第 1 章 引论

1. 本书算法解决的问题有两个：

(1) 选择问题：设有一组 N 个数而要确定其中第 k 个最大者。这是排序算法。

(2) 解字迷问题：输入是由一些字母和单词的二维数组组成，目标是要找出字迷的单词，这些单词可能是水平、垂直工沿对角线以任何方向放置的。

第 3 章 表、栈和队列

1. 抽象数据类型（ADT）是一些操作的集合。对于集合 ADT，我们可以有诸如并、交、测定大小以及求余等操作。或者，我们也可以只要两种操作：并和查找。
2. 我们将处理一般的形如 A_1、A_2……A_N 的表，我们说，这个表的大小是 N。A_{i+1} 后继 A_i，并称 A_{i-1} 前驱 A_i。
3. 链表由一系列不必在内存中相连的结构结成。每一个结构均包含 Next 指针，最后一个 Next 指针指向 NULL。表头（又称为哑结点）是一个标志结点，它并不包含数据。链表使用头结点来表示一个链表，但事实上所有结点都是同一个结构体。注意：链表实际上并不存在真正的表示方法，链表的第一个元素的 struct 结构也表示整个链表的 struct。
4. 双链表：在数据结构上附加一个域，使用它包含指向前一个单元的指针。

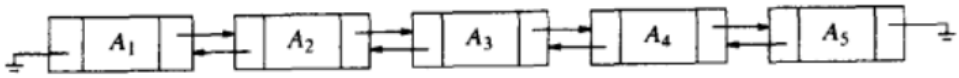


图 3-16 一个双向链表

5. 循环链表：第一个单元的前驱元指向最后的单元。

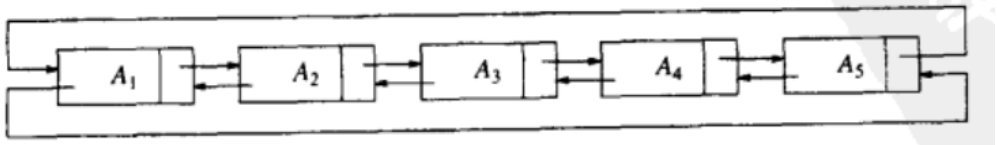


图 3-17 一个双向循环链表

6. 链表 ADT 集合（12 个）：

```
List MakeEmpty( List L );
int IsEmpty( List L );
int IsLast( Position P, List L );
Position Find( ElementType X, List L );
void Delete( ElementType X, List L );
Position FindPrevious( ElementType X, List L );
void Insert( ElementType X, List L, Position P );
void DeleteList( List L );
Position Header( List L );
Position First( List L );
Position Advance( Position P );
ElementType Retrieve( Position P );
```

7. 栈是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫做栈的顶。对栈的基本操作有 Push（进栈）和 Pop（出栈），前者相当于插入，后者则是删除最后插入的元素。栈有时又做 LIFO（后进先出）表（栈、同一位置、后进先出）

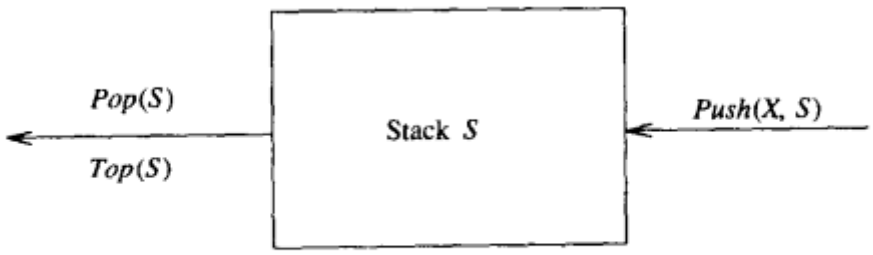


图 3-37　栈模型：通过 Push 向栈输入，通过 Pop 从栈输出

8. 由于栈是一个表，因此任何实现表的方法都能实现栈。
9. 栈 ADT

```
#ifndef _Stack_h

struct StackRecord;
typedef struct StackRecord *Stack;

int IsEmpty( Stack S );
int IsFull( Stack S );
Stack CreateStack( int MaxElements );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );
ElementType TopAndPop( Stack S );

#endif  /* _Stack_h */
```

10. 像栈一样，队列也是表。然而，使用队列时插入在一端进行而删除则在另一端进行。队列的基本操作是 Enqueue（入队），它是在表的末端（做队尾 rear）插入一个元素，还有 Dequeue（出队），它是删除（或返回）在表头（叫队头 front）的元素。　　（队列、先进先出）

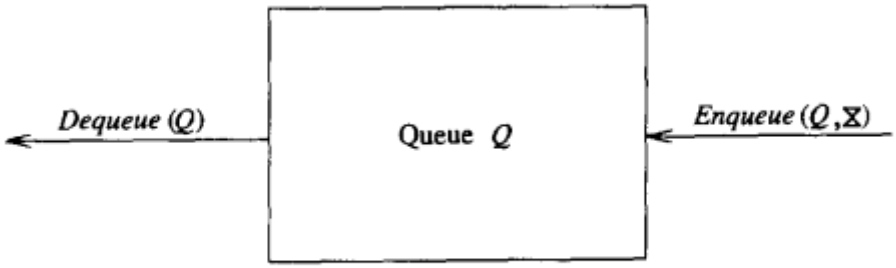


图 3-56　队列模型

使用数组实现队列时，如果 Rear 到达数组尾部，则 Rear 要绕回数组头部。

11. 队列 ADT

```
#ifndef _Queue_h

struct QueueRecord;
typedef struct QueueRecord *Queue;

int IsEmpty( Queue Q );
int IsFull( Queue Q );
Queue CreateQueue( int MaxElements );
void DisposeQueue( Queue Q );
void MakeEmpty( Queue Q );
void Enqueue( ElementType X, Queue Q );
ElementType Front( Queue Q );
void Dequeue( Queue Q );
ElementType FrontAndDequeue( Queue Q );

#endif  /* _Queue_h */
```

12. 本章的表指的是链表、栈和队列。

13. 链表、栈、队列的基本 ADT 操作：

- 创建表
- 删除表
- MakeEmpty
- IsEmpty
- IsLast 或者 Isfull
- 插入表元素

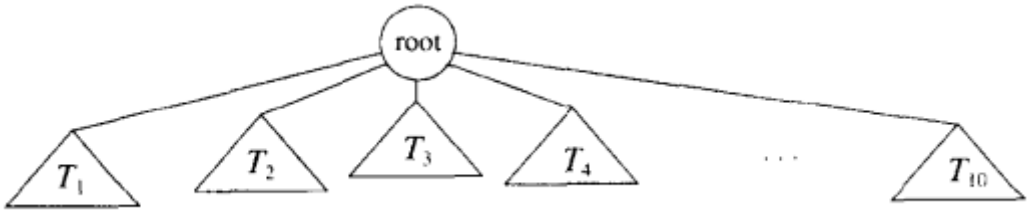
- 删除表元素
- 查找，如 find
- 查找前一个元素，如 FindPrevious
- 查找后一个元素
- 返回指定位置的表元素

表在实用时可适当增减操作。

第 4 章 树

1. 本章介绍的数据结构大部分操作的运行时间平均为 O(log N)。这种数据结构叫二叉查找树。
2. 哈希适合等于性的查找，树结构适合“范围查找”。
3. 一棵树是一些节点的集合。这个集合可以是空集。若非空，则一棵树由根 r 的节点以及 0 个或多个非空子树组成，这些子树中每一棵的根都被来自根 r 的一条有向边所连接。
4. 概念：

- 第一棵子树的根叫做根 r 的儿子，而 r 是每一棵子树的根的父亲。没有儿子的节点称为树叶，具有相同父亲的节点称为兄弟。
- 对任意节点 n_i，n_i 的深度为从根到 n_i 的惟一路径的长。因此，根的深度为 0。n_i 的高是从 n_i 到一片树叶的最长路径的长。因此，所有树叶的高都是 0。树的高就等于它的根的高。
- 如果存在从 n_1 到 n_2 的一条路径，那么 n_1 是 n_2 的一位祖先，而 n_2 是 n_1 的后裔。如果 $n_1 \neq n_2$，那么 n_1 是 n_2 的一位真祖先，而 n_2 是 n_1 的真后裔。



4. 由于每个节点的儿子数可以很大而且事先并不知道，因此在数据结构中建立到各儿子节点直接的链接并不可行。
5. 树的典型声明：

```
struct TreeNode
{
    ElementType Element;
```

```
    SearchTree  FirstChild;     // 第一个儿子
     SearchTree  NextSibling;    //下一个兄弟
};
```

6. 先序遍历：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：首先访问左子树，然后遍历根结点，最后遍历右子树。

   后序遍历：首先访问左子树，然后遍历右子树，最后遍历根结点。

7. 二叉树是一棵树，其中每个节点都不能有多于两个的儿子。二叉树的一个性质是平均二叉树的深度要比 N 小得多，二叉查找树的深度平均值为 O(log N)，不幸的是，这个深度最大可以大到 N-1（只有左子树或只有右子树时可以大到 N-1）。　　（二不多二）

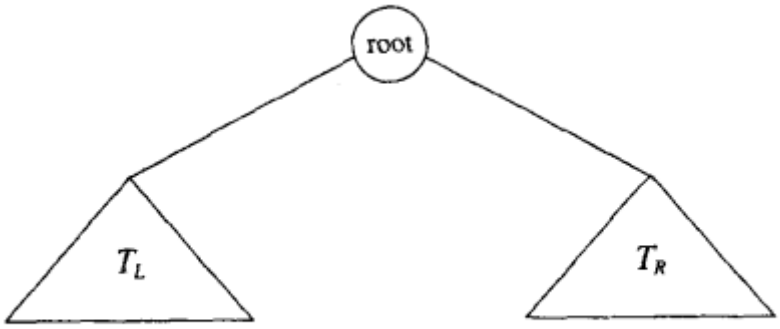


图 4-11　一般二叉树

8. 二叉树的定义：

```
struct TreeNode
{
    ElementType Element;
    SearchTree  Left;        // 左子树
    SearchTree  Right;       // 右子树
};
```

应用于链表的许多法则都可以应用到树上。特别地，当进行一次插入时，必须调用 malloc 创建一个节点，调用 free 删除后被释放。

9. 在二叉查找树中，对于树中的每个节点 X，它的左子树中所有关键字值小于 X 的关键字值，而它的右子树中所有关键字值大于 X 的关键字值。（二查、左<中<右）

10. 二叉查找树 ADT：

- MakeEmpty：清空查找树，主要用于初始化。
- Find：查找
- FindMin 和 FindMax：返回树中最小元和最大元的位置。
- Insert
- Delete

11. AVL 树是带有平衡条件的二叉查找树，这个平衡条件必须要容易保持，而且它必须保证树的深度是 O(log N)。最简单的想法是要求左右子树具有相同的高度。另一种平衡条件是要求每个节点必须要有相同高度的左子树和右子树。（AVL 平衡）
12. AVL 树的定义：

```
struct AvlNode
{
    ElementType Element;
    AvlTree  Left;          // 左子树
    AvlTree  Right;         // 右子树
    int      Height;        // 高度
};
```

13. 一棵 AVL 树是其每个节点的左子树和右子树的高度最多差 1 的二叉查找树。注意，这里差 1 指的是根节点的所有叶子节点之间最多相差 1，而不是某个节点的叶子节点最多相差 1。如果某叶子节点没有兄弟，则不能在该叶子节点上再插入数据。 （AVL 差 1）
14. AVL 树插入隐含着的困难在于，插入一个节点可能破坏 AVL 树的特性。在插入以后，只有那些从插入点到根节点的路径上的节点的平衡可能被改变。因为只有这些节点的子树可能发生变化。
15. AVL 树插入后破坏了节点平衡，让我们把必须重新平衡的节点叫做 a。由于任意节点最多有两个儿子，因此高度不平衡时，a 点的两棵子树的高度差 2。容易看出，这种不平衡可能出现在下面四种情况：（外边单、内部双）

- 对 a 的左儿子的左子树进行一次插入
- 对 a 的左儿子的右子树进行一次插入
- 对 a 的右儿子的左子树进行一次插入
- 对 a 的右儿子的右子树进行一次插入

16. 第一种情况是插入发生在“外边”的情况（即左-左或者右-右），该情况通过对树的一次单旋转完成调整。第二种情况是插入发生在“内部”的情形（即左-右或者右-左），该情况通过稍微复杂些的双旋转来处理。

