数据结构

第一章 绪论

数据的逻辑结构是独立于存储结构的，而存储结构是逻辑结构在计算机上的映射。
时间复杂度排序：
O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)
时间复杂度计算方法：
while类：

1. 找关键语句
2. 遇到while设t
3. 解出t

举例：

```
i = 0, sum = 0;
while (sum < n ){
  sum += ++i;
}
```

设语句执行了t次

	1	2	3	…	t
i	1	2	3	…	t
sum	1	3	6	…	t(t+1)/2

t(t+1)/2 < n 可以推出 t< $\sqrt{n}$.
for类：

1. 找关键语句（最内层循环）
2. 遇到for求和Σ
3. 计算

举例

```
for(int i = 1; i < n; i ++)
  for(inr j = 0; j < i; j ++)
    sum++;
```

结果为 $\sum_{i=1}^{n}\sum_{j=0}^{i-1}1$.

第二章 线性表

顺序表插入删除查找时间复杂度都是O（n），但按序号查找只需O（1）
顺序表中的元素类型相同，连续存放，可以随机存取。而一维数组中的元素可以不连续存放。
线性表元素的序号从1开始，在第n+1个位置相当于在尾部添加。
例题：
将顺序表中保存的序列循环左移p个位置：
思路：相当于把ab转为ba，可以采用reverse(reverse(a)reverse(b))解决。

必要时可以用空间换取时间来降低时间复杂度。
对某一结点进行前插可以转化为后插然后交换两个结点中的元素数据。
删除结点也是同理，可以删除后继，把后继的数据填到待删除的结点。
链式存储设计时，结点内的存储单元地址一定连续。
带头结点的循环单链表L，判断该表为空表的条件是：头结点的指针域与L的值相等。
链表逆置可以用头插法遍历整个链表。
找出两个单链表的第一个公共结点的思想是每条链表遍历到最后，如果指针重合则有公共结点。然后是遍历两个链表并记录长度，长度的差值为k，则让长的那个链表先走k步，再一起走，如果两个指针相等，则找到了第一个公共结点。

链表原地逆置代码：

```
Linklist Reverse(Linklist L){
  Lnode *p,*r;
  p = L->next;
  L->next = NULL;
  while(p!=NULL){
    r = p->next;
    p->next = L->next;
    L->next = p;
    p = r;
  }
  return L;
}
```

判断单链表中是否有环的方法是设置**快慢指针**，慢指针每次前移一个，快指针每次前移两个，最终若相遇，则有环，否则没有环（**相遇点是环的起点**）。
求倒数第k个位置的结点可以用**先后指针**的方法，第一个指针遍历k个结点后第二个结点再出发。

第三章 栈、队列和数组

栈的数学性质：当n个不同元素进栈时，出栈元素不同排列的个数为 $\frac{1}{n+1}C_{2n}^{n}$,这个公式成为卡特兰公式。
n个位置的循环队列，front指向队首元素的前一个位置，rear指向队尾元素，则队空标志是front == rear，队满标志是(rear + 1) % n == front
与顺序队列相比，链式队列缺点是不能根据对手指针和队尾指针计算队列的长度。
【例题】将一个10x10阶对称矩阵M的上三角部分的元素 $m_{i,j}$ （1 ≤ i ≤ j ≤ 10)按列优先存入C语言的一维数组N中，元素 $m_{7,2}$ 在N中的下标是（）
答案：22.
设计一个栈，使它可以在O(1)时间内实现找到最小值的操作。
思路：使用双栈，一个记录所有入栈元素，一个记录最小元素，第二个栈的栈顶元素永远是最小的，每次栈一入栈时，栈二比较栈顶元素与入栈元素，如果入栈元素更小，就进入栈二。

第四章 串

串的模式匹配算法——KMP算法
最浅显易懂的 KMP 算法讲解_哔哩哔哩_bilibili

```python
def kmp_search(string, patt):
    next = build_next(patt)  # 假设我们已经算出了 next 数组（马上讲到）

    i = 0  # 主串中的指针
    j = 0  # 子串中的指针
    while i < len(string):
        if string[i] == patt[j]:  # 字符匹配，指针后移
            i += 1
            j += 1
        elif j > 0:  # 字符失配，根据 next 跳过子串前面的一些字符
            j = next[j - 1]
        else:  # 子串第一个字符就失配
            i += 1

        if j == len(patt):  # 匹配成功
            return i - j
```

next数组是已经看过的字符串的最长前后缀数量。
比如

S	a	b	c	a	c
PM	0	0	0	1	0
next(伪)	–1	0	0	0	1
next	0	1	1	1	2

右移位数 = 已匹配的字符数 – 对应的部分匹配值
Move = (j–1)–PM[j–1]
KMP算法的时间复杂度是O(m+n)
进一步优化：
当p P_j = $P_{next[j]}$ 时，计算nextval[j] = next[next[j]]

主串	a	a	a	b	a
模式串	a	a	a	a	b
j	1	2	3	4	5
next[j]	0	1	2	3	4
nextval[j]	0	0	0	0	4

next数组是针对模式串的，跳过的是模式串里的next(伪）位，从第next位开始比较。

第五章 树与二叉树

树的性质

1. 树的结点数n等于所有结点度数之和加1
2. 度为m的树中第i层上最多有 m^{i-1} 个结点。
3. 高度为h的m叉树至多有 $m^h-1)/(m-1)$ 个结点。
4. 度为m，有n个结点的树的最小高度是 $\log_m(n(m-1)+1)$
5. 度为m，具有n个结点的树的最大高度是n–m+1

