



算法分析(3)

重要的数据结构



二分查找 (Binary Search)

```
template <class T>
int BinarySearch(T a[], const T& x, int n)
{ // Search a[0] <= a[1] <= ... <= a[n-1] for x.
  // Return position if found; return -1 otherwise.
  int left = 0; int right = n - 1;
  while (left <= right) {
    int middle = (left + right) / 2;
    if (x == a[middle]) return middle;
    if (x > a[middle]) left = middle + 1;
    else right = middle - 1;
  }
  return -1; // x not found
}
```

Program 2.30 Binary search

1. 判定树

- 判定树适合于描述具有层次结构的数据

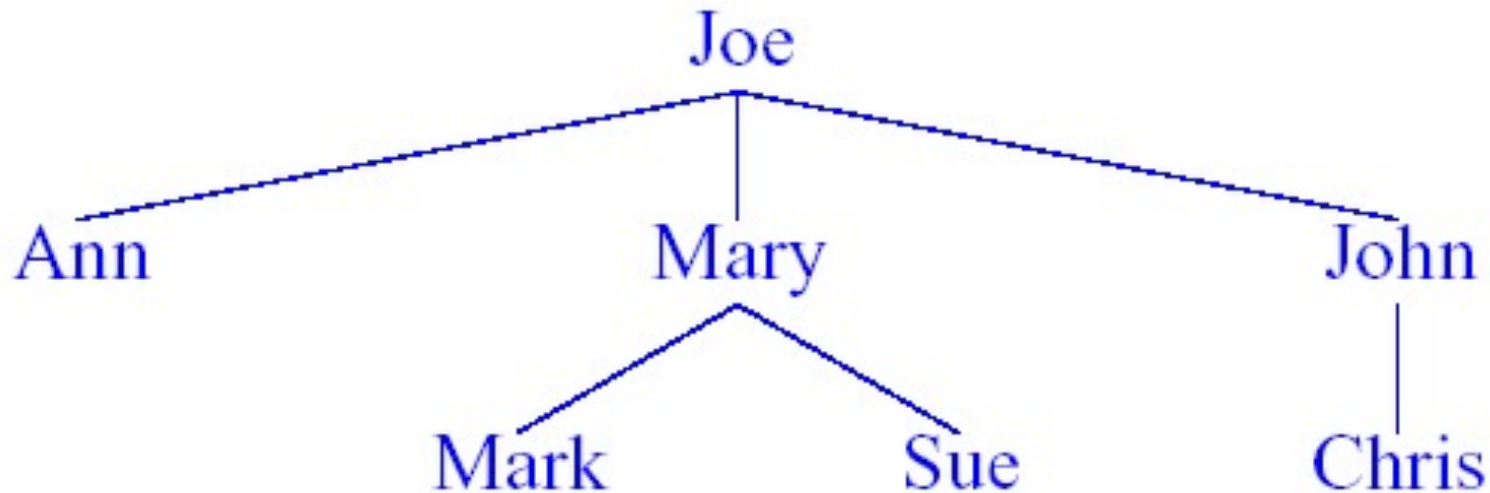


Figure 8.1 Descendants of Joe



树的定义

- 树(tree) t 是一个**非空的有限元素的集合**. 其中一个元素为根(root), 余下的元素(如果有的话)组成 t 的子树(subtree).
- **层数(Level)**: 指定树根的层数为1, 其子节点(如果有)的层数为2, 子节点的子节点的层数为3, 等等。
- **节点的度(degree of an element)**是指其孩子的个数。
- **树的度(degree of a tree)**是其元素度的最大值。



二叉树

- 二叉树(binary tree) t 是有限个元素的集合(可以为空).当二叉树非空时,其中有一个称为根的元素.余下的元素(如果有的话)被组成2个二叉树,分别称为 t 的左子树和右子树.
- 二叉树的**高度**(height)或深度(depth)是指该二叉树的层数.
- 从根节点到每个节点有唯一一条路径,路径上的边数称为该路径的**长度**.