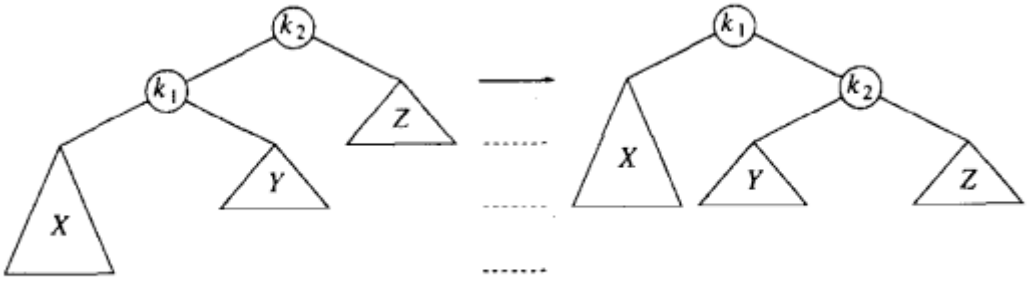


图 4-31　调整情形 1 的单旋转

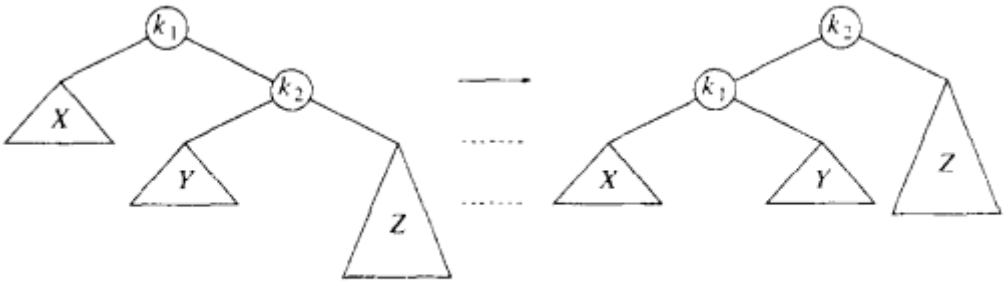


图 4-33　单旋转修复情形 4

17. 单旋转：

```
/* 仅当 K2 有一个左子树时，这个函数被调用 */
/* 在 K2 和它的左子树之间进行一次旋转 */
/* 更新高，并返回新的根节点 */
/* 其实就是修改左右子树的指针 */
static Position SingleRotateWithLeft( Position K2 )
  {
        Position K1;

        K1 = K2->Left;
        K2->Left = K1->Right;       // 修改 K2 的左子树
        K1->Right = K2;           // 修改 K1 的右子树

        K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
        K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;

        return K1;  /* New root */
  }

  /* 仅当 K1 有一个右子树时，这个函数被调用 */
```

```
/* 在 K1 和它的右子树之间进行一次旋转 */
/* 更新高，并返回新的根节点 */
static Position SingleRotateWithRight( Position K1 )
{
    Position K2;

    K2 = K1->Right;
    K1->Right = K2->Left;
    K2->Left = K1;

    K1->Height = Max( Height( K1->Left ), Height( K1->Right ) ) + 1;
    K2->Height = Max( Height( K2->Right ), K1->Height ) + 1;

    return K2;  /* New root */
}
```

18. 双旋转（也就是进行两次单旋转）：　　（双插中）

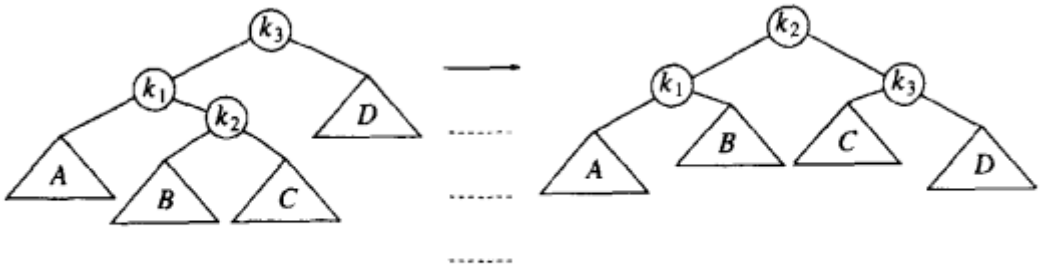


图 4-35　左-右双旋转修复情形 2

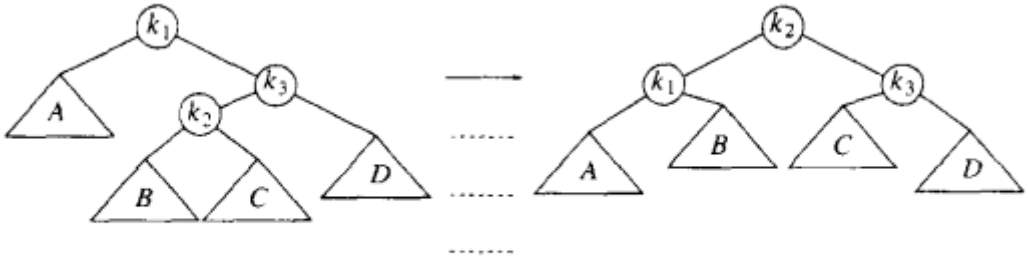


图 4-36　右-左双旋转修复情形 3

```
/* 仅当 K3 有一个左子树以及 K3 的左子树有一个右子树时 */
/* 这个函数被调用 */
/* 进行左-右双旋转 */
```

```
/* 更新高，并且返回新的根节点 */
static Position DoubleRotateWithLeft( Position K3 )
{
    /* Rotate between K1 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left );

    /* Rotate between K3 and K2 */
    return SingleRotateWithLeft( K3 );
}

/* This function can be called only if K1 has a right */
/* child and K1's right child has a left child */
/* Do the right-left double rotation */
/* Update heights, then return new root */
/* 仅当 K1 有一个右子树以及 K1 的右子树有一个左子树时 */
/* 这个函数被调用 */
/* 进行右-左双旋转 */
/* 更新高，并且返回新的根节点 */
static Position DoubleRotateWithRight( Position K1 )
{
    /* Rotate between K3 and K2 */
    K1->Right = SingleRotateWithLeft( K1->Right );

    /* Rotate between K1 and K2 */
    return SingleRotateWithRight( K1 );
}
```

19. 将一个数插入 AVL 树，如果树的高度不变，则插入完成，如果高度发生变化，则需要做适当的单旋转或双旋转。子树的高度差不能大于 1。

20. AVL 树 ADT：

```
#ifndef _AvlTree_H

struct AvlNode;
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;

AvlTree MakeEmpty( AvlTree T );
Position Find( ElementType X, AvlTree T );
Position FindMin( AvlTree T );
Position FindMax( AvlTree T );
AvlTree Insert( ElementType X, AvlTree T );
AvlTree Delete( ElementType X, AvlTree T );
ElementType Retrieve( Position P );

#endif  /* _AvlTree_H */
```

21. 伸展树，也叫分裂树，是一种二叉查找树，它能在 O(log n)内完成插入、删除和查找操作。伸展树的特点是被查频率高的条目处于靠近根的位置。每次查找后对树进行重构，把被查找的条目移动到根附近，伸展树应运而生。

22. 伸展树重构存在两种方法：

- 单旋：在查找完位置节点 X 的条目 i 之后，旋转链接 X 和其父节点的边。
- 伸展：X、X 的父亲和祖父之间呈现之字形时（左-右或者右-左），进行双旋转，否则，进行一字形旋转。

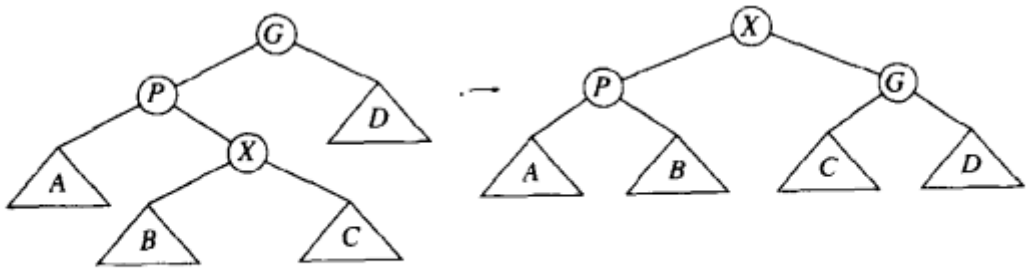


图 4-44　之字形(Zig-zag)情形

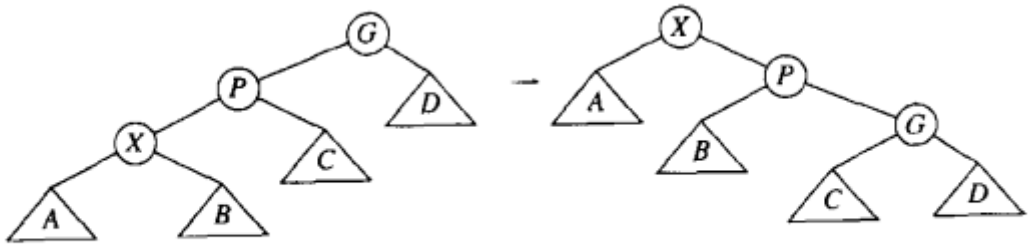


图 4-45　一字形(Zig-zig)情形

22. 阶为 M 的 B-树是一棵具有下列结构特性的树：

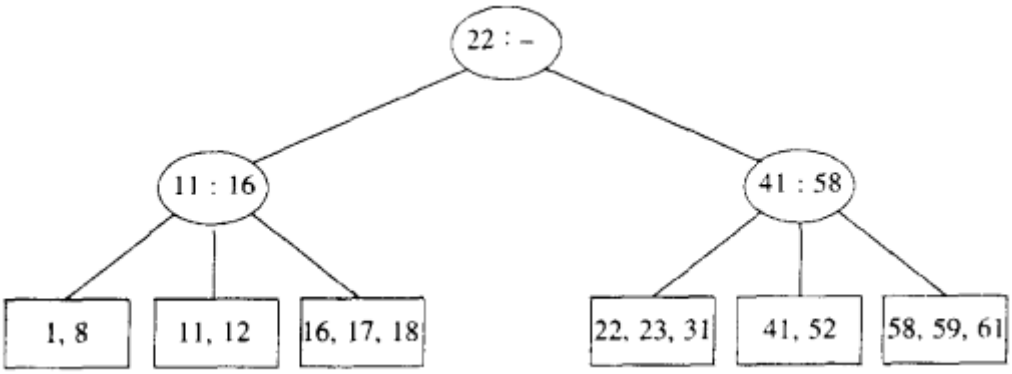
- 树的根或者是一片树叶，或者其儿子数在 2 和 M 之间；
- 除根外，所有非树叶节点的儿子数在 M/2 和 M 之间；
- 所有的树叶都在相同的深度上。

所有的数据都存储在树叶上。B-树的深度为 $O(\log_{M/2} N)$。每一个内部节点上皆含有指向该节点各儿子的指针 P1、P2……Pm 和分别代表在子树 P1、P2……Pm 中发现的最小关键字的值 K1、K2……Km-1。4 阶 B-树更流行的称呼是 2-3-4 树，而 3 阶 B-树叫做 2-3 树。

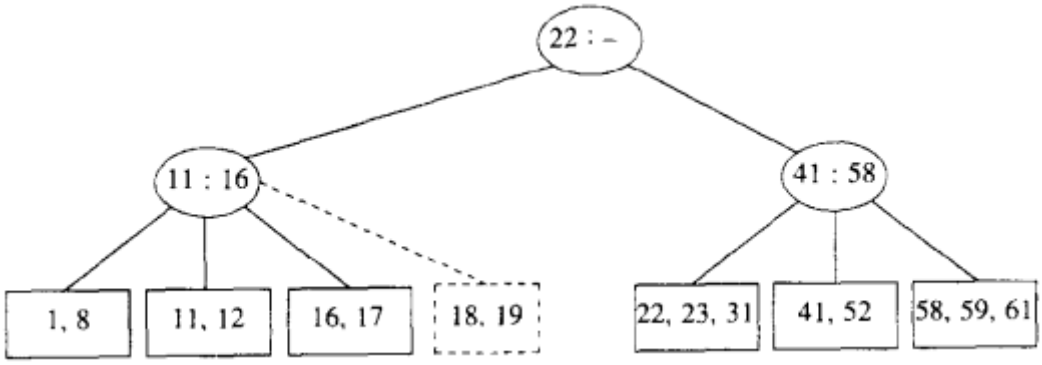
23. 插入数据到 M 阶 B-树有两种情况：　（小插大分）

- 如果叶子节点数据小于 M，则直接插入数据到该叶子数据。
- 如果叶子节点数据等于 M，则需要向上将内部节点分裂成两个节点。如果内

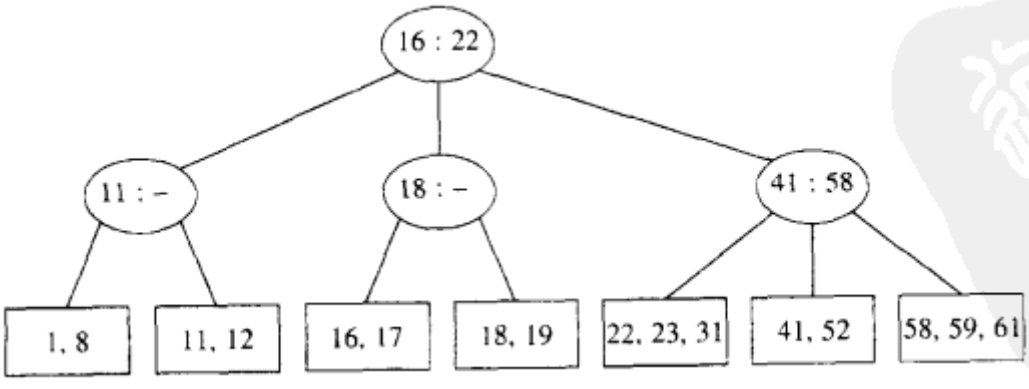
部节点大于 M，则继续向上分裂内部节点。如图，3 阶 B-树，



将 19 插入 18 后面，则叶子节点数据个数大于 4，该叶子分裂成两个节点



分裂之后，叶子节点个数大于 3，则继续分裂。



注：2-3 树内部节点的第一个数据表示第 2 个子节点的第一个后裔的值，第二个数据表示第 3 个子节点的第一个后裔的值，依此类推。注意是后裔的值，而不是子节点的值。