完全二叉树：最后一层不一定满的满二叉树。
二叉排序树：左子树小于右子树。
平衡二叉树：AVL树。
正则二叉树：树中只有度为0或2的结点。
二叉树性质：

1. $n_0 = n_2 + 1$
2. 带入上面树的性质

当树中只有一个结点时，根节点就是叶结点。
顺序存储结构的二叉树每一层都要分配完整一层的储存空间。
【例题】对于任意一棵高度为5且有10个结点的二叉树，若用顺序存储结构保存，每个结点占一个存储单元，则存放该二叉树需要的存储单元数量至少是?
【答案】31

树的遍历

中序遍历的非递归算法：

```
void InOrder(BiTree T){
  InitStack(S);BiTree p = T;
  while(p||!IsEmpty(S)){
    if(p){
      Push(S,p);
      p = p->left;
    }
    else{
      Pop(S,p);visit(p);
      p = p->right;
    }
  }
}
```

先序遍历的非递归算法：

```
void PreOrder(BiTree T){
  InitStack(S);BiTree p = T;
  while(p||!IsEmpty(S)){
    if(p){
      visit(p);
      Push(S,p);
      p = p->left;
    }
    else{
      Pop(S,p);
      p = p->right;
    }
  }
}

```

后序遍历的非递归算法：

```
void PostOrder(BiTree T){
  InitStack(S);BiTNode *p = T, *r = NULL;
  while(p||!IsEmpty(S)){
    if(p){
      Push(S,p);
      p = p->left;
    }
    else{
      GetTop(S,p);
      if(p->right && p->right!=r)
        p = p->right;
      else{
        Pop(S,p);
        visit(p);
        r = p;//r记录最近访问的结点
        p = NULL;
      }
    }
  }
}
```

层次遍历：

```
void LevelOrder(BiTree T){
  InitQueue(Q);
  BiTree p = T;
  Enqueue(Q, p);
  while(!IsEmpty(Q)){
    DeQueue(Q, p);
    visit(p);
    if(p->left){
      EnQueue(Q, p->left);
    }
    if(p->right){
      Enqueue(Q, p->right);
    }
  }
}
```

线索二叉树

节点结构：

lchild	ltag	data	rtag	rchild

ltag = 0 , lchild指向左孩子；
ltag = 1 , lchild指向前驱；
rtag = 0 , rchild指向右孩子；
rtag = 1 , rchild指向后继。

在二叉树中有m是n的祖先，则使用（后序遍历）可以找到从m到n的路径。（这个题要用非递归算法中栈内元素分析而不是遍历结果）。

线索二叉树是一种物理结构。
二叉树线索化后，仍不能有效求解后序线索二叉树中求后序后继的问题。
【例题】先序序列为abcd的不同二叉树的个数是?
【答案】14. 卡特兰公式是求解栈中元素排列次序数量的公式，可以用在这里。
前序序列和中序序列的关系相当于以前序序列为入栈次序，以中序序列为出栈次序。
计算树高的算法：
用层次遍历，设置变量level记录当前结点所在层数，设置last指向当前层最右边的结点，每次层次遍历出队时与last指针比较，若两者相等，则层数加1，并让last指向下一层最右结点。

```
int Btdepth(BiTree T){
  if(!T) return 0;
  int front = -1, rear = -1;
  int last = 0, level = 0;
  BiTree Q[Maxsize];
  Q[++rear] = T;
  BiTree p;
  while(front<rear){
    p = Q[++front];
    if(p->left){
      Q[++rear] = p->left;
    }
    if(p->right){
      Q[++rear] = p->right
    }
    if(front == last){
      level++;
      last = rear;
    }
  }
  return level;
}
```

递归版本：

```
int Btdepth(BiTree T){
  if(T == NULL) return 0;
  ldep = Btdepth2(T->left);
  rdep = Btdepth2(T->right);
  if(ldep > rdep){
    return ldep+1;
  }
  else return rdep+1;
}
```

求公共祖先结点采用后序遍历的非递归形式，栈中的都是公共祖先节点。
满二叉树已知前序求后序：
【思路】后序序列的最后一个结点（根结点）等于前序序列的第一个结点（根节点）

```
void Pre2Post(char* pre[], int l1, int h1, char* post[], int l2, int h
2){
  int half;
  if(h1>=l1){
    post[h2] = pre[l1];
    half = (h1-l1) / 2;
    Pre2Post(pre, l1+1, l1+half, post, l2, l2+half-1);
    Pre2Post(pre, l1+half+1, l1, post, l2+half, h2-1);
  }
}
```

计算带权路径和WPL的方法：

法1：WPL = 树中全部叶节点带权路径长度之和

```
int WPL(Tree *T){
  return WPL1(T, 0);
}
int WPL1(Tree *T, int depth){
  if(T->left == T->right == NULL){
    return T->data*depth;
  }
  else return WPL1(T->left, depth+1)+WPL1(T->right, depth+1);
}
```

法2：WPL = 数中所有非叶结点的权值之和

```
int WPL(Tree *T){
  if(T->left == T->right == NULL){
    return 0;
  }
  else{
    w = WPL(T->left)+WPL(T->right);
    T->data = T->left->data + T->right->data;
    return w+T->weight;
  }
}
```

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

在树中，只要没有左结点，就是叶结点。

【例题】已知一颗有2011个结点的树，其叶结点个数为116，该树对应的二叉树中无右孩子的结点个数是?

