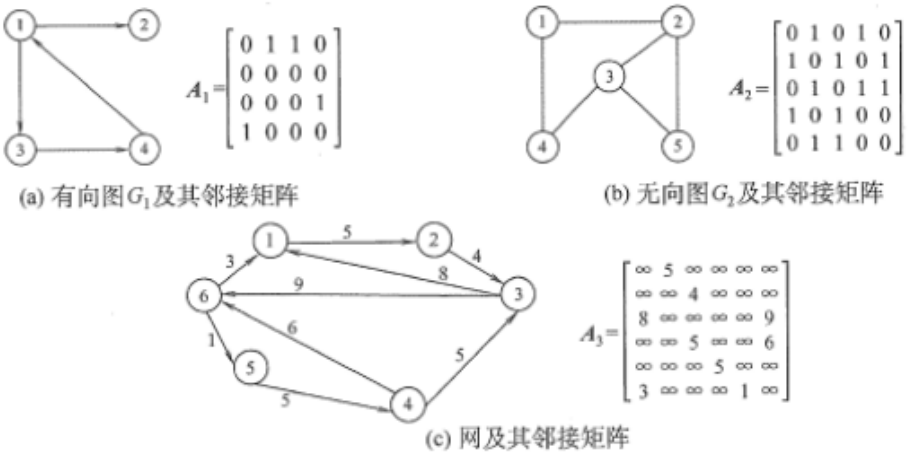
> 哈希表又称为散列表，是根据关键字码的值直接进行访问的数据结构，即它通过把关键码的值映射到表中的一个位置以加快查找速度，其中映射函数叫做散列函数，存放记录的数组叫做散列表。
>
> 哈希函数的构造方法包括：直接定址法，除留余数法，数字分析法，平方取中法，折叠法，随机数法
>
> （1）直接定址法：取关键字的某个线性函数值作为散列地址，H(key)=a*key+b。
>
> （2）除留余数法：取关键字对p取余的值作为散列地址，其中p<m,即H(key)=key%p (p<m)。
>
> （3）数字分析法：当关键字的位数大于地址的位数，对关键字的各位分布进行分析，选出分布均匀的任意几位作为散列的地址，适用于所有关键字都已知的情况。
>
> （4）平方取中法：对关键字求平方，再取结果中的中间几位作为散列地址。
>
> （5）折叠法：将关键字分为位数相同的几部分，然后取这几部分的叠加和作为散列地址。适用于关键字位数较多，且关键字中每一位上数字分布大致均匀。
>
> （6）随机数法：选择一个随机函数，把关键字的随机函数值作为散列地址。适合于关键字的长度不相同时。
>
> 哈希冲突的解决方法包括：开放定址法和拉链法，当冲突发生时，使用某种探测技术形成一个探测序列，然后沿此序列逐个单单元查找，直到找到该关键字或者碰到一个开放的地址为止，探测到开放的地址表明该表中没有此关键字，若要插入，则探测到开放地址时可将新节点插入该地址单元。其中开放定址法包括：线性探查法，二次探查法，双重散列法
>
> （1）线性探查法：基本思想，探查时从地址d开始，首先探查T[d],在探查T[d+1]...直到查到T[m-1]，此后循环到T[0],T[1]...直到探测到T[d-1]为止。

> （2）二次探查法：基本思想，探查时从地址d开始，首先探查T[d],再探查T[d+1^2],T[d+2^2]...等，直到探查到有空余地址或者探查到T[d-1]为止，缺点是无法探查到整个散列空间。
>
> （3）双重散列法：基本思想，使用两个散列函数来确定地址，探查时从地址d开始，首先探查T[d],再探查T[d+h1(d)],T[d+2*h1(d)]...
>
> 链接法：将所有关键字为同义词的节点链接在同一个单链表中，若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组，凡是散列地址为i的节点均插入到头指针为i的单链表中。

13. 数据结构中存储图的方法

27.图的存储结构:

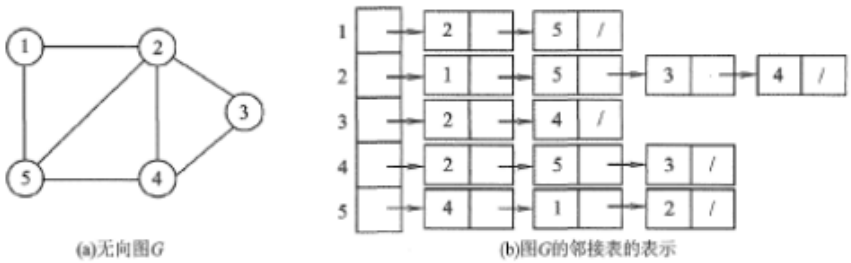
> **1.邻接矩阵法:**
>
> 所谓邻接矩阵存储，是指用一个一维数组存储图中顶点的信息，用一个二维数组存储图中边的信息（即各顶点之间的邻接关系），存储顶点之间邻接关系的二维数组称为邻接矩阵。有向图、无向图和网对应的邻接矩阵实例图如下：



(a) 有向图G_1及其邻接矩阵　　(b) 无向图G_2及其邻接矩阵

(c) 网及其邻接矩阵

> 适合稠密图。
>
> **2.邻接表法:**
>
> 当一个图为稀疏图时，使用邻接矩阵法显然要浪费大量的存储空间，而图的邻接表法结合了顺序存储和链式存储方法，大大减少了这种不必要的浪费。所谓邻接表，是指对图G 中的每个顶点V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, 的边（对于有向图则是以顶点v, 为尾的弧），这个单链表就称为顶点vi 的边表（对于有向图则称为出边表）。边表的头指针和顶点的数据信息采用顺序存储（称为顶点表），所以在邻接表中存在两种结点：顶点表结点和边表结点。