24. 二叉查找树：左小于父，右大于父。

AVL 树：左右高度最多差 1 的二叉查找树。

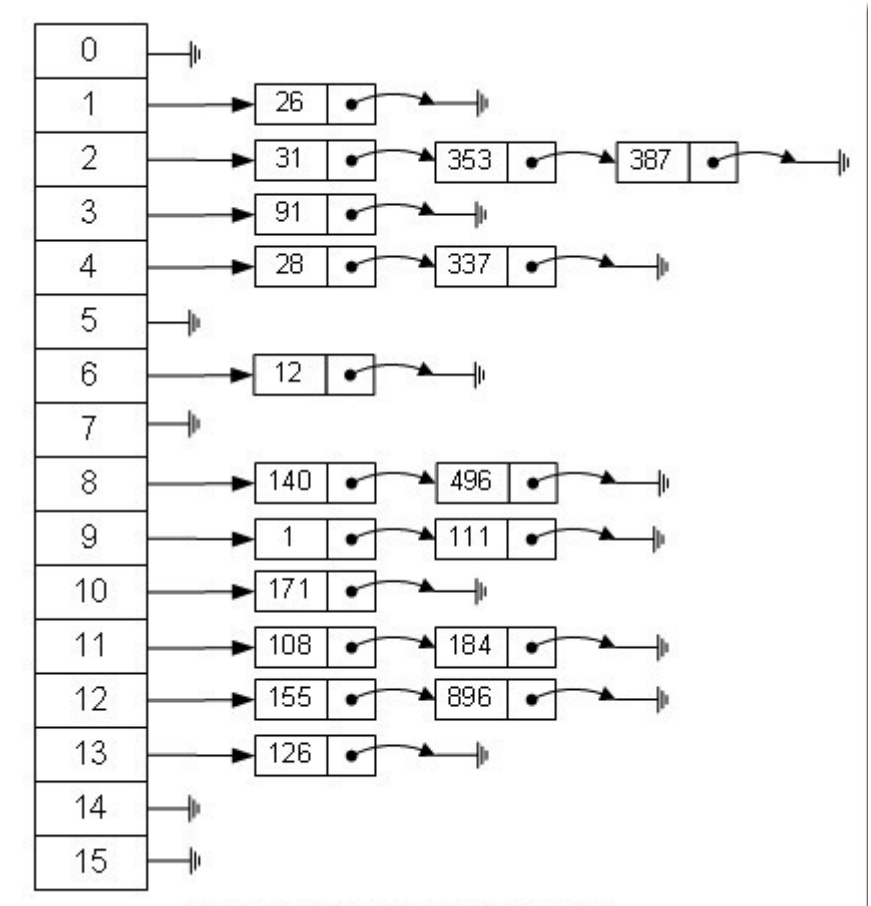
伸展树：被查频率高的条目处于靠近根的位置

第 5 章 散列

1、散列表用于根据给定的部分数据，查找出该数据背后更多的数据。比如，根据姓名查找出具体的员工信息。散列表不支持诸如 FindMin、FindMax 以及以线性时间将排过序的整个表进行打印的操作。

2、符号表是一种用于存储键值对的数据结构，平常使用的数组也可以看做是一个特殊的符号表，数组中的“键”即为数组索引，值为相应的数组元素。也就是说，当符号表中所有的键都是较小的整数时，我们可以使用数组来实现符号表，将数组的索引作为键，而索引处的数组元素即为键对应的值。

散列表是对以上策略的一种“升级”，使用散列函数将给定的键转化为一个“数组的索引”，得到了索引后，通过这个索引访问到相应的值。



用斐波那契散列法重新调整过的哈希表

0 到 15 是一个数组，称为桶，散列函数的作用就是根据给定的键和散列函数，计算出该键对应的数据位于哪个数组索引之中。

3、假设桶大小为 M，散列函数应该能够均匀并独立地将所有的键散布于 0 到 M – 1 之间。

4、散列表大小最好是个素数。

5、当两个关键字散列到同一个值时称为冲突。例如，不同的姓名可能散列到同一个值。

6、处理冲突：

(1) 分离链接法：桶中的元素存放的是链表，初始时所有链表均为空，当一个键被散列到一个桶时，这个键就成为相应桶中链表的首结点，之后若再有一个键因冲突被散列到这个桶，第二个键就会成为链表的第二个结点，以此类推。得到链表后再依照其它方法比对并查找数据。分离链接法 ADT：

```
#ifndef _HashSep_H

struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable( int TableSize );
void DestroyTable( HashTable H );
Position Find( ElementType Key, HashTable H );
void Insert( ElementType Key, HashTable H );
ElementType Retrieve( Position P );
/* Routines such as Delete and MakeEmpty are omitted */

#endif  /* _HashSep_H */
```

(2) 开放定址法的主要思想是：用大小为 M 的数组保存 N 个键值对，其中 M > N，数组中的空位用于解决碰撞问题。

- 线性探测法：当计算映射关系时发现冲突时，检查数组的下一个位置，直到找到一个空单元插入为止。这也就意味着，当数组已满时，查找会陷入死循环。
- 平方探测法：根据下列公式算出索引值：

$$h_i(x) = (Hash(x) + f(i))\%TableSize$$

其中 Hash(x)是冲突的索引值，$f(i)=i^2$，i 从 1 开始取值，TableSize 是表的大小。假设冲突值为 10，TableSize 为 100，则第一次索引值为（$10+1^2$）%100=11，若 11 还是冲突，则取 i 为 2，依此类推。

7、再散列：对于使用平方探测的开放定址散列法，如果表的元素填得太满，那么操作的运行时间将过长，且 Insert 操作可能失败。解决方法是使用一个相关的新散列函数建立另外一个大约两倍大的表，扫描整个原始散列表，计算每个（未删除的）元素的新散列值并将其插入到新表中。

8、当数据量太大以至于装不进主存的情况时，可考虑可扩散列。可扩散列的实现原理和 B-树相同，都是当数据量太大时将一个表分裂成两个表。

第 6 章 优先队列（堆）

1、优先队列是允许至少下列两种操作的数据结构： Insert(插入) 以及 DelteMin(删除最小者)。

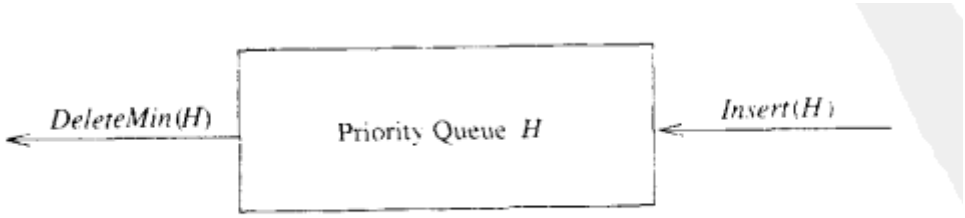


图 6-1 优先队列的基本模型

2、优先队列的简单实现：

- 链表
- 始终让表保持排序状态
- 二叉查找树

3、优先队列普遍使用二叉堆实现。

4、二叉堆： （堆、满）

1) 结构性质：堆是一棵完全填满的二叉树，有可能的例外是在底层，底层上的元素从左到右填入，这样的树称为完全二叉树。一项重要的观察发现，因为完全二叉树很有规律，所以它可以用一个数组表示而不需要指针。对于数组中任一位置 i 上的元素，其左儿子在位置 2i，右儿子在左儿子后的单元（2i+1）中，i 的父亲则在位置（i/2）位置上。因此，一个堆数据结构将由一个数组（不管关键字是什么类型）、一个代表最大值的整数以及当前的堆大小组成。

```
struct HeapStruct
        {
            int Capacity;
            int Size;
            ElementType *Elements;
        };
```

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

图 6-3　完全二叉树的数组实现

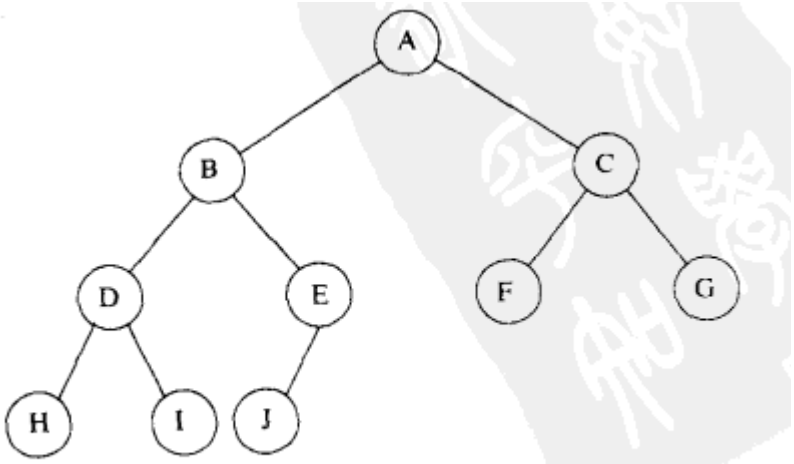


图 6-2　一棵完全二叉树

注意：数组实现的保存次序，以及完全二叉树并不是二叉查找树。

关键词：完全填满、有序性　　（满、父小于子）

1　堆序性质：在一个堆中，对于每一个节点 X，X 的父亲的关键字小于（或等于）X 的关键字，根节点除外（它并没有父亲）。注意：这里并没有说左子树大于右子树或者右子树大于左子树。

2　基本的堆操作：（Insert 空穴）

- Insert(插入)：为将 X 插入到堆中，我们在下一个空闲位置创建一个空穴，如果 X 可以放在该空穴中而并不破坏堆的序，那么插入完成。否则，我们把空穴的父节点上的元素移入空穴中，这样，空穴朝着根的方向上行一步。继续该过程直到 X 被放入空穴中为止。这种一般的策略叫做上滤。

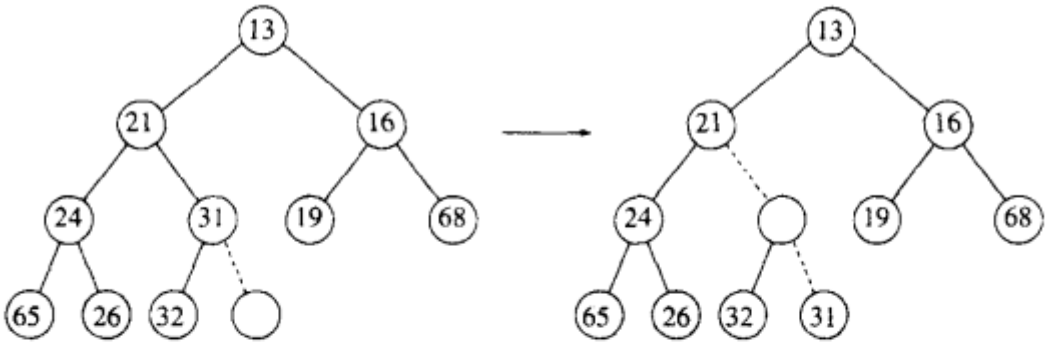


图 6-6　尝试插入 14：创建一个空穴，再将空穴上冒

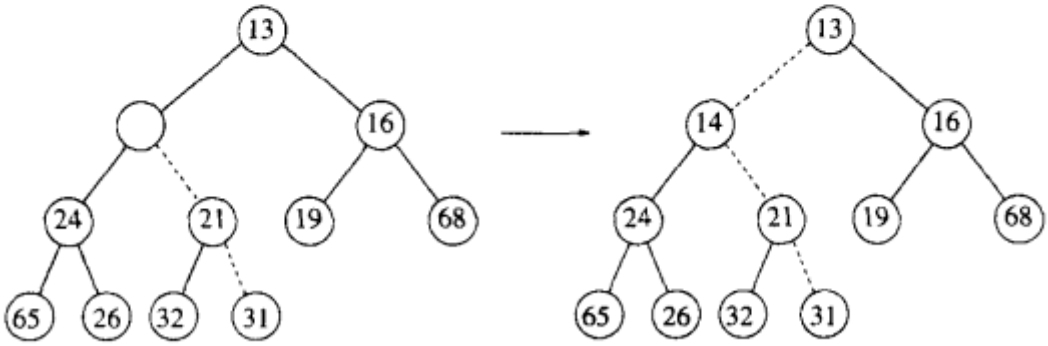


图 6-7　将 14 插入到前面的堆中的其余两步

```
void Insert( ElementType X, PriorityQueue H )
        {
            int i;

            if( IsFull( H ) )
            {
                Error( "Priority queue is full" );
                return;
            }

            /* i/2 是最后一行最后一个子树的的父亲，如果该子树小于其父亲，则将父亲与该子树交换值(事实上只给子树赋值) */
            for( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
                H->Elements[ i ] = H->Elements[ i / 2 ];

            /* 注意，经过 i/=2 的计算，这里的 i 和上面的 i 已经不一样了 */
            H->Elements[ i ] = X;
```

- DeleteMin(删除最小元)：删除一个元素时，会在根节点处产生一个空穴。将空穴的两个儿子中较小者移入空穴，这样就把空穴向下推了一层。重复该步骤直到堆的最后一个元素 X 可以被放入空穴中。