【答案】1896

【解析】假设除这116个叶结点外全是独生子女，则转为二叉树后只有115个结点有兄弟，2011−115是1896

WPL = $\sum_{i=1}^{n} w_i l_i$（w是第i个叶结点的权值，l是叶结点到根节点的路径长度。

WPL最小的二叉树叫哈夫曼树或最优二叉树。

并查集

【例题】设哈夫曼编码的长度不超过4，若已对两个字符编码为1和01，则最多可对（）个字符编码。

【答案】4

最长的情况是0000，0001，0010，0011

并查集的结构是一种**双亲表示法**存储的树。

n个元素构成的集合树的深度是n，find操作的最坏时间复杂度是O（n），作优化后可以使深度不超过 $\lfloor log_2 n \rfloor + 1$ 。

合并两个长度分别为m和n的有序表，最坏情况下需要比较m+n−1次。

第六章 图

线性表可以是空表，树可以是空树，但图不可以是空图（顶点集不能为空，但边集可以为空）。

完全图：有n(n−1)/2条边的无向图是完全图。

连通图和连通分量都是无向图的概念（所有点都可达）。

强连通图和强连通分量是有向图的概念（所有点都双向可达）。

稀疏图：E<VlogV

若一个图有n个顶点，且有大于n−1条边，则此图一定有环。

对于一个有n个顶点的图，若是连通无向图，则其边数至少为n−1，若是强连通有向图，则其边数至少是n。

图的存储

邻接矩阵法：一个一维数组存顶点，一个二维数组存边。

```
#define MaxVertexNum 100
typedef struct{
  char vex[MaxVertexNum];
  int edge[MaxVertexNum][MaxVertexNum];
  int vexnum,arcnum;
} MGraph;
```

无向图第i行非0元素是顶点的度数，有向图第i行是出度，第i列是入度。

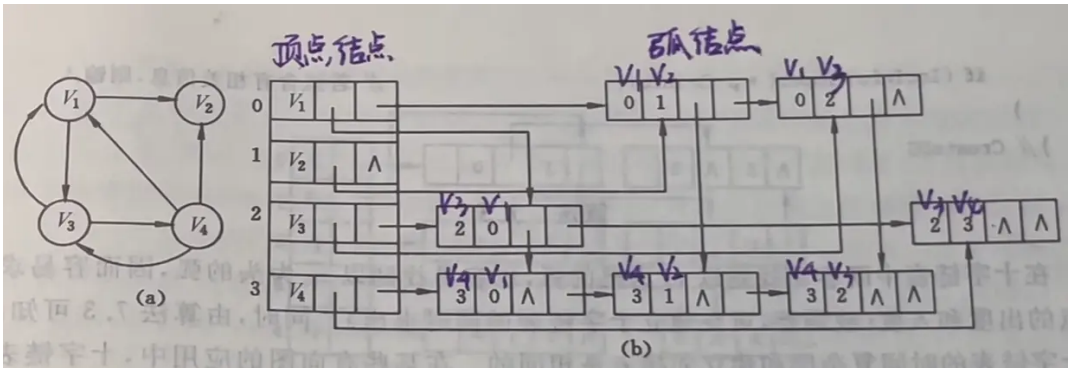
设图G的邻接矩阵为A， A^n **的元素** $A^n[i][j]$ **=由顶点i到j的长度为n的路径的数目。**

邻接表法：

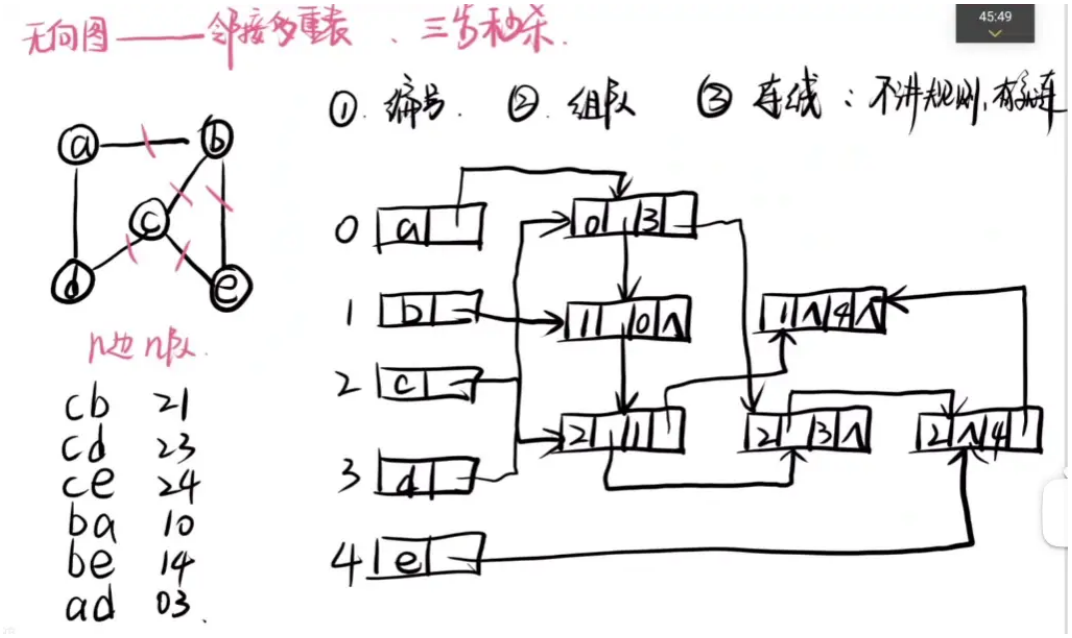
G为无向图，则需要的存储空间是O（V+2E），若为有向图，则为O（V+E）。

稀疏图用邻接表，稠密图用邻接矩阵。

十字链表法（有向图）：



邻接多重表（无向图）：



BFS和DFS

BFS和层次遍历很像。

BFS的空间复杂度是O（V），时间复杂度在邻接矩阵是O（ V^2 ），邻接表是O（V+E）。

BFS解决单源最短路径问题的算法如下：

```
void BFS_MIN(Graph G, int u){
  for(int i = 0; i < G.vexnum; i++){
    d[i] = inf;
  }
  visited[u] = TRUE; d[u] = 0;
  EnQueue(Q, u);
  while(!IsEmpty(Q)){
    DeQueue(Q, u);
    for(w = FirstNeighbour(G, u);w>=0;w = NextNeighbour(G,u,w)){
      if(!visited[w]){
        visited[w] = true;
        d[w] = d[u] + 1;
        EnQueue(Q, w);
      }
    }
  }
}
```