二叉树的数学性质

- **性质1** : 包含 n ($n > 0$)个元素的树的边数为 $n-1$.
- **性质2** : 若二叉树的高度为 h , $h \geq 0$, 则该二叉树最少有 h 个元素, 最多有 $2^h - 1$ 个元素.
- **性质3** : 包含 n 个元素的二叉树的高度最大为 n , 最小为 $\lceil \log_2(n+1) \rceil$:

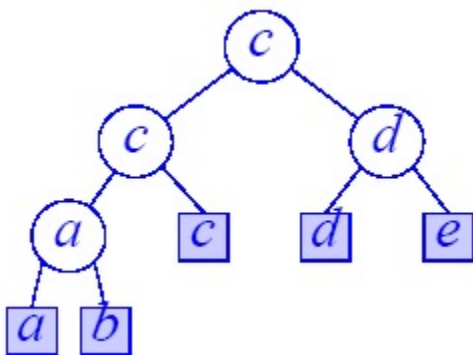
$$n \leq 2^h - 1 \Rightarrow 2^h \geq n + 1$$

所以, $h \geq \log_2(n+1)$, 即:

$$h \geq \lceil \log_2(n+1) \rceil$$

续

- **扩充的二叉树**: 补齐外节点的二叉树. 设 n 为其内节点数, 则外节点数为 $n+1$;



- 有 n 个节点的扩充二叉树, 当外节点分布在相邻两层时其深度, 外路和内路长度达到最小(**平衡原理**).
- 设扩充前深度为 k , 则有:

$$2^{k-1} \leq n < 2^k \Rightarrow k = \lfloor \log n \rfloor + 1$$



续

- 设 **E** 为根节点到外节点的路径长度之和; **I** 为根节点到内节点的路径长度之和.
- 对有 n 个内节点的扩充二叉树, 用归纳法可证明: **$E=I+2n$** (练习)
- 假定扩充二叉树的外节点分布在相邻两层
则: $E=m(k-1)+(n+1-m)k$, 其中 m 为第 k 层上的外节点数. 又因外节点数 $n+1=m+2(2^{k-1}-m)$,
所以: $m=2^k-(n+1)$, **$E=(k+1)(n+1)-2^k$**



二分查找的平均复杂度

- 设 I 为内路长度, E 为外路长度,则:
平均失败查找次数 $U(n)=E/(n+1)$;
平均成功查找次数 $S(n)=(I+n)/n$
- 平均失败查找次数 $U(n)=E/(n+1)=\Theta(\log n)$
- 平均成功查找次数 $S(n)=(I+n)/n=\Theta(\log n)$