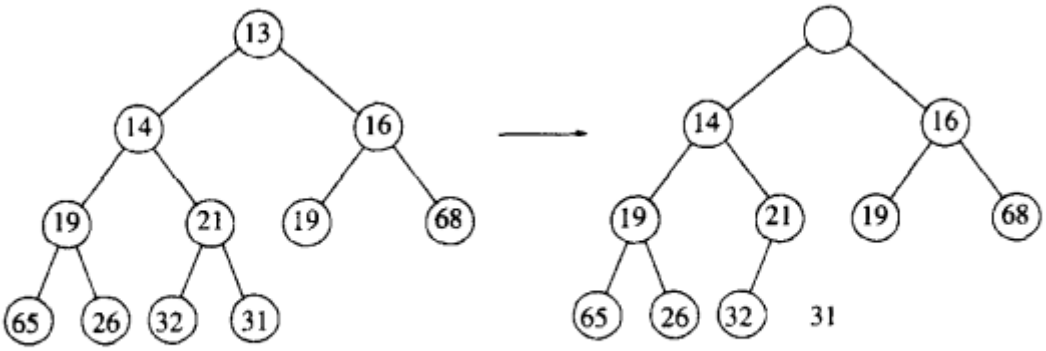


图 6-9　在根处建立空穴

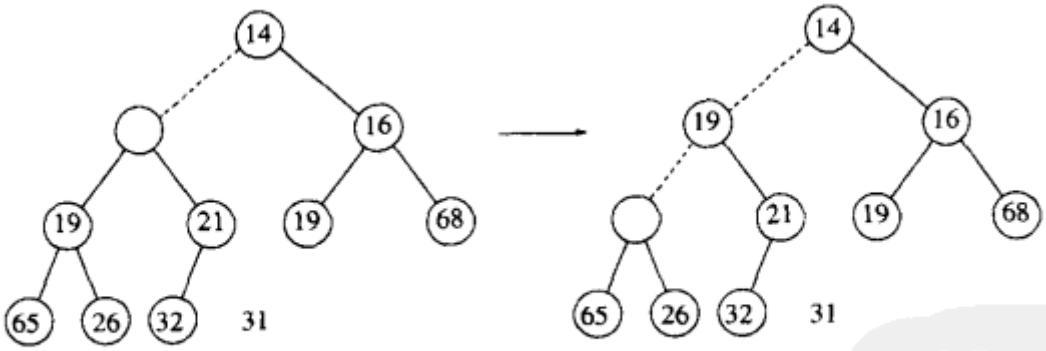


图 6-10　在 DeleteMin 中的接下来的两步

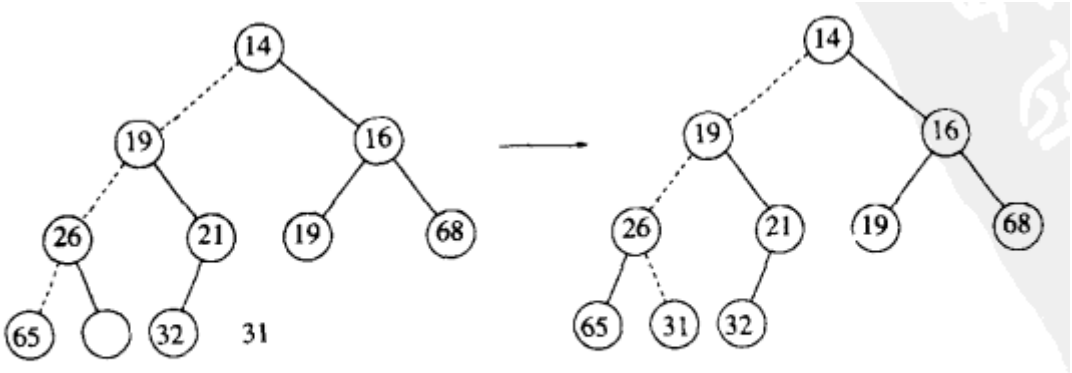


图 6-11　在 DeleteMin 中的最后两步

```
ElementType DeleteMin( PriorityQueue H )
{
    int i, Child;
```

```
    ElementType MinElement, LastElement;

    if( IsEmpty( H ) )
    {
        Error( "Priority queue is empty" );
        return H->Elements[ 0 ];
    }
    MinElement = H->Elements[ 1 ];
    LastElement = H->Elements[ H->Size-- ];

    for( i = 1; i * 2 <= H->Size; i = Child )
    {
        /* Find smaller child */
        Child = i * 2;
        if( Child != H->Size && H->Elements[ Child + 1 ]
                                    < H->Elements[ Child ] )
            Child++;

        /* Percolate one level */
        if( LastElement > H->Elements[ Child ] )
            H->Elements[ i ] = H->Elements[ Child ];
        else
            break;
    }
    H->Elements[ i ] = LastElement;
    return MinElement;
}
```

5、二叉堆 ADT：

```
#ifndef _BinHeap_H

struct HeapStruct;
typedef struct HeapStruct *PriorityQueue;

PriorityQueue Initialize( int MaxElements );
void Destroy( PriorityQueue H );
void MakeEmpty( PriorityQueue H );
void Insert( ElementType X, PriorityQueue H );
ElementType DeleteMin( PriorityQueue H );
ElementType FindMin( PriorityQueue H );
int IsEmpty( PriorityQueue H );
int IsFull( PriorityQueue H );

#endif
```

6、d-堆是二叉堆的简单推广，它恰像一个二叉堆，只是所有的节点都有 d 个儿子（因此二叉堆是 2-堆）。（d-堆、恰像二叉堆、d 个儿子）

7、左式堆

1) 像二叉堆那样，左式堆也具有结构特性和有序性。左式堆也是二叉树。左式堆和二叉树间惟一的区别是：左式堆不是理想的平衡，而实际上是趋向于非常不平衡。定义左式堆：

```
struct TreeNode
    {
        ElementType   Element;
        PriorityQueue Left;
        PriorityQueue Right;
        int           Npl;
    };
```

1 零路径长：把任一节点 X 的零路径长 Npl(X)定义为从 X 到一个没有两个儿子的节点的最短路径的长。因此，具有 0 个或 1 个儿子的节点的 Npl 为 0。

2 左式堆性质是：对于堆中每一个节点 X，左儿子的零路径长至少与右儿子的零路径长一样大。左式堆趋向于加深左路径，所以右路径应该短。注意：这里说的是零路径长至少一样大，并没有说相等，所以左儿子的零路径长可以大于右儿子的零路径长，但右儿子的零路径长不能大于左儿子的零路径长。

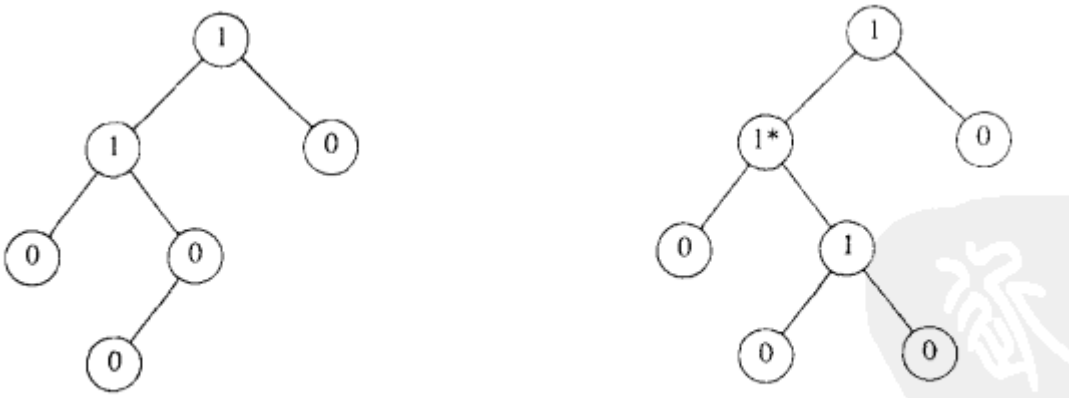


图 6-20 两棵树的零路径长，只有左边的树是左式树

1) 定理：在右路径上有 r 个节点的 2^r-1 左式树必然至少有个节点。

2) 左式堆的基本操作是合并。注意，插入只是合并的特殊情形。左式堆的合并共有四步：

1 如果有一棵树是空树，则返回另一棵树；否则递归地合并根节点较小的堆的右子树和根节点较大的堆。并、小、右、根、大

2 使形成的新堆作为较小堆的右子树。新、小、右

3　如果违反了左式堆的特性，交换两个子树的位置。

4　更新 Npl。

3)　左式堆的 DeleteMin 也很简单，就是把根节点删除，把两棵子树合并。

4)　代码：

合并左式堆的驱动例程是用于除去一些特殊情形并保证 H1 有较小根的驱动例程：

```
   PriorityQueue Merge( PriorityQueue H1, PriorityQueue H2 )
            {
/* 1*/     if( H1 == NULL )
/* 2*/        return H2;
/* 3*/     if( H2 == NULL )
/* 4*/        return H1;
/* 5*/     if( H1->Element < H2->Element )
/* 6*/        return Merge1( H1, H2 );
                 else
/* 7*/        return Merge1( H2, H1 );
            }
```

合并左式堆的实际例程：

```
            static PriorityQueue Merge1( PriorityQueue H1, PriorityQueue H2 )
            {
/* 1*/     if( H1->Left == NULL )  /* Single node */
/* 2*/        H1->Left = H2;      /* H1->Right is already NULL,
                                                    H1->Npl is already 0 */
                 else
                 {
/* 3*/        H1->Right = Merge( H1->Right, H2 );
/* 4*/        if( H1->Left->Npl < H1->Right->Npl )
/* 5*/           SwapChildren( H1 );

/* 6*/        H1->Npl = H1->Right->Npl + 1;
                 }
/* 7*/     return H1;
            }
```

6、斜堆是左式堆的自调节形式。斜堆和左式堆之间的关系类似于伸展树和 AVL 树间的关系。斜堆是具有堆序的二叉树，但不存在对树的结构限制。斜堆并不保留零路径长。斜堆的基本操作也是合并。

7、左式堆和斜堆主要用于堆合并。

8、二项队列

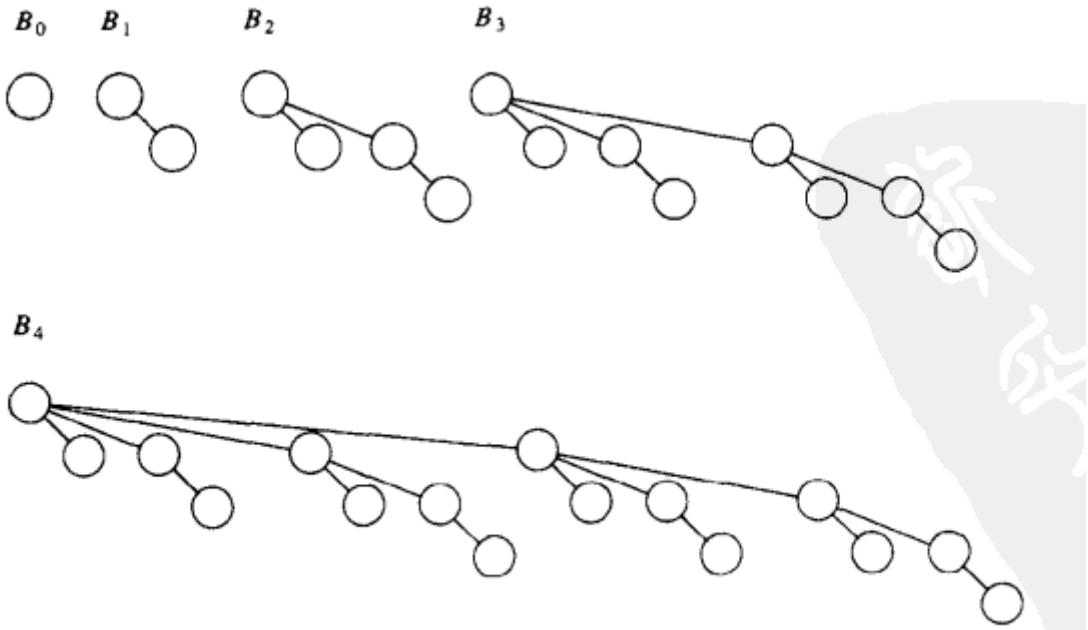


图 6-34　二项树 B_0、B_1、B_2、B_3 以及 B_4

1) 二项队列不是一棵堆序的树，而是堆序树的集合，称为森林。堆序树中的每一棵都是有约束的形式，叫做二项树。高度为 0 的二项树是一棵单节点树，高度为 k 的二项树 B_k 通过将一棵二项树 B_{k-1} 附接到另一棵二项树 B_{k-1} 的根上而构成。高度为 k 的二项树的节点个数为 2^k。二项队列是二项树的数组。二项树的每一个节点包含数据、第一个节点和右兄弟。　（二队、树集）

2) 二项队列的操作：

- 查找最小项：只需要查找每个二项树的根节点就可以了，因此时间复杂度为 O(log N)。
- 合并：通过把两个队列相加在一起完成。时间复杂度 O(log N)。
- 插入：插入也是一种合并。
- DeleteMin：首先找出具有最小根的二项树，令该树为 B_k，并令原始优先队列为 H。从 H 的树的森林中除去 B_k，形成新的二项树队列 H′，再除去 B_k 的根得到一些新的二项树 B_0、B_1……B_{k-1}，它们共同形成优先队列 H″。合并 H′和 H″操作结束。

3) 二项树的每一个节点都包含数据、第一个儿子以及右兄弟。二项树中诸儿子以递减次序排列。

```
struct BinNode          //二项树的定义
    {
        ElementType Element;
        Position    LeftChild;
        Position    NextSibling;
    };

    struct Collection         // 二项队列的定义
    {
        int CurrentSize;
        BinTree TheTrees[ MaxTrees ];
    };
```

第 7 章 排序

1、

2、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，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的。 （外、大、多）

3、内部排序是指待排序列完全存放在内存中所进行的排序过程，适合不太大的元素序列。 （内小）

4、排序：插入排序、希尔排序(shell sort)、堆排序、归并排序、快速排序、桶排序。

5、插入排序

1) 所谓插入排序法，就是检查第 P 个数字，如果在它的左边的数字比它大，进行交换，这个动作一直继续下去，直到这个数字的左边数字比它还要小，就可以停止了。 （插、左、大）

2) 说明：

1

初始	34	8	64	51	32	21	移动的位置
在 $p=1$ 之后	8	34	64	51	32	21	1
在 $p=2$ 之后	8	34	64	51	32	21	0
在 $p=3$ 之后	8	34	51	64	32	21	1
在 $p=4$ 之后	8	32	34	51	64	21	3
在 $p=5$ 之后	8	21	32	34	51	64	4