DFS类似于树的先序遍历。

时间复杂度与BFS一样，空间复杂度也是。

最小生成树算法

1. Prim算法：

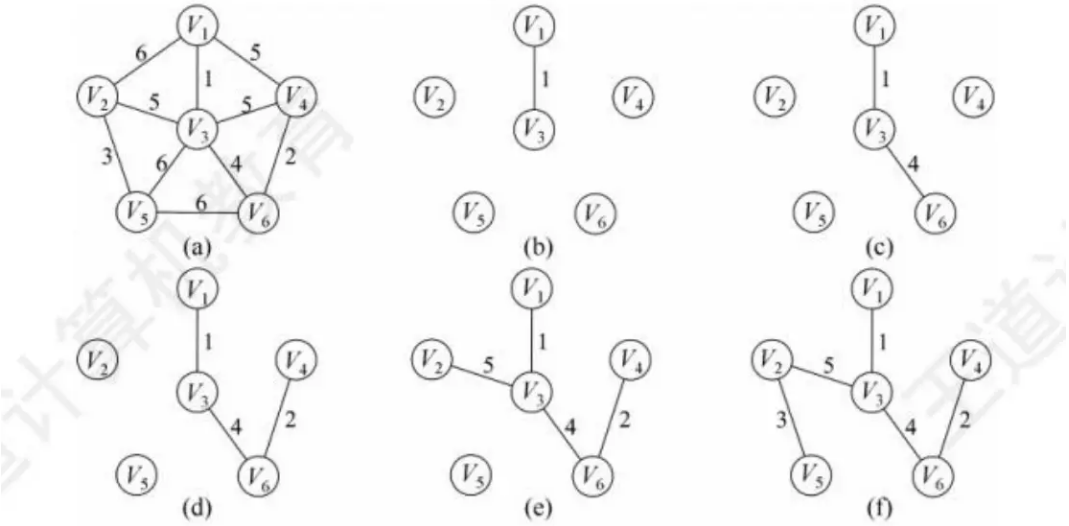


图 6.15　Prim 算法构造最小生成树的过程

Prim算法就是不断将距离现在已纳入集合中的点集最近的点纳入这个集合中。

时间复杂度是O（ V^2 ），适用于求解边稠密的图。

2. Kruskal算法：

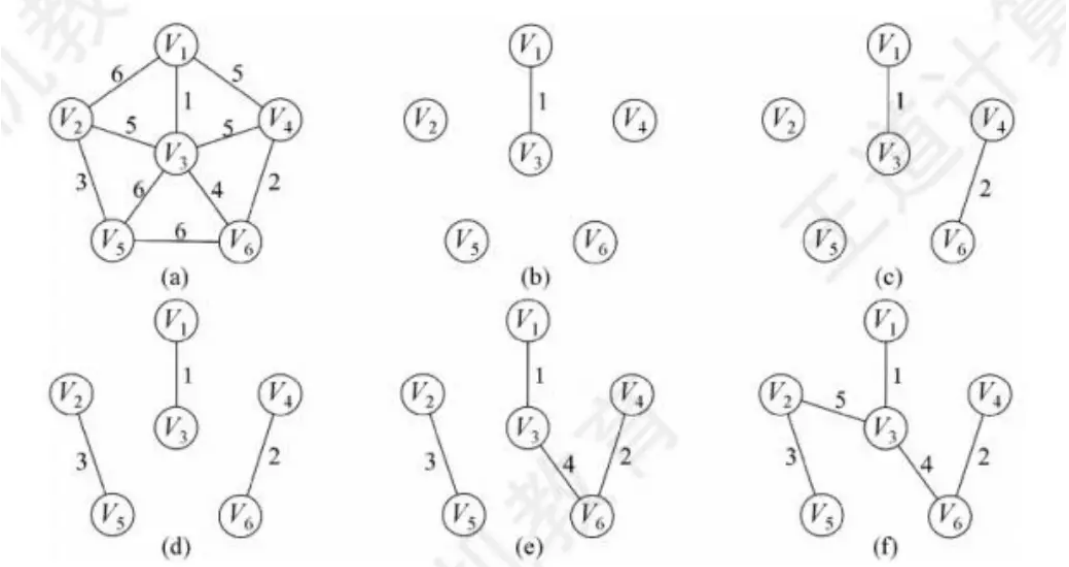


图 6.16　Kruskal 算法构造最小生成树的过程

Kruskal算法就是不断选取目前未被选取过且权值最小的边。

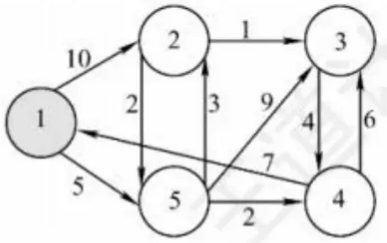
时间复杂度是O（ElogE）适合于边稀疏而顶点较多的图。

最短路径问题

Dijkstra算法求单源最短路径问题：

时间复杂度是O（ V^2 ），dijkstra算法不能求带负值的边。

Dijkstra算法和Prim算法很相似：



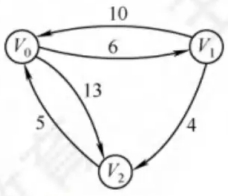
每轮得到的最短路径如下：
第1轮：1→5，路径距离为5
第2轮：1→5→4，路径距离为7
第3轮：1→5→2，路径距离为8
第4轮：1→5→2→3，路径距离为9