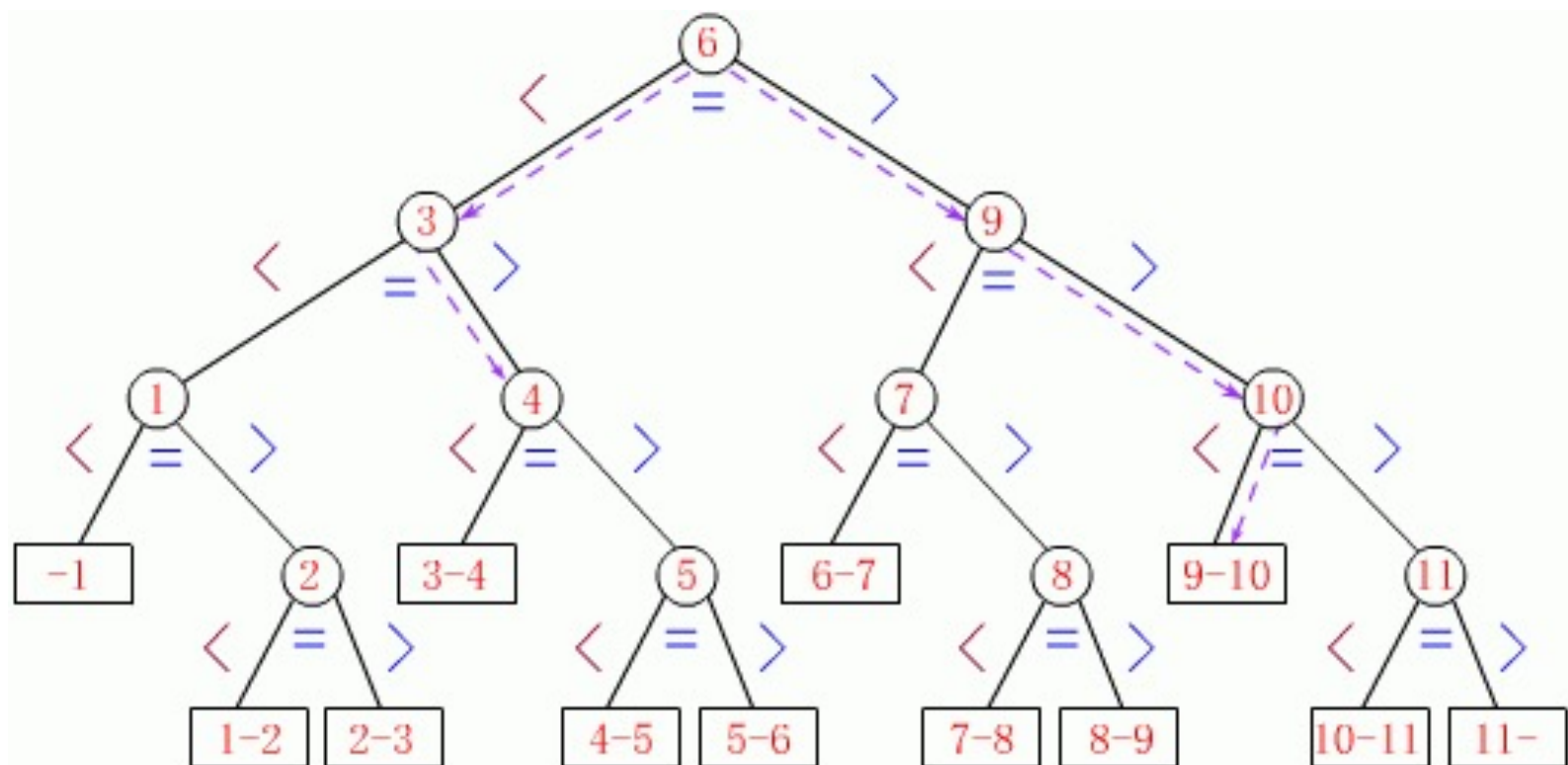


二分查找判定树

- 二分查找过程可用二叉树来描述：把当前查找区间的中间位置上的结点作为根，左子表和右子表中的结点分别作为根的左子树和右子树。由此得到的二叉树，称为描述二分查找的判定树(Decision Tree)或比较树(Comparison Tree)。
- 注意：

判定树的形态只与结点个数 n 相关，而与输入实例中 $R[1..n]$ 关键字的取值无关。

- 具有11个结点的有序表可用下图所示的判定树来表示



R[1..11]的二分查找的判定树(n=11)



■ 二分查找判定树的组成

- ①圆结点即树中的内部结点。树中圆结点内的数字表示该结点在有序表中的位置。
- ②外部结点：圆结点中的所有空指针均用一个虚拟的方形结点来取代，即外部结点。
- ③树中某结点*i*与其左(右)孩子连接的左(右)分支上的标记" $<$ "、" $($ "、" $>$ "、" $)$ "表示：当待查关键字 $K < R[i].key$ ($K > R[i].key$)时，应走左(右)分支到达*i*的左(右)孩子，将该孩子的关键字进一步和*K*比较。若相等，则查找过程结束返回，否则继续将*K*与树中更下一层的结点比较。



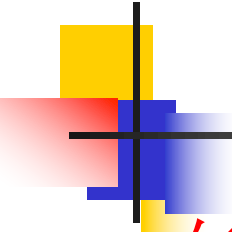
续

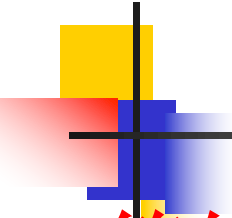
- 二分查找算法的判定树为有 n 个内节点的扩充二叉树而且叶节点分布在相邻两层:二分查找算法最坏和平均情形的时间复杂度为 $\Theta(\log n)$
- 基于关键字比较的有序表查找算法至少要做 $\Theta(\log n)$ 次比较