图 7-1 每趟后的插入排序

插入排序例程：

```
/* START: fig7_2.txt */
        void InsertionSort( ElementType A[ ], int N )
        {
            int j, P;
            ElementType Tmp;

/* 1*/    for( P = 1; P < N; P++ )
            {
/* 2*/      Tmp = A[ P ];          //注意：p=4 时，tmp=32
/* 3*/      for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )  //这个 for 找 tmp 的位置
/* 4*/        A[ j ] = A[ j - 1 ];
/* 5*/        A[ j ] = Tmp;
            }
        }
/* END */
```

6、希尔排序　　（希、分）

1) 希尔排序的实质就是分组插入排序，该方法又称缩小增量排序

2) 该方法的基本思想是：先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前两种方法有较大提高。

3) 例：以 n=10 的一个数组 49, 38, 65, 97, 26, 13, 27, 49, 55, 4 为例

第一次 gap = 10 / 2 = 5

49 38 65 97 26 13 27 49 55 4

1A 1B

2A 2B

3A 3B 4A 4B

5A 5B

1A,1B，2A,2B 等为分组标记，数字相同的表示在同一组，大写字母表示是该组的第几个元素，每次对同一组的数据进行直接插入排序。即分成了五组(49, 13) (38, 27) (65, 49) (97, 55) (26, 4)这样每组排序后就变成了(13, 49) (27, 38) (49, 65) (55, 97) (4, 26)，下同。

第二次 gap = 5 / 2 = 2

排序后

13 27 49 55 4 49 38 65 97 26

1A 1B 1C 1D 1E

2A 2B 2C 2D 2E

第三次 gap = 2 / 2 = 1

4 26 13 27 38 49 49 55 97 65

1A 1B 1C 1D 1E 1F 1G 1H 1I 1J

第四次 gap = 1 / 2 = 0 排序完成得到数组：

4 13 26 27 38 49 49 55 65 97

1 例程：

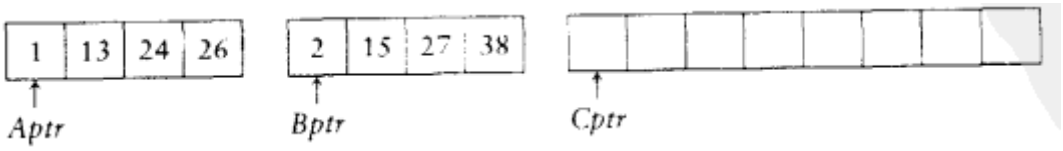
```
 void Shellsort( ElementType A[ ], int N )
          {
                int i, j, Increment;
                ElementType Tmp;

/* 1*/     for( Increment = N / 2; Increment > 0; Increment /= 2 )
/* 2*/        for( i = Increment; i < N; i++ )
                       {
/* 3*/           Tmp = A[ i ];
/* 4*/           for( j = i; j >= Increment; j -= Increment )
/* 5*/              if( Tmp < A[ j - Increment ] )
/* 6*/                 A[ j ] = A[ j - Increment ];
                                  else
/* 7*/                 break;
/* 8*/           A[ j ] = Tmp;
                       }
          }
/* END */
```

6、堆排序：堆分大顶堆和小顶堆（也叫大根堆和小根堆），大根堆指根节点是所有节点中最大值的堆，小顶堆指根节点是所有节点中最小值的堆。要实现堆排序首先要建立堆。堆排序就是把最大堆堆顶的最大数取出，将剩余的堆继续调整

为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。
堆排序实现：（堆、顶、调）

```
/* START: fig7_8.txt */

            #define LeftChild( i )  ( 2 * ( i ) + 1 )

            void PercDown( ElementType A[ ], int i, int N )
            {
                  int Child;
                  ElementType Tmp;

/* 1*/     for( Tmp = A[ i ]; LeftChild( i ) < N; i = Child )
                  {
/* 2*/        Child = LeftChild( i );
/* 3*/        if( Child != N - 1 && A[ Child + 1 ] > A[ Child ] )
/* 4*/           Child++;
/* 5*/        if( Tmp < A[ Child ] )
/* 6*/           A[ i ] = A[ Child ];
                        else
/* 7*/           break;
                  }
/* 8*/     A[ i ] =Tmp;
            }

            void Heapsort( ElementType A[ ], int N )
            {
                  int i;

/* 1*/     for( i = N / 2; i >= 0; i-- )  /* BuildHeap */
/* 2*/        PercDown( A, i, N );
/* 3*/     for( i = N - 1; i > 0; i-- )
                  {
/* 4*/        Swap( &A[ 0 ], &A[ i ] );  /* DeleteMax */
/* 5*/        PercDown( A, 0, i );
                  }
            }
/* END */
```

7、归并排序　　（归、序、三）



1) 基本的合并算法是取两个输入有序数组 A 和 B，一个输出数组 C，以及三个计数器 Aptr，Bptr，Cptr，它们初始置于对应数组的开始端。A[Aptr]和 B[Bptr]中的较小都被拷贝到 C 中的下一个位置，相关计算器向前推进一步。当两个表有一个用完时，则将另一个表中剩余部分拷贝到 C 中。

2) 归并排序例程：

```
/* START: fig7_10.txt */
        /* Lpos = start of left half, Rpos = start of right half */

        void Merge( ElementType A[ ], ElementType TmpArray[ ],
                            int Lpos, int Rpos, int RightEnd )
        {
            int i, LeftEnd, NumElements, TmpPos;

            LeftEnd = Rpos - 1;
            TmpPos = Lpos;
            NumElements = RightEnd - Lpos + 1;

            /* main loop */
            while( Lpos <= LeftEnd && Rpos <= RightEnd )
                if( A[ Lpos ] <= A[ Rpos ] )
                    TmpArray[ TmpPos++ ] = A[ Lpos++ ];
                else
                    TmpArray[ TmpPos++ ] = A[ Rpos++ ];

            while( Lpos <= LeftEnd )  /* Copy rest of first half */
                TmpArray[ TmpPos++ ] = A[ Lpos++ ];
            while( Rpos <= RightEnd ) /* Copy rest of second half */
                TmpArray[ TmpPos++ ] = A[ Rpos++ ];

            /* Copy TmpArray back */
            for( i = 0; i < NumElements; i++, RightEnd-- )
                A[ RightEnd ] = TmpArray[ RightEnd ];
        }
/* END */
```

8、快速排序：是在实践中最快的已知排序算法，平均运行时间是 O(N log N)。

1) 基本思想：先从数列中取出一个数作为基准数（即枢纽元）；然后将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。再对左右

区间进行排序，最后把排序后的区间合并在一起。具体过程如下：（枢纽元从 j 向前搜索、小于 key、从 i 向后搜索、大于 key）

1　设置两个变量 i、j，排序开始的时候：i=0，j=N-1；

2　以第一个数组元素作为枢纽元，赋值给 key，即 key=A[0]；

3　从 j 开始向前搜索，即由后开始向前搜索(j--)，找到第一个小于 key 的值 A[j]，将 A[j]和 A[i]互换；

4　从 i 开始向后搜索，即由前开始向后搜索(i++)，找到第一个大于 key 的 A[i]，将 A[i]和 A[j]互换；

5　重复第 3、4 步，直到 i=j；　(3,4 步中，没找到符合条件的值，即 3 中 A[j]不小于 key,4 中 A[i]不大于 key 的时候改变 j、i 的值，使得 j=j-1，i=i+1，直至找到为止。找到符合条件的值，进行交换的时候 i，　j 指针位置不变。另外，i==j 这一过程一定正好是 i+或 j-完成的时候，此时令循环结束）。

以一个数组作为示例，取区间第一个数为枢纽元。

0	1	2	3	4	5	6	7	8	9
72	6	57	88	60	42	83	73	48	85

初始时，i = 0; j = 9;　X = a[i] = 72

由于已经将 a[0]中的数保存到 X 中，可以理解成在数组 a[0]上挖了个坑，可以将其它数据填充到这来。

从 j 开始向前找一个比 X 小或等于 X 的数。当 j=8，符合条件，将 a[8]挖出再填到上一个坑 a[0]中。a[0]=a[8]; i++; 这样一个坑 a[0]就被搞定了，但又形成了一个新坑 a[8]，这怎么办了？简单，再找数字来填 a[8]这个坑。这次从 i 开始向后找一个大于 X 的数，当 i=3，符合条件，将 a[3]挖出再填到上一个坑中 a[8]=a[3]; j--;

数组变为：

0	1	2	3	4	5	6	7	8	9
48	6	57	88	60	42	83	73	88	85

依此类推。

2) 选取枢纽元　　（随机）

- 错误的方法：将第一个元素用作枢纽元。因为输入的数据可能是排序或者预排序的，所以这个方法不正确。
- 一种安全的方针是随机选取枢纽元。但随机数的生成一般是昂贵的，根本减少不了算法其余部分的平均运行时间。
- 三数中值分割法：使用左端、右端和中心位置的三个元素的中值作为枢纽元

3) 例程：

1 快速排序的驱动例程：

```
 /* START: fig7_12.txt */
        void Quicksort( ElementType A[ ], int N )
        {
            Qsort( A, 0, N - 1 );
        }
/* END */
```

2 实现三数中值分割方法的程序

```
 /* START: fig7_13.txt */
        /* Return median of Left, Center, and Right */
        /* Order these and hide the pivot */

        ElementType Median3( ElementType A[ ], int Left, int Right )
        {
            int Center = ( Left + Right ) / 2;

            if( A[ Left ] > A[ Center ] )
                Swap( &A[ Left ], &A[ Center ] );
            if( A[ Left ] > A[ Right ] )
                Swap( &A[ Left ], &A[ Right ] );
            if( A[ Center ] > A[ Right ] )
                Swap( &A[ Center ], &A[ Right ] );

            /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */

            Swap( &A[ Center ], &A[ Right - 1 ] );  /* Hide pivot */
            return A[ Right - 1 ];              /* Return pivot */
        }
/* END */
```

3　快速排序的主全程例程：

```
 /* START: fig7_14.txt */
          #define Cutoff ( 3 )

          void Qsort( ElementType A[ ], int Left, int Right )
          {
                int i, j;
                ElementType Pivot;

/* 1*/     if( Left + Cutoff <= Right )
                {
/* 2*/        Pivot = Median3( A, Left, Right );
/* 3*/        i = Left; j = Right - 1;
/* 4*/        for( ; ; )
                      {
/* 5*/          while( A[ ++i ] < Pivot ){ }
/* 6*/          while( A[ --j ] > Pivot ){ }
/* 7*/          if( i < j )
/* 8*/             Swap( &A[ i ], &A[ j ] );
                            else
/* 9*/             break;
                      }
/*10*/       Swap( &A[ i ], &A[ Right - 1 ] );  /* Restore pivot */

/*11*/       Qsort( A, Left, i - 1 );
/*12*/       Qsort( A, i + 1, Right );
                }
                else  /* Do an insertion sort on the subarray */
/*13*/       InsertionSort( A + Left, Right - Left + 1 );
          }
/* END */
```

9、在排序时交换两个数据的代价可能非常大，因为结构可能非常大，在这种情况下，实际的解法是让输入数组包含指向结构的指针，通过交换指针在进行排序。

10、桶式排序　　（桶式、低位）

1) 前提条件：第一，待排序数组元素为非负整数；第二，待排序数组元素有界。

2) 以 64、8、216、512、27、729、0、1、3343 和 125 为例：

1　第一步按照最低位优先进行桶式排序，以 10 为基：

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

图 3-24　第一趟基数排序后的桶

2　第二步按照次低位（即十位上的数字）进行排序得到：

8 1 0	216 512	729 27 125		343		64			
0	1	2	3	4	5	6	7	8	9

图 3-25　第二趟基数排序后的桶

3　然后按最高位进行排序，得到：

64 27 8 1 0	125	216	343		512		729		
0	1	2	3	4	5	6	7	8	9

图 3-26　最后一趟基数排序后的桶

4　对各桶进行合并。

11、迄今为止，我们考查过的所在算法都需要将输入数据装入内存。然而，存在一些应用程序，它们的输入数据太大装不进内存，这就需要外部排序。

12、外部排序：（分段排序--->两两归并）

1) 假设要对 900 MB 的数据进行排序，但机器上只有 100 MB 的可用内存时，外归并排序按如下方法操作：

a. 读入 100 MB 的数据至内存中，用某种常规方式（如快速排序、堆排序、归并排序等方法）在内存中完成排序。

b. 将排序完成的数据写入磁盘。

c. 重复步骤 1 和 2 直到所有的数据都存入了不同的 100 MB 的块（临时文件）中。在这个例子中，有 900 MB 数据，单个临时文件大小为 100 MB，所以会产生 9 个临时文件。

d. 读入每个临时文件（顺串）的前 10 MB（ = 100 MB / (9 块 + 1)）的数据放入内存中的输入缓冲区，最后的 10 MB 作为输出缓冲区。两两合并的关键是这里：读入每个临时文件的前 10M。每个临时文件前 10M 是最大或最小的，所以对这些文件进行排序可以得到一个完整的排序。（9+1、输入缓冲区、关键是、读入前 10M）

e. 执行九路归并算法，将结果输出到输出缓冲区。一旦输出缓冲区满，将缓冲区中的数据写出至目标文件，清空缓冲区。一旦 9 个输入缓冲区中的一个变空，就从这个缓冲区关联的文件，读入下一个 10M 数据，除非这个文件已读完。这是“外归并排序”能在主存外完成排序的关键步骤 -- 因为“归并算法”(merge algorithm)对每一个大块只是顺序地做一轮访问(进行归并)，每个大块不用完全载入主存。

2) 我们把每组排过序的记录叫顺串。

	堆数组中的 3 个元素			输出	读入的下一元素
	H[0]	H[1]	H[2]		
顺串 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	End of Run.	Rebuild Heap
顺串 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	15*
	58	99	15*	58	end of tape
	99		15*	99	
			15*	End of Run.	Rebuild Heap
顺串 3	15			15	