图 6.17　应用 Dijkstra 算法图

表 6.2　从 v_1 到各终点的 dist 值和最短路径的求解过程

顶点	第 1 轮	第 2 轮	第 3 轮	第 4 轮
2	10 $v_1 \rightarrow v_2$	8 $v_1 \rightarrow v_5 \rightarrow v_2$	**8** **$v_1 \rightarrow v_5 \rightarrow v_2$**	
3	∞	14 $v_1 \rightarrow v_5 \rightarrow v_3$	13 $v_1 \rightarrow v_5 \rightarrow v_4 \rightarrow v_3$	**9** **$v_1 \rightarrow v_5 \rightarrow v_2 \rightarrow v_3$**
4	∞	**7** **$v_1 \rightarrow v_5 \rightarrow v_4$**		
5	**5** **$v_1 \rightarrow v_5$**			
集合 S	**{1, 5}**	**{1, 5, 4}**	**{1, 5, 4, 2}**	**{1, 5, 4, 2, 3}**

Floyd算法求每对顶点间的最短路径：



(a) 有向图 G

$$\begin{bmatrix} 0 & 6 & 13 \\ 10 & 0 & 4 \\ 5 & \infty & 0 \end{bmatrix}$$

(b) G 的邻接矩阵

图 6.19　带权有向图 G 及其邻接矩阵

没有连接的点标为无穷，连到自身的点标为0。

表 6.3　Floyd 算法的执行过程

A	$A^{(-1)}$			$A^{(0)}$			$A^{(1)}$			$A^{(2)}$		
	V_0	V_1	V_2	V_0	V_1	V_2	V_0	V_1	V_2	V_0	V_1	V_2
V_0	0	6	13	0	6	13	0	6	**10**	0	6	10
V_1	10	0	4	10	0	4	10	0	4	**9**	0	4
V_2	5	∞	0	5	**11**	0	5	11	0	5	11	0

每一次操作黄色部分不动，红色部分做和，若是大于交叉点，则改变交叉点，否则不变。
时间复杂度是O（ V^3 ）
最短路径一定是简单路径。

拓扑排序

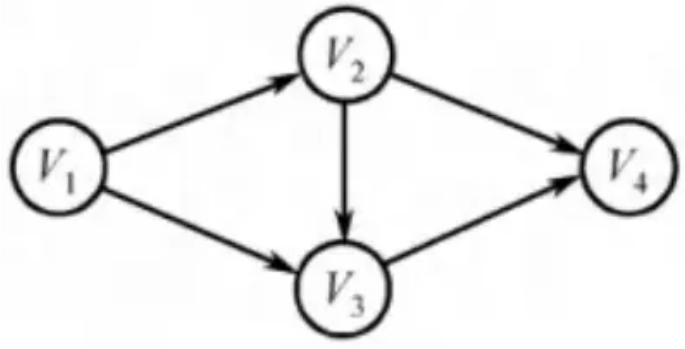
AOV网：有向无环图表示的一个工程。i必须先于活动j进行的一种关系。
对AOV网进行拓扑排序的算法：

1. 从AOV网中选择一个没有直接前驱（入度为0）的顶点输出。
2. 从网中删除该顶点和所有以它为起点的有向边。
3. 重复12直到当前网中不存在无前驱的顶点为止。

```
bool TopologicalSort(Graph G){
  InitStack(S);
  for(int i = 0; i < G.vexnum; i++){
    if(indegree[i] == 0) Push(S, i);
  }
  int count = 0;
  while(!IsEmpty(S)){
    Pop(S, i);
    print[count++] = i;
    for(p = G.vertices[i].firstarc;p;p=p->next){
      v = p->adjvex;
      if(!(--indegree[v])) Push(S, v);
    }
  }
  if(count < G.vexnum) return false;
  else return true;
}
```

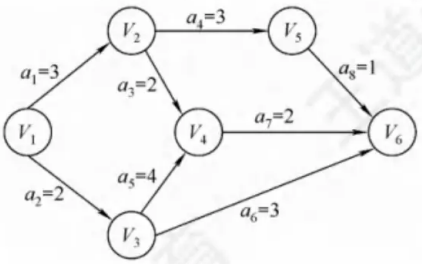
时间复杂度O（V+E)或O(V^2)。

在拓扑排序中为暂存入度为0的点，可以用栈也可以用队列。
若有向图的拓扑排序有序序列唯一，则图中入度为0和初读为0的点都仅有一个。
对有向图中的顶点适当地编号，使其邻接矩阵为三角矩阵且主对角线元素为0的充要条件是该有向图可以进行拓扑排序。
记住特殊的拓扑排序：1234



关键路径

AOE网：边上的权值表示开销，称为用边表示活动的网。（AOV是用点表示活动的网）
求关键路径是以拓扑排序为基础的。



	v_1	v_2	v_3	v_4	v_5	v_6
$v_e(i)$	0	3	2	6	6	8
$v_l(i)$	0	4	2	6	7	8

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(i)$	0	0	3	3	2	2	6	6
$l(i)$	1	0	4	4	2	5	6	7
$l(i)-e(i)$	1	0	1	1	0	3	0	1

其中 v_e 是从前往后计算的最大值， v_l 是从后往前计算的最小值。

e(i)是该弧的起点的顶点的 v_e (i)，l(i)是该弧的终点的顶点的 v_l 减去该弧的持续时间。

根据l(i)–e(i)=0的关键活动，就能得到关键路径，上图的关键路径就是（1，3，4，6）

时间复杂度	Dijkstra	Floyd	Prim	Kruskal	FDS	BFS	拓扑排序	关键路径
邻接矩阵	n^2	n^3	n^2		n^2	n^2	n^2	n^2
邻接表				eloge	n+e	n+e	n+e	n+e

【例题】序列中顶点不重复出现的路径称为简单路径，回路不是简单路径。

最小生成树不一定是最短路径，所以Prim和kruskal并不能求最短路径。

当带权连通图的任意一个环所包含的边的权值均不相等时，其最小生成树MST是唯一的。

第七章 查找

线性查找

平均查找长度： $ASL=\sum_{i=1}^{n}P_iC_i$ ，P是查找的概率，C是比较次数，n是查找表长度。

分块查找：块内无序，块间有序。

ASL = LI（索引查找）+LS（块内查找）

当s = $\sqrt{n}$ 时，平均查找长度为 $\sqrt{n}$ +1.