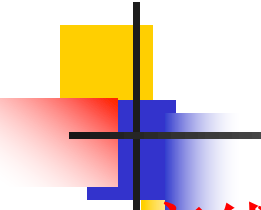


2. 等价类问题

- 假定有一个具有 n 个元素的集合 $U = \{1, 2, \dots, n\}$, 另有一个具有 r 个关系的集合 $R = \{(i_1, j_1), (i_2, j_2), \dots, (i_r, j_r)\}$ 。关系 R 是一个**等价关系** (equivalence relation), 当且仅当如下条件为真时成立:
 - 对于所有的 a , 有 $(a, a) \in R$ 时 (即关系是反身的)
 - 当且仅当 $(b, a) \in R$ 时 $(a, b) \in R$ (即关系是对称的)
 - 若 $(a, b) \in R$ 且 $(b, c) \in R$, 则有 $(a, c) \in R$ (即关系是传递的)

- 
- 在给出等价关系 R 时，我们通常会忽略其中的某些关系，这些关系可以利用等价关系的反身、对称和传递属性来获得
 - 例3-3 假定 $n=14$, $R = \{ (1, 11), (7, 11), (2, 12), (12, 8), (11, 12), (3, 13), (4, 13), (13, 14), (14, 9), (5, 14), (6, 10) \}$ 。
 - 我们忽略了所有形如 (a, a) 的关系，因为按照反身属性，这些关系是隐含的。同样也忽略了所有的对称关系。
 - 比如 $(1, 11) \in R$ ，按对称属性应有 $(11, 1) \in R$ 。其他被忽略的关系是由传递属性可以得到的属性。例如根据 $(7, 11)$ 和 $(11, 12)$ ，应有 $(7, 12) \in R$ 。

- 
- **等价类 (equivalence class)** 是指相互等价的元素的最大集合。
 - “最大”意味着不存在类以外的元素，与类内部的元素等价。
 - 考察例3-3中的等价关系。
 - 由于元素1与11，11与12是等价的，因此，元素1, 11, 12是等价的，它们应属于同一个等价类。不过，这三个元素还不能构成一个等价类，因为还有其他的元素与它们等价（如7）。所以{ 1, 11, 12 }不是等价元素的最大集合。集合{ 1, 2, 7, 8, 11, 12 }才是一个等价类。
 - 关系R还定义了另外两个等价类：{ 3, 4, 5, 9, 13, 14 } 和{ 6, 10 }。

- 
- **离线等价类** (offline equivalence class) : 已知 n 和 R , 确定所有的等价类。
 - **在线等价类** (online equivalence class) : 初始时有 n 个元素, 每个元素都属于一个独立的等价类。需要执行以下的操作
 - **Combine(a, b)** 把包含 a 和 b 的等价类合并成一个等价类。
 - **Find(e)** 确定哪个类包含元素 e , 搜索的目的是为了确定给定的两个元素是否在同一个类之中
 - 可以利用两个**Find**操作和一个**Union**操作产生一个组合操作, 该操作能把两个不同的类合并成一个类。
 - **Combine(a, b)**等价于: $i = \text{Find}(a); j = \text{Find}(b);$
 $\text{if}(i \neq j) \text{Union}(i, j);$



在线等价类

■ 在线等价类问题也称作离散集合合并/搜索问题（disjoint set union-find problem）

■ Union-Find数据结构

- 设 $U=\{1,2,\dots,n\}$, A_i 是 U 的某些不交子集;
- $\text{union}(A_i, A_j)$ 指对这些子集中的 A_i 和 A_j 做并操作 $A_i \cup A_j$;
- $\text{find}(x)$, $x \in U$, 指找 x 所在的子集 A_i , 即, $x \in A_i$;
- 假定初始有 n 个单元元素的子集 $A_i = \{i\}$, $1 \leq i \leq n$;
- 试图找一种表示集合的数据结构和算法, 使得在线 (online) 地执行任何 $n-1$ 个 union 和 $m \geq n$ 个 find 操作的混合序列的累积时间接近线性的.