图 7-18 顺串构建的例

在这个例子中，得到 3 个顺串，即 3 个输出数组。

替换选择并不比标准算法更好。然而，输入数据常常从排序或几乎从排序开始，此时替换选择仅仅产生少数非常长的顺串，这种类型的输入通常要进行外部排序，这就使得替换选择具有特殊的价值。

第 8 章 并查集

1. 并查集是一种树型的数据结构，但并不能像链表那样直接存取，而是使用函数来进行存取，其实质有点像树概念中的森林。查找使用 find，而插入使用的是 Union，也就是将要插入的数据当作一个集合，将两个集合合并。
2. 并查集要满足一个等价关系，即满足自反性、对称性、传递性。

- 自反性：这个关系必须对自己有效。
- 对称性：A 向 B 有这种关系，B 向 A 必须也有这种关系。
- 传递性：如果 A 和 B 之间有关系 R，B 和 C 之间有关系 R，A 和 C 之间也必须有这种关系 R。

例如，大学宿舍就是一个并查集。自反性指 A 自己就是并查集的一员。对称是指 A 是 B 的舍友，则 B 也是 A 的舍友。传递性是指 A 和 B 是舍友，B 和 C 是舍友，则 A 和 C 也是舍友。

3. 并查集：

- Find：确定元素属于哪一个子集。
- Union：将两个子集合并成同一个集合。

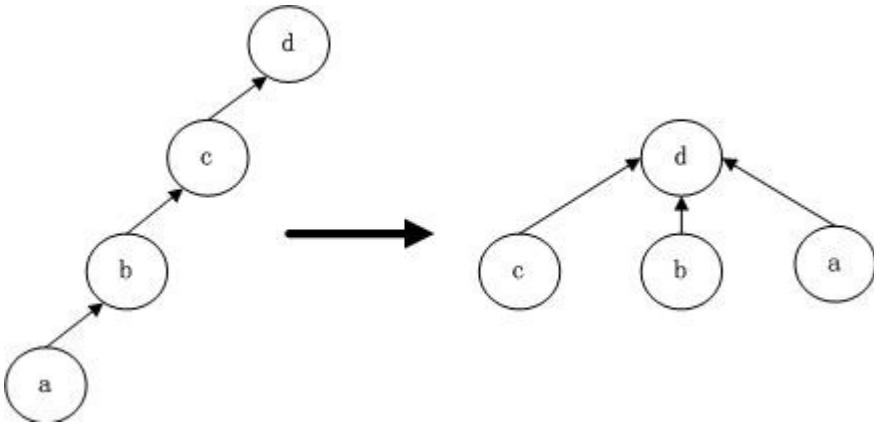
```
#ifndef _DisjSet_H

typedef int DisjSet[ NumSets + 1 ];
typedef int SetType;
typedef int ElementType;

void Initilialize( DisjSet S );
void SetUnion( DisjSet S, SetType Root1, SetType Root2 );
SetType Find( ElementType X, DisjSet S );

#endif /* _DisjSet_H */
```

Initilize()：把每一个元素初始化为一个集合。初始化后每一个元素的父亲节点是它本身，每一个元素的祖先节点也是它本身。

SetUnion()：合并 x,y 所在的两个集合。合并两个不相交集合操作很简单：利用 Find_Set 找到其中两个集合的祖先，将一个集合的祖先指向另一个集合的祖先。

Find()：查找一个元素所在的集合。查找一个元素所在的集合，其精髓是找到这个元素所在集合的祖先！这个才是并查集判断和合并的最终依据。

4. 路径压缩指将每一个高度 N 的树转化为高度为 1 的树。类似下列的行为：



5. 并查集的简单应用：若两公司合并，要对员工信息进行合并，老板 A 变高管 A。若之前某个基层员工的最终老板是 A，则将该员工所有的同事以及所有的领导都归到 A 的名下，这就是一种等价关系。

第 9 章 图论算法

1、图论是一种表示 "多对多" 的关系。

2、图可以分为有向图和无向图，也可以分为有权图和无权图。

- (v, w) 表示无向边，即 v 和 w 是互通的
- <v, w> 表示有向边，该边始于 v，终于 w

3、图中的顶点有度的概念：

- 度(Degree)：所有与它连接点的个数之和
- 入度(Indegree)：存在于有向图中，所有接入该点的边数之和
- 出度(Outdegree)：存在于有向图中，所有接出该点的边数之和

4、图在程序中的表示一般有两种方式：

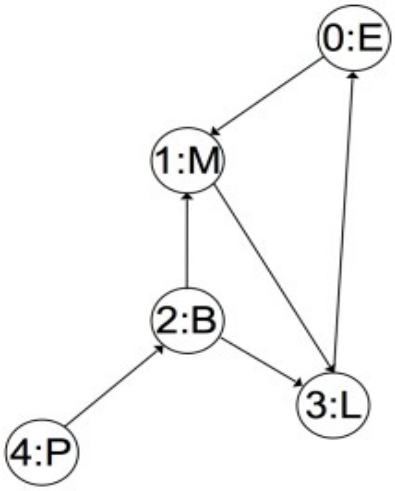
(1) 邻接矩阵

- 在一个无权图中，矩阵坐标中每个位置值为 1 代表两个点是相连的，0 表示两点是不相连的
- 在一个有权图中，矩阵坐标中每个位置值代表该两点之间的权重，0 表示该两点不相连

(2) 邻接链表

- 对于每个点，存储着一个链表，用来指向所有与该点直接相连的点
- 对于有权图来说，链表中元素值对应着权重

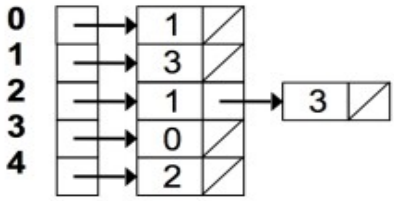
例如，在有向无权表示如下：



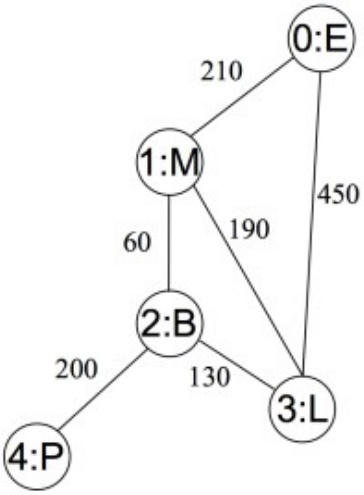
Adjacency matrix

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	0	1	0
2	0	1	0	1	0
3	1	0	0	0	0
4	0	0	1	0	0

Adjacency list

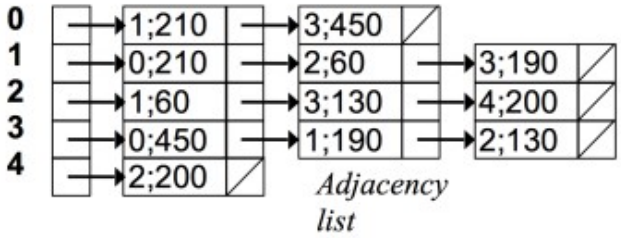


在有权无向图中：



	0	1	2	3	4
0	0	210	0	450	0
1	210	0	60	190	0
2	0	60	0	130	200
3	450	190	130	0	0
4	0	0	200	0	0

Adjacency matrix



Adjacency list

关键点：

(1) 如何表示向：

- 邻接矩阵：纵坐标表示起点，横坐标表示终点。
- 邻接链表：数组的值表示起点，数组项指向的链表值表示终点，每一个链表值表示该数组项的一条边。

(2) 如何表示权：

- 邻接矩阵：权直接标在矩阵中。
- 邻接链表：如果是无权图，使用“点号”表示；如果是有权图，使用“点号;权”的方式表示。

5、图的遍历就是要找出图中所有的点，一般有以下两种方法：

- 深度优先遍历（DFS）
- 广度优先搜索：(BFS)

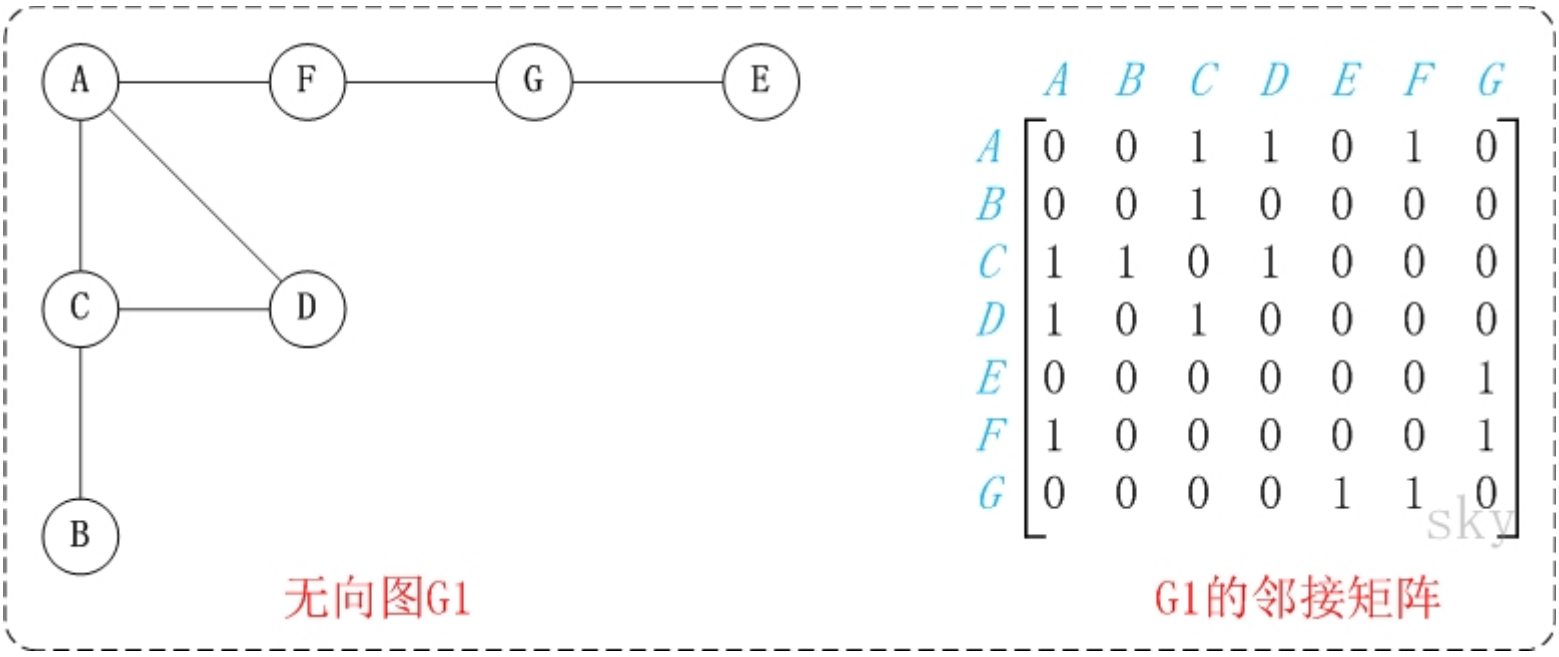
6、存在着一条从顶点 A 到顶点 B 的路径，路径 AB 称为环。

7、邻接矩阵无向图是指通过邻接矩阵表示的无向图。

1) 基本定义：

```
// 邻接矩阵
typedef struct _graph
{
    char vexs[MAX];        // 顶点集合
    int vexnum;            // 顶点数
    int edgnum;            // 边数
    int matrix[MAX][MAX]; // 邻接矩阵
}Graph, *PGraph;
```

2) 例：



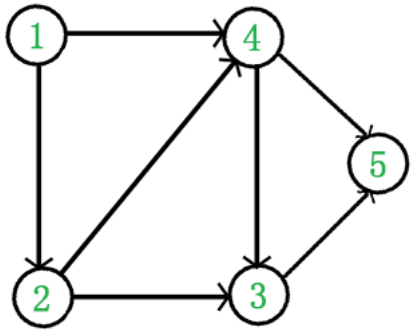
上面的图 G1 包含了"A,B,C,D,E,F,G"共 7 个顶点，而且包含了"(A,C),(A,D),(A,F),(B,C),(C,D),(E,G),(F,G)"共 7 条边。由于这是无向图，所以边(A,C)和边(C,A)是同一条边；这里列举边时，是按照字母先后顺序列举的。

上图右边的矩阵是 G1 在内存中的邻接矩阵示意图。A[i][j]=1 表示第 i 个顶点与第 j 个顶点是邻接点，A[i][j]=0 则表示它们不是邻接点；而 A[i][j]表示的是第 i 行第 j 列的值；例如，A[1,2]=1，表示第 1 个顶点(即顶点 B)和第 2 个顶点(C)是邻

接点。

8、拓扑排序是对有向无圈（即回路）图的顶点的一种排序，它使得如果存在一条从 v_i 到 v_j 的路径，那么在排序中 v_j 出现在 v_i 后面。如果图中含有圈，那么拓扑排序是不可能的，此外排序不必是惟一的，任何合理的排序都是可以的。