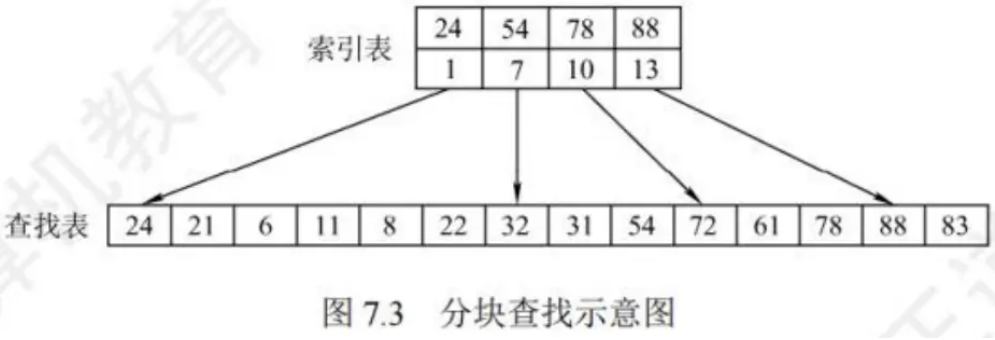


图 7.3 分块查找示意图

	顺序查找	折半查找	分块查找
时间复杂度	O(n)	O(logn)	
ASL	$\frac{n+1}{2}$	$\frac{n+1}{n}log(n+1)-1$	$\frac{s^2+2s+n}{2s}$ (s为每块有s个记录)

折半查找性能可以用二叉判定树衡量，但二叉排序树的查找性能1和数据输入顺序有关，最好情况下与折半查找相同，最坏情况即形成单支树时，其查找长度变成O（n）。

对表长为n的有序表进行折半查找，其判定树的高度是 $\lceil log_2(n+1) \rceil$

要想一棵树成为折半查找树，一定要满足的条件是：

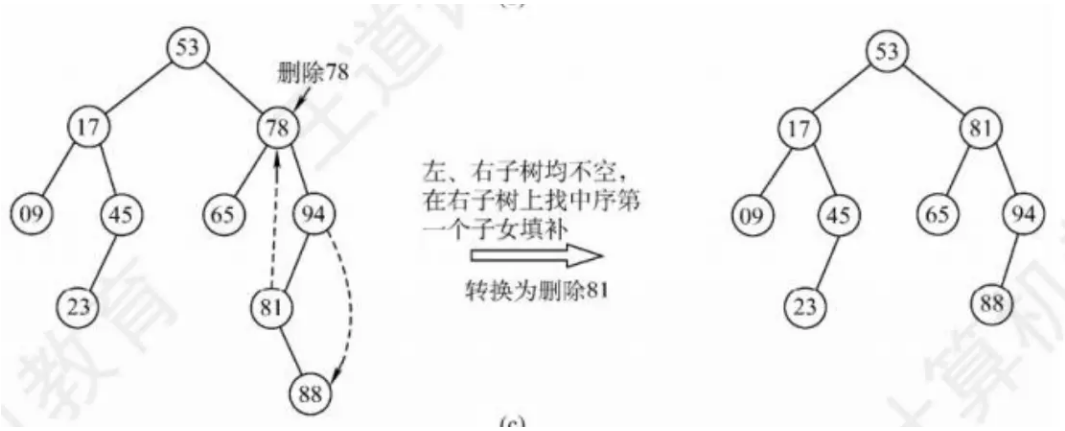
1. 要么，左子树与右子树结点数相等或左子树比右子树最多多一个结点。
2. 要么，左子树与右子树结点数相等或左子树比右子树最多少一个结点。

树型查找

二叉排序树的目的不是排序，而是加快查找插入和删除。

二叉排序树的删除：

1. 右子树空左子女补，左子树空右子女补。
2. 左右子树都不空，在右子树上找中序第一个子女填补。



二叉排序树的插入和删除平均时间复杂度是O（logn），但当对象是有序顺序表时就会变成O(n)。

平衡二叉树查找：假设以 n_h 表示深度为h的平衡二叉树中含有的最少结点数，有

$n_0 = 0, n_1 = 1, n_2 = 2, n_h = n_{h-2} + n_{h-1} + 1$

红黑树：

结论1：从根到叶结点的最长路径不大于最短路径的2倍。

结论2：有n个内部结点的红黑树的高度 $h \leq 2log_2(n+1)$.