第一种解决方案

- 在线等价类问题的一种简单解决办法是使用一个数组E，并令E(e) 为代表包含元素e 的等价类。
- 完成初始化、合并及搜索操作的函数如程序3-46所示。
- n 是元素的数目，n 和E 均被定义为全局变量。为了合并两个不同的类，可从类中任取一个元素，然后把该类中所有元素的E 值修改成另一个类中元素的E 值。
- Initialize和Union函数的复杂性均为 $\Theta(n)$ ，Find的复杂性为 $\Theta(1)$ 。
- 在应用这些函数时，通常执行一次初始化，u 次合并和f 次搜索，故所需要的总时间为 $\Theta(n + u * n + f) = \Theta(u * n + f)$ 。



程序3-46 使用数组的在线等价类函数

```
void Initialize(int n)
{
    // 初始化n个类，每个类仅有一个元素
    E = new int [n + 1];
    for (int e = 1; e <= n; e++)
        E[e] = e;
}

void Union(int i, int j)
{
    // 合并类 i 和类 j
    for (int k = 1; k <= n; k++)
        if (E[k] == j) E[k] = i;
}

int Find(int e)
{
    // 搜索包含元素 i 的类
    return E[e];
}
```

第二种解决方案

- 针对每个等价类设立一个相应的链表，可以降低合并操作的时间复杂性，可以沿着类 j 的链表找到所有 $E[k]=j$ 的元素，而不必去检查所有的 E 值

- 在使用链表时，初始化和搜索操作的复杂性仍分别保持为 $\Theta(n)$ 和 $\Theta(1)$ 。
- 每次合并操作的开销为(较小类的大小)。令 i 表示合并操作中的较小类。在合并期间， i 中的每个元素从类 i 移向类 j ，因此 u 次合并的复杂性由移动元素的总次数确定。移动类 i 之后，新类的大小至少是类 i 的两倍（因为在移动前有 $\text{size}[i] \leq \text{size}[j]$ ，而移动之后新类的大小为 $\text{size}[i] + \text{size}[j]$ ）。因此，由于在操作结束时没有哪个类的元素数会超过 $u+1$ ，所以在 u 次合并期间，没有哪个元素的移动次数超过 $\log_2(u+1)$ 。在 u 次合并过程中，最多可以移动 $2u$ 个元素，所以，元素移动的总次数不会超过 $2u \log_2(u+1)$ 。至此可以知道执行 u 次合并操作所需要的时间为 $O(u \log u)$
- 1次初始化、 u 次合并操作和 f 次搜索的复杂性为 $O(n + u \log u + f)$

```
void Initialize(int n)
{ // 初始化n个类，每个类仅有一个元素
  node = new EquivNode [n + 1];
  for (int e = 1; e <= n; e++) {
    node[e].E = e;
    node[e].link = 0;
    node[e].size = 1;
  }
}

void Union(int i, int j)
{ // 合并类 i 和类 j
  // 使 i 代表较小的类
  if (node[i].size > node[j].size)
    swap(i,j);
  //改变较小类的 E值
  int k;
  for (k = i; node[k].link; k = node[k].link)
    node[k].E = j;
  node[k].E = j; // 链尾节点
  //在链表j的首节点之后插入链表 i
  // 并修改新链表的大小
  node[j].size += node[i].size;
  node[k].link = node[j].link;
  node[j].link = i;
}

int Find(int e)
{ //搜索包含元素 i 的类
  return node[e].E;
}
```



在线等价类的第三种解决方案

- 把每个集合(类)描述为一棵树
- 而树中每个非根节点都指向其父节点
- 用根元素作为集合标识符
- 如图8.15

Union-Find算法

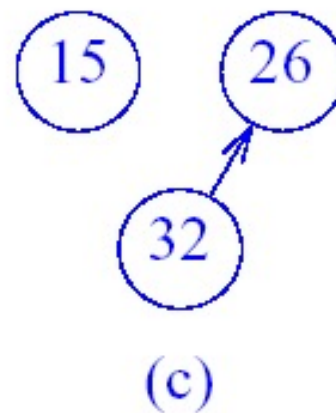
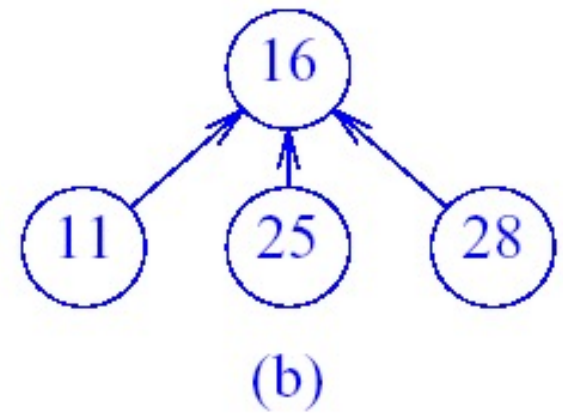