例如，下面这个图：

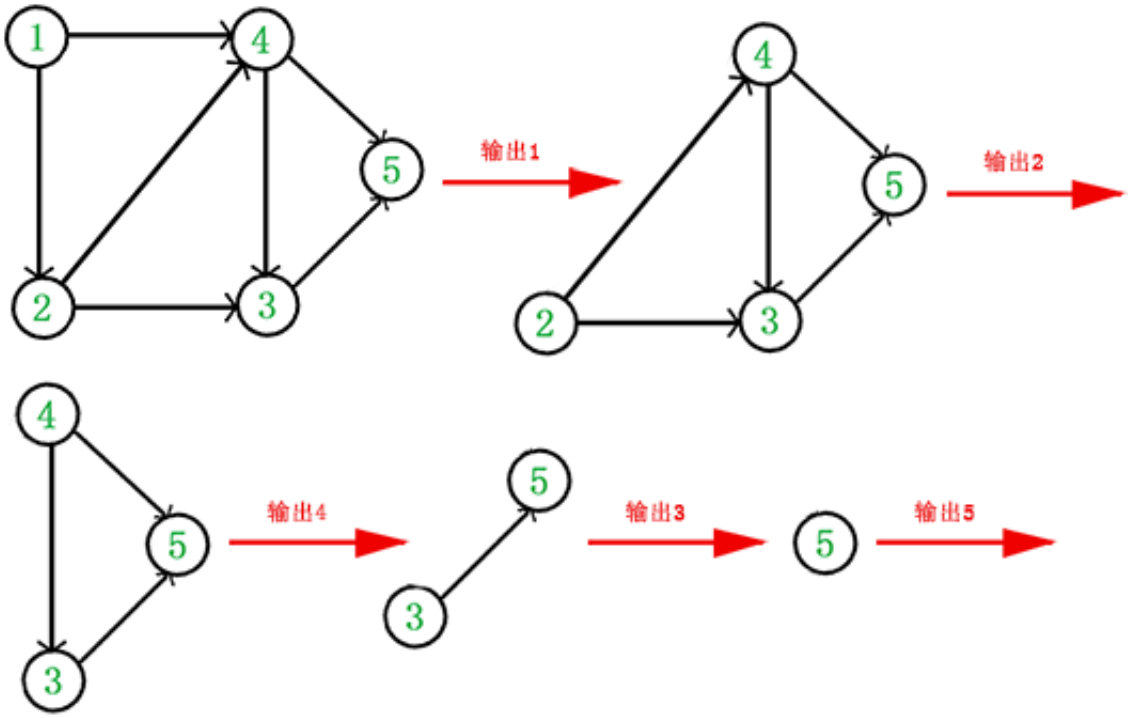


它是一个 DAG 图，那么如何写出它的拓扑排序呢？这里说一种比较常用的方法：

1　从 DAG 图中选择一个没有前驱（即入度为 0）的顶点并输出。

2　从图中删除该顶点和所有以它为起点的有向边。

3　重复 1 和 2 直到当前的 DAG 图为空或当前图中不存在无前驱的顶点为止。后一种情况说明有向图中必然存在环。



于是，得到拓扑排序后的结果是 { 1, 2, 4, 3, 5 }。

通常，一个有向无环图可以有一个或多个拓扑排序序列。

应用：选课就是应用拓扑排序的例子。学习数据结构前必须学习 C 语言等等每门课程相当于有向图中的一个顶点，而连接顶点之间的有向边就是课程学习的先后关系。

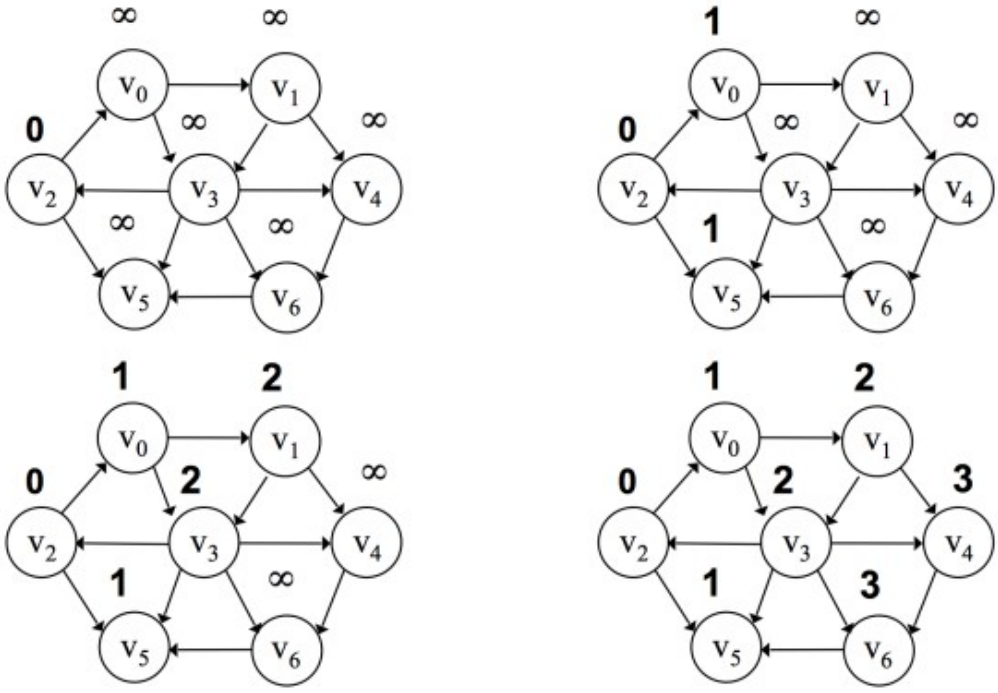
9、无权图的最短路径基本步骤：

(1) 将所有点的距离 d 设为无穷大

(2) 挑选初始点 s，将 ds 设为 0，将 shortest 设为 0

(3) 找到所有距离为 d 为 shortest 的点，查找他们的邻接链表的下一个顶点 w，如果 dw 的值为无穷大，则将 dw 设为 shortest + 1

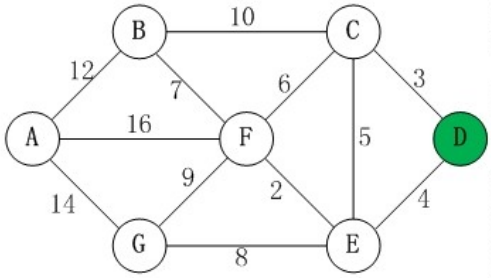
(4) 增加 shortest 的值，重复步骤 3，直到没有顶点的距离值为无穷大



10、在有权图中，常见的最短路径算法有 Dijkstra 算法 Floyd 算法。

11、Dijkstra 算法适用于权值为正的的图。

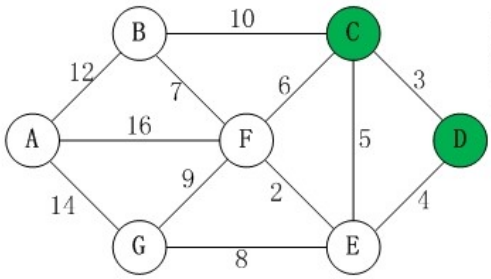
1) Dijkstra 算法图解：



第1步：
选取顶点D

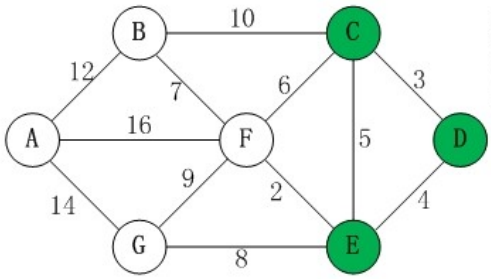
S={D(0)}
U={A(∞),B(∞),C(3),E(4),F(∞),G(∞)}

注：
(01) S是已计算出最短路径的定点的集合
(02) U是未计算出最短路径的定点的集合
(03) C(3)表示顶点C到起点D的最短距离是3



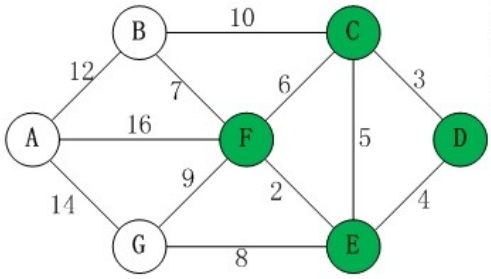
第2步：
选取顶点C

S={D(0),C(3)}
U={A(∞),B(23),E(4),F(9),G(∞)}



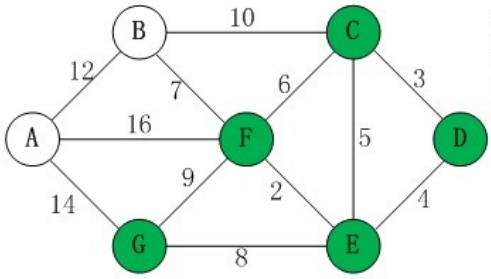
第3步：
选取顶点E

S={D(0),C(3),E(4)}
U={A(∞),B(23),F(6),G(12)}



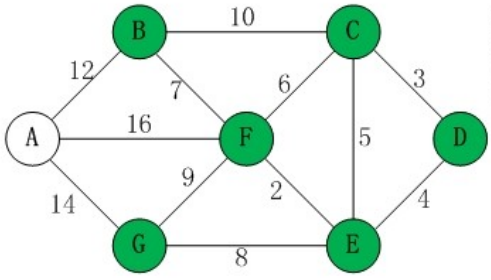
第4步：
选取顶点F

S={D(0),C(3),E(4),F(6)}
U={A(22),B(13),G(12)}



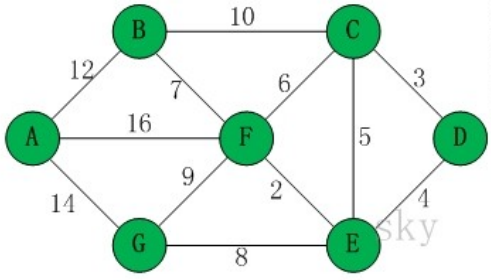
第5步：
选取顶点G

S={D(0),C(3),E(4),F(6),G(12)}
U={A(22),B(13)}



第6步：
选取顶点B

S={D(0),C(3),E(4),F(6),G(12),B(13)}
U={A(22)}



第7步：
选取顶点A

S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}

注意：如果图具有负边值，那么 Dijkstra 算法行不通。

12、佛洛伊德 Floyd 算法正确处理有向图或负权（但不可存在负权回路)的最短路径问题。

1) 从任意节点 i 到任意节点 j 的最短路径不外乎 2 种可能：

- 是直接从 i 到 j
- 是从 i 经过若干个节点 k 到 j

2) 假设 Dis(i,j) 为节点 u 到节点 v 的最短路径的距离，对于每一个节点 k，我们检查 Dis(i,k) + Dis(k,j) < Dis(i,j) 是否成立，如果成立，证明从 i 到 k 再到 j 的路径比 i 直接到 j 的路径短，我们便设置 Dis(i,j) = Dis(i,k) + Dis(k,j)，这样一来，当我们遍历完所有节点 k，Dis(i,j) 中记录的便是 i 到 j 的最短路径的距离。

3) 时间复杂度： $O(n^3)$

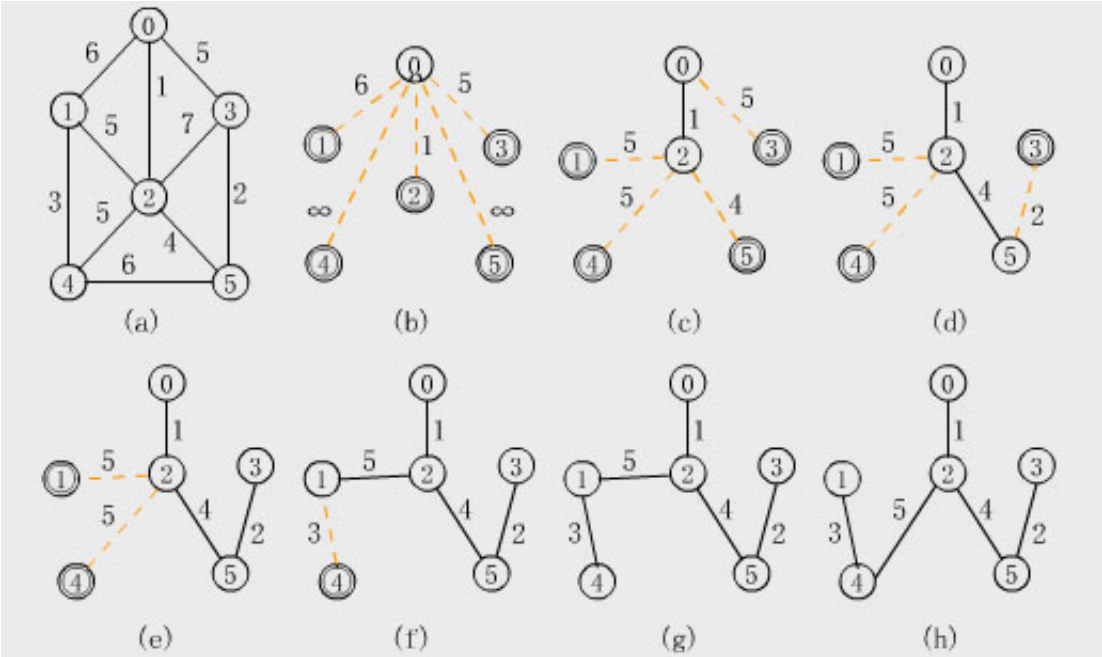
```
for(int k=0; k<n; k++) {
    for(i=0; i<n; i++) {
        for(j=0; j<n; j++)
            if(A[i][j]>(A[i][k]+A[k][j])) {
                A[i][j]=A[i][k]+A[k][j];
                path[i][j]=k;
            }
    }
}
```

13、大体上说来，一个无向图 G 的最小生成树就是由该图的那些连接 G 的所有顶点的边构成的树，且其总价值最低。

构造最小生成树一般使用贪心策略，有 prime 算法和 kruskal 算法

14、Prim 算法：首先以一个结点作为最小生成树的初始结点，然后以迭代的方式找出与最小生成树中各结点权重最小边，并加入到最小生成树中。加入之后如果产生回路则跳过这条边，选择下一个结点。当所有结点都加入到最小生成树中之后，就找出了连通图中的最小生成树了。