结论3：新插入红黑树的结点初始着为红色。

红黑树插入：叔为红，父爷反色；叔为黑，父爷黑根红叶。

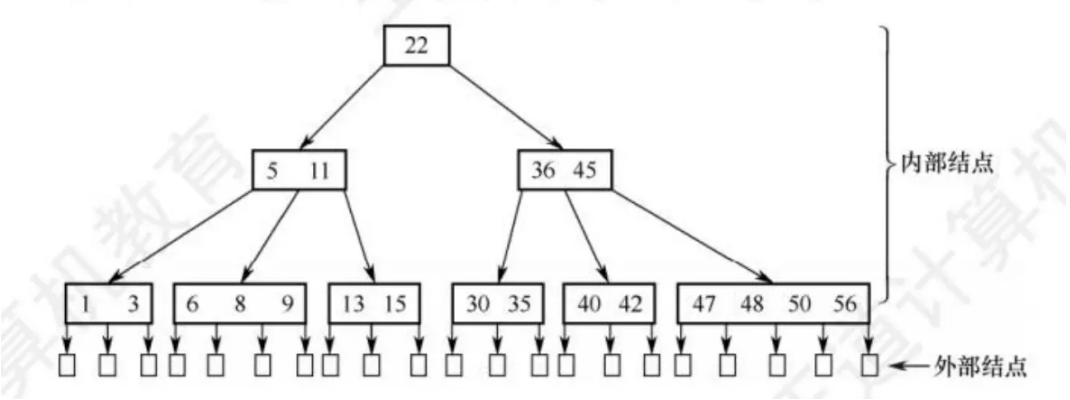
m阶B树就是所有结点的平衡因子均等于0的m路平衡树。

树中每个结点至多有m棵子树，即至多有m−1个关键字。

B树和B+树的区别就在于B树中没有重复的结点，B+树所有结点都在叶结点中。

B树性质：

1. 结点的孩子个数等于该结点中关键字个数加1。
2. 若根结点有关键字，则其子树个数必然大于等于2，因为子树个数等于关键字个数加一。
3. 除根结点外所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树。至多m棵子树。
4. 所有叶结点均在失败的位置。



B树的高度 $log_m(n+1) \le h \le log_{\lceil m/2 \rceil}((n+1)/2)+1$.

散列表

散列函数：

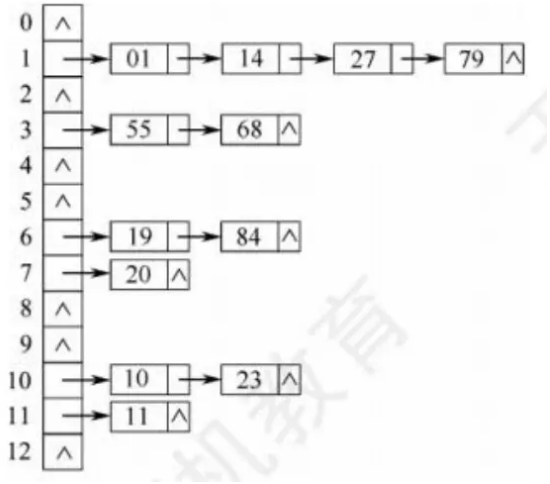
1. 直接定址法：H(key)=a*key+b
2. 除留余数法：H(key)=key%p
3. 数字分析法：与进制有关。
4. 平方取中法：取关键字的平方值的中间几位作为散列地址。

处理冲突的办法：

1. 开放定址法：Hi = （H(key)+di)%m

- 线性探测法：di = 1,2,3……
- 平方探测法：di = 1,−1,4,−4,9,−9
- 双散列法：di = i*Hash2(key)
- 伪随机序列法：di = 伪随机序列。

2. 拉链法：



散列表的查找效率取决于三个因素：散列函数、处理冲突的方法和装填因子。

装填因子是a=表中记录数/散列表长度

【例题】现有长度为11且初始为空的散列表HT，散列函数是H(key) = key%7,采用线性探查解决冲突，将关键字序列87，40，30，6，11，22，98，20依次插入HT后，HT查找失败的平均查找长度是?

【答案】6

【解析】插入结果：

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字	98	22	30	87	11	40	6	20			

查找失败的可能性是比如找0，遍历过0-7都没有，再比较第8个格子，一共比较了9次，以此类推，一共是9+8+7+6+5+4+3/7 = 6
【拓展】当散列函数变为H(key) = key%11时，将会变成9+8+7+6+5+4+3+2+1+1+1/11
在哈希表中删除元素后还要做个标记，不能直接删除。

第八章 排序

对任意n个关键字进行排序的比较次数至少为 $\log_2(n!)$.

插入排序

直接插入排序：将后面未经排序的数据逐个插入前面已完成排序的序列中。（适用于顺序存储和链式存储的线性表）
折半插入排序：与直接插入相比，在有序序列中查找的部分变成折半查找，时间复杂度不变，但效率提升了。（仅适用于顺序存储的线性表）
希尔排序：选取一个增量d，把数据分成d组。

```
void shellSort(int A[], int n){
  for(d = n/2; d >= 1; d/=2){
    for(int i = d+1; i <= n; i++){
      if(A[i]<A[i - d]){
        A[0] = A[i];
        for(int j = i-d; j > 0 && A[0]<A[j];j-=d){
          A[j+d] = A[j];
        }
        A[j+d] = A[0];
      }
    }
  }
}
```

n在某个特定范围时，希尔排序的时间复杂度约为 $O(n^{1.3})$ ，最坏情况和插入排序一样。

交换排序

冒泡排序：从前往后或从后往前两两比较相邻元素的值交换，每次能让一个最值到正确位置。（适用于顺序存储和链式存储）（**注意要在中途判断是否已经有序的话还需再多一次遍历**）
快速排序：每次选择一个枢轴，让比它小的在它前面，比它大的在后面，再对划分后的两部分继续选枢轴然后重复操作。（**快速排序在每次枢轴都将序列划分为长度相近的两段时效率最高**）

```
void QuickSort(int A[], int low, int high){
  if(low < high){
    int pivotpos = Partition(A, low, high);
    QuickSort(A, low, pivotpos-1);
    QuickSort(A, pivotpos + 1, high);
  }
}

int Partition(int A[], int low, int high){
  int pivot = A[low];
  while(low < high){
    while(low<high&&A[high]>=privot) --high;
```

```
    A[low] = A[high];
    while(low<high&&A[low]<=privot) ++low;
    A[high] = A[low];
  }
  A[low] = pivot;
  return low;
}
```

对n个元素进行第一趟快排后会确定一个基准元素，根据这个基准元素在数组中的位置有两种情况：

1. 基准元素在数组的首端或尾端，接下来对剩下的n-1个元素构成的子序列进行第二趟快排，结果：两趟快排后至少确定两个元素的最终位置，至少一个在首或尾；
2. 基准元素不在首尾，一趟后确定一个，两趟后又确定了两个，一共应该是3个。