(a)无向图G　　(b)图G的邻接表的表示

3.十字链表法：

十字链表法是有向图的一种链式存储结构。在十字链表中，对应于有向图中的每条弧有一个结点，对应于每个顶点也有一个结点。

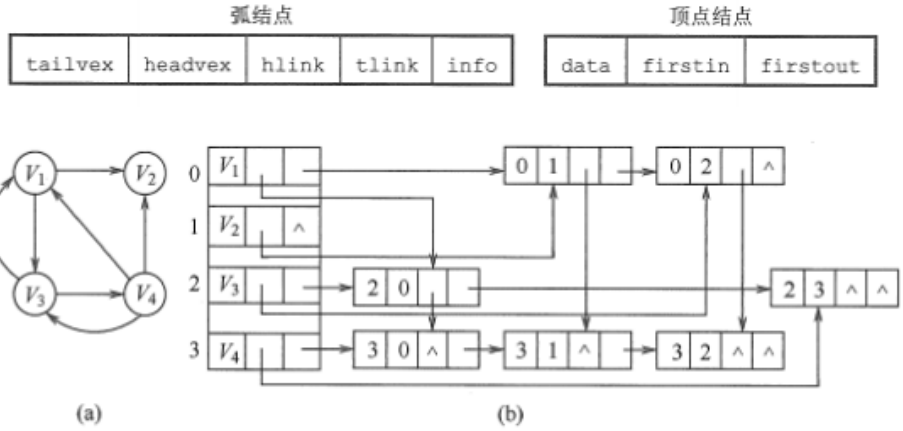
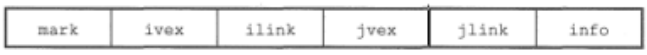


图 6.9　有向图的十字链表表示

4.邻接多重表：

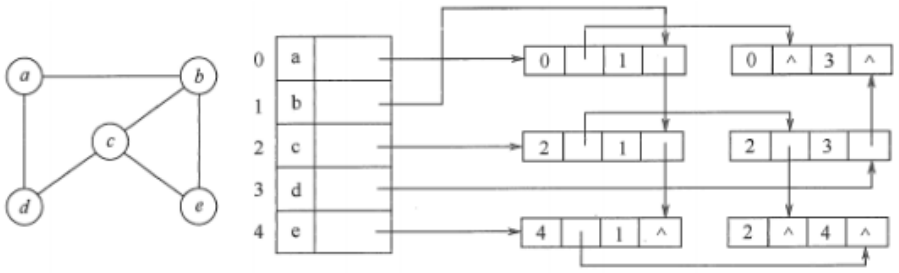
邻接多重表是无向图的另一种链式存储结构。
在邻接表中，容易求得顶点和边的各种信息，但在邻接表中求两个顶点之间是否存在边而对边执行删除等操作时，需要分别在两个顶点的边表中遍历，效率较低。与十字链表类似，在邻接多重表中，每条边用一个结点表示，每个顶点也用一个结点表示。



其中，mark 为标志域，可用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 为指向和边相关的各种信息的指针域。

每个顶点也用一个结点表示，它由如下所示的两个域组成。

其中，data 域存储该顶点的相关信息，firstedge 域指示第一条依附于该顶点的边。



14. C 语言如何写一个变长参数的函数

我们这里想写一个函数 `sum`，对函数参数进行求和：

```
int sum1 = sum(1);    // sum1 should be 0
int sum2 = sum(1, 2); // sum2 should be 2
int sum3 = sum(2, 2, 3); // sum3 should be 2 + 3 = 5
int sum4 = sum(2, 2, 3, 4); // sum4 should be 2 + 3 = 5
int sum5 = sum(3, 2, 3, 4); // sum5 should be 2 + 3 + 4 = 9
```

> 我们这里定义第一个参数为求和数的数量，事实上第一个参数不能缺少，但必须存在，后面会揭示原因.

我们通过头文件 `<stdarg.h>` 和带省略号的函数参数来实现上面的需求：

```
#include <stdarg.h>
int sum(...) { ... }
```

具体步骤如下：

1. 定义一个函数，最后一个参数为省略号，省略号前是可以设置自定义参数的.
2. 在函数定义中创建一个 va_list 类型变量，该类型是在 stdarg.h 头文件中定义的.
3. 使用 int 参数和 va_start 宏来初始化 va_list 变量为一个参数列表。宏 va_start 是在 stdarg.h 头文件中定义的.
4. 使用 va_arg 宏和 va_list 变量来访问参数列表中的每个项.
5. 使用宏 va_end 来清理赋予 va_list 变量的内存.

```
#include <stdio.h>
#include <stdargs.h>
double sum(int num, ...) {
    // step 1
    va_list valist;
    // step 2
    double ret = 0.0;
    int i = 0;
    va_start(valist, num); // step 3
    for (int i = 0; i < num; i++) {
        ret += va_arg(valist, double); // step 4
    }
    va_end(valist); // step 5
    return ret;
}

int main() {
    printf("Sum of 2, 3 is %f\n", sum(2, 2, 3));
    printf("Sum of 2, 3, 4, 5 is %f\n", sum(4, 2, 3, 4,
5));
```

```
}
```

15. 完全二叉树与满二叉树的区别

完全二叉树：若设二叉树的高度为h，除第 h 层外，其它各层$(1\sim h-1)$的结点数都达到**最大个数**，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。

对于k层的完全二叉树，节点数的范围

$2^{k-1}-1<N<2^k-1$

满二叉树：除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树。也就是说，如果一个二叉树的层数为k，且结点总数是2^k-1，则它就是满二叉树。