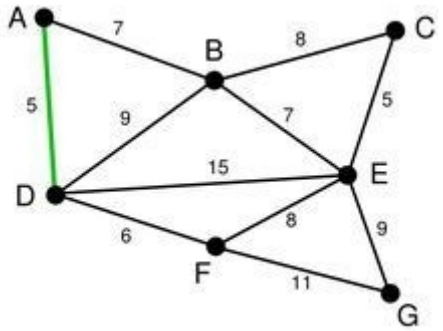
最小生成树实际上就是求出初始结点到所有点的的最小距离：



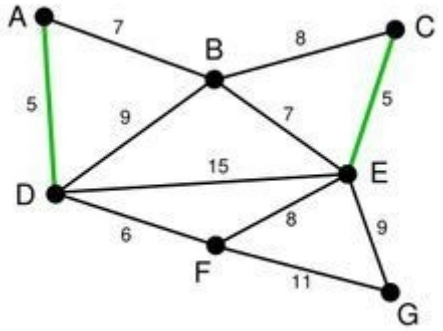
Prim算法构造最小生成树的过程

15、 Kruskal 算法与 Prim 算法的不同之处在于，Kruskal 在找最小生成树结点之前，需要对所有权重边做从小到大排序。将排序好的权重边依次加入到最小生成树中，如果加入时产生回路就跳过这条边，加入下一条边。当所有结点都加入到最小生成树中之后，就找出了最小生成树。

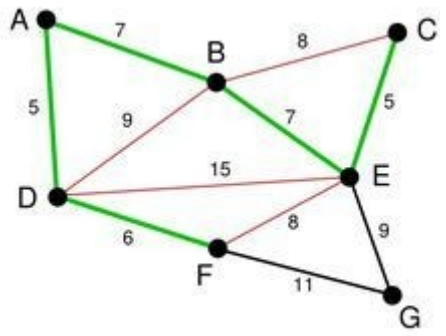
例如，要从 A 点连接到 G 点：



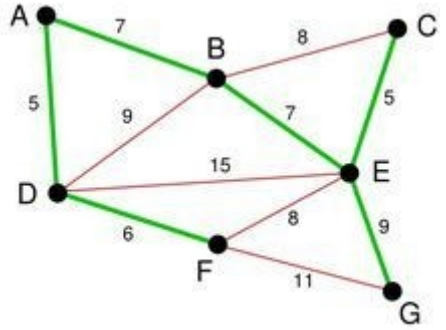
在对所有边进行排序之后，我们得到一个边集合，从边集合中取出最小权的边 AD



剩下的边中寻找。我们找到了 CE。这里边的权重也是 5，依次类推我们找到了 6,7,7



尽管现在长度为 8 的边是最小的未选择的边。但是他们已经连通了。



最后就剩下 EG 和 FG 了。当然我们选择了 EG

16、Kruskal 算法在效率上要比 Prim 算法快，因为 Kruskal 只需要对权重边做一次排序，而 Prim 算法则需要做多次排序。

第 10 章 算法设计技巧

1、所谓贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。第 9 章的 Dijkstra 算法、Kruskal 算法和 Prim 算法都是贪婪算法。

2、贪婪算法分阶段地工作。在每个阶段，可以认为所作决定是好的，而不考虑将来的返璞归真。一般地说，这意味着选择的是某个局部的最优。

3、分治算法

1) 设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

2) 分治策略是：对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

3) 可使用分治法求解的一些经典问题：

- 二分搜索；
- 合并排序
- 快速排序

4、动态规划：

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

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

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

4) 能采用动态规划求解的问题的一般要具有 3 个性质：

- 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。
- 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。
- 有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

5) 示例：设有由 n 个不相同的整数组成的数列 b[n],若有下标 $i_1 < i_2 < \cdots < i_L$ 且 $b[i_1] < b[i_2] < \cdots < b[i_L]$，则称存在一个长度为 L 的不下降序列。求最长了不下降序列。

13,7,9,16,38,24,37,18,44,19,21,22,63,15

思路：假如要求得某一段的最优，就要想更小段的最优怎么求，再看看由最小段的最优能否扩大推广到最大段的最优。

解决方案：

a) 先从倒数第二项 63 算起，在它的后面仅有一项，因此仅作一次比较，因

为63>15，所以从63出发，不作任何链接，长度还是为1。

b) 再看倒数第三项22，在它的后面有2项，因此必须要在这2项当中找出比22大，长度又是最长的数值作为链接，由于只有22<63,所以修改22的长度为2，即自身长度加上所链接数值的长度，并修改链接位置为13，也就是63的下标。

c) 再看倒数第四项21，在它的后面有3项，因此必须要在这3项当中找出比21大，长度又是最长的数值作为链接(注意:是长度)，很容易看出，数值22满足该条件，因此，修改21的长度为3，并修改链接位置为12，即22的序列下标。

d) 依次类推

5、回溯法：是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

1) 基本思想：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。而若使用回溯法求任一个解时只要搜索到问题的一个解就可以结束。

2) 算法框架

- 非递归回溯框架

```
int a[n],i;
初始化数组 a[];
i = 1;
while (i>0(有路可走)  and  (未达到目标))  // 还未回溯到头
{
      if(i > n)                    // 搜索到叶结点
      {
            搜索到一个解，输出；
      }
     else                        // 处理第 i 个元素
      {
            a[i]第一个可能的值；
            while(a[i]在不满足约束条件且在搜索空间内)
            {
                  a[i]下一个可能的值；
            }
            if(a[i]在搜索空间内)
```

```
            {
                标识占用的资源；
                i = i+1;                    // 扩展下一个结点
            }
            else
            {
                清理所占的状态空间；                    // 回溯
               i = i –1;
            }
        }
}
```

- 递归

```
int a[n];
  try(int i)
  {
      if(i>n)
          输出结果;
      else
      {
          for(j = 下界; j <= 上界; j=j+1)  // 枚举 i 所有可能的路径
          {
              if(fun(j))            // 满足限界函数和约束条件
              {
                  a[i] = j;
                  ...                  // 其他操作
                  try(i+1);
                  回溯前的清理工作（如 a[i]置空值等）;
                  }
              }
        }
  }
```

第 12 章 高级数据结构及其实现

1、R-B Tree，全称是 Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

2、红黑树的特性:

(1) 每个节点或者是黑色，或者是红色。

(2) 根节点是黑色。

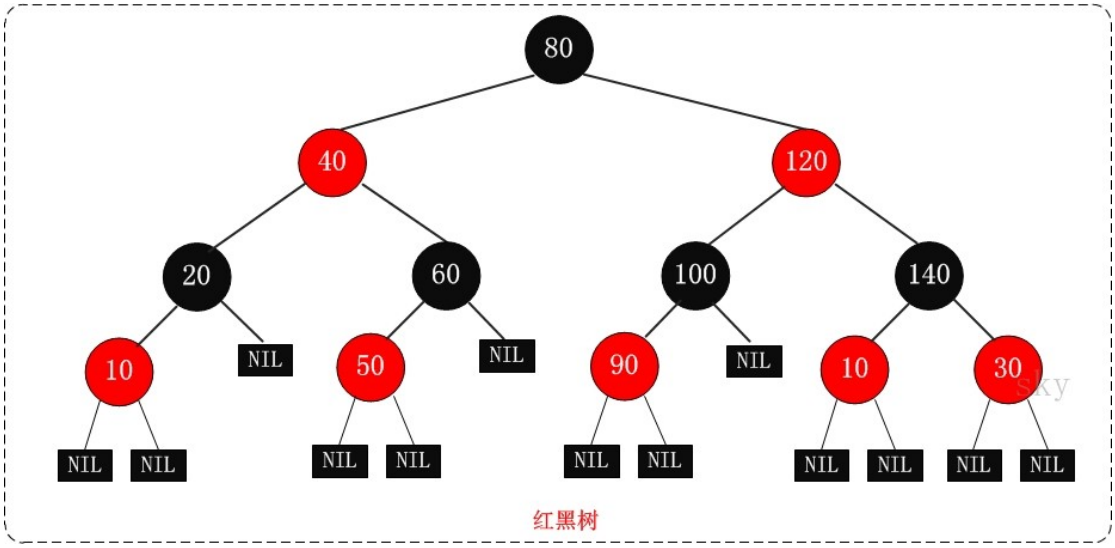
(3) 每个叶子节点（NIL）是黑色。

(4) 如果一个节点是红色的，则它的子节点必须是黑色的。

(5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：

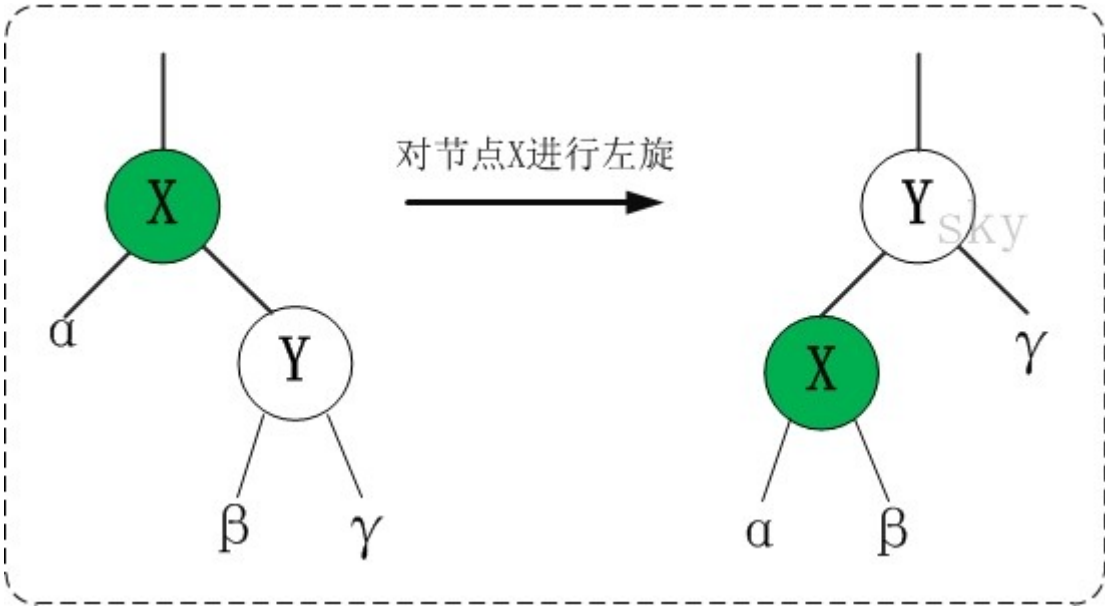
- 特性(3)中的叶子节点，是只为空(NIL 或 null)的节点。
- 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。



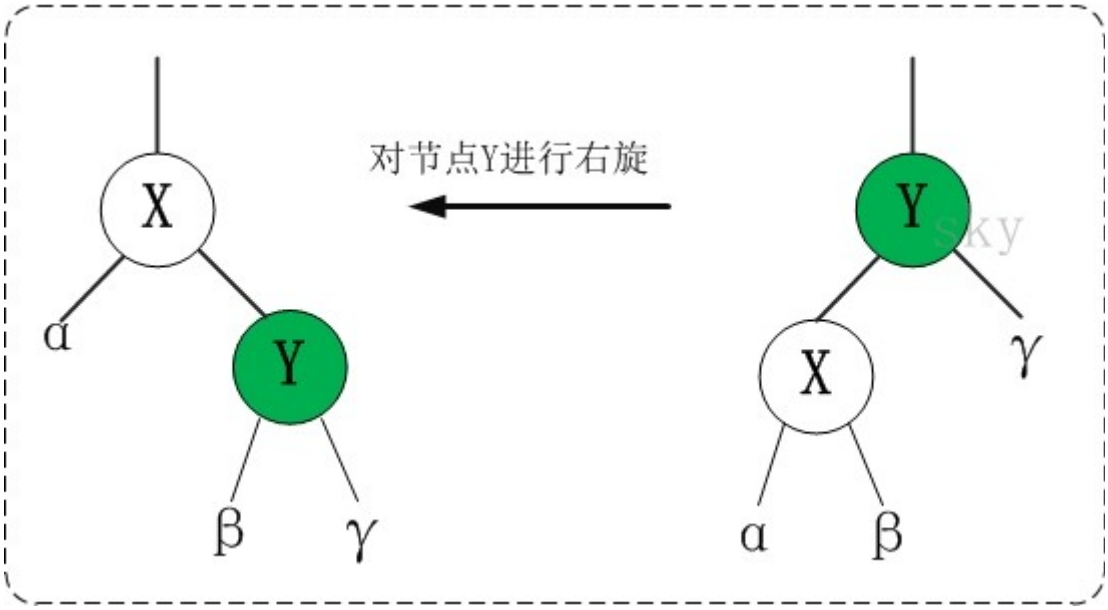
3、红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是 O(lgn)，效率非常之高。例如，Java 集合中的 TreeSet 和 TreeMap，C++ STL 中的 set、map，以及 Linux 虚拟内存的管理，都是通过红黑树去实现的。

4、红黑树的基本操作是添加、删除。在对红黑树进行添加或删除之后，树的特性会发生变化，需要用旋转方法来调整红黑树的特性。

- 左旋



- 右旋



5、添加操作：

(1) 第一步: 将红黑树当作一颗二叉查找树，将节点插入。

(2) 第二步：将插入的节点着色为"红色"。

(3) 第三步: 通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。（过于复杂，应用到再谷歌查询）

6、删除操作：

(1) 第一步：将红黑树当作一颗二叉查找树，将节点删除。

(2) 第二步：通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

7、红黑树追求的是局部平衡而不是 AVL 树中的非常严格的平衡。红黑树是一个更高效的检索二叉树，因此常常用来实现关联数组。

说明

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						