选择排序

简单选择排序：每次找出一个最小或最大元素，移到合适的位置。（适用于顺序存储和链式存储）

堆排序：

堆的建立：每次从第n/2个元素开始从下往上调整堆。

堆的删除：输出根后，把最后一个元素作为根，然后从上往下调整。

（只适用于顺序存储的线性表）

堆的插入：插到最后检查上升。

归并排序

将两个或两个以上的有序表合并成一个新的有序表即归并。

```
int *b = (int*)malloc(sizeof(int)*(n+1))
void Merge(int a[],int low,int mid, int high){
  int i,j,k;
  for(k = low;k<high;k++) b[k] = a[k];
  for(i = low, j = mid+1, k=i; i <= mid&&j<=high; k++){
    if(b[i]<b[j]){
      a[k] = b[i++];
    }
    else a[k] = b[j++];
  }
  while(i<=mid) a[k++] = b[i++];
  while(i<=high) a[k++] = b[j++];
}

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

适用于顺序和链式线性表。

基数排序：基于关键字各位的大小进行排序。有MSD（最高位优先）和LSD（最低位优先）。
基数排序中建立的队列个数=进制数，与其他都无关。
计数排序：对每个待排序元素x，统计小于x的元素个数，就可以确认最终位置。
对于每一个数据项中有多个数据的比如（i,j)要同时让两个元素有序，就要让优先级高的最后完成以确保不会乱序，然后考虑稳定性，后排的要稳定，不然前面排好的就乱了。

算法性能比较

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
二路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

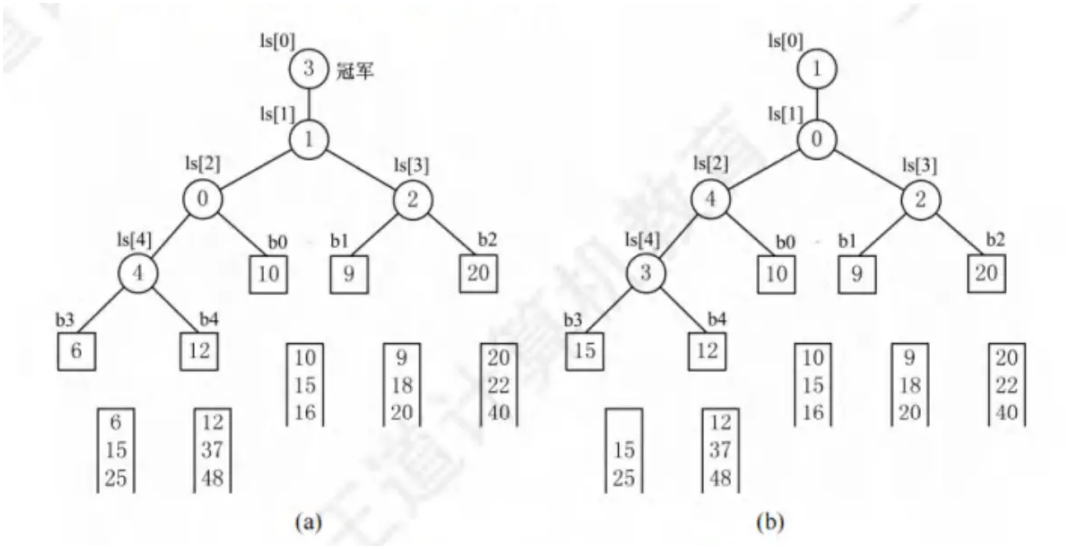
小结：

1. 若n较小，可采用直接插入排序或简单选择排序。两者比简单选择排序更好。
2. 若n较大，用快速排序、堆排序或归并排序。当待排序的关键字随机分布式，快排最好。若要最稳定的就是归并排序。
3. 若文件的初始状态关键字基本有序，用直接插入排序或冒泡排序。
4. 当文件的n个关键字随机分布时，任何借助“比较”的排序算法至少需要O(nlogn)
5. n很大且记录的关键字位数较少可分解用基数排序。
6. 记录本身信息量较大时，用链表作存储结构。

简单选择排序、折半插入排序和归并排序都是无论序列怎样，总比较次数都一定。

外部排序

增加归并路数k能减少归并趟数S，进而减少I/O次数。但与此同时，内部排序的困难加大。为此引入了败者树：



置换–选择排序：

表 8.2　置换–选择排序过程示例

输出文件 FO	工作区 WA	输入文件 FI
—	—	17, 21, 05, 44, 10, 12, 56, 32, 29
—	17 21 **05**	44, 10, 12, 56, 32, 29
05	**17** 21 44	10, 12, 56, 32, 29
05 17	10 **21** 44	12, 56, 32, 29
05 17 21	10 12 **44**	56, 32, 29
05 17 21 44	10 12 **56**	32, 29
05 17 21 44 56	10 12 32	29
05 17 21 44 56 #	**10** 12 32	29
10	29 **12** 32	—
10 12	**29** 32	—
10 12 29	**32**	—
10 12 29 32	—	—
10 12 29 32 #	—	—

最佳归并树：思想与哈夫曼树相同，都是计算最小WPL。

对严格k叉树来说有 $n_k = \dfrac{n_0 - 1}{k - 1}$

1. 若nk可以整除，说明内结点就是nk个。
2. 若有余数u，则需要再加k–u–1个空归并段。

多路平衡归并的作用是减少归并趟数。

归并趟数计算公式是 $S = \lceil log_m r \rceil$,m是归并路数，r是归并段个数。

在做m路平衡归并排序时，为实现输入/内部归并/输出的并行处理，需要设置2m个输入缓冲区和2个输出缓冲区。

在败者树中选取最小关键字的时间复杂度取决于败者树的高度，所需时间是O（logk）。

对n个记录，工作区大小为m，则生成的第一个初始归并段的长度的最大值是n，最小值是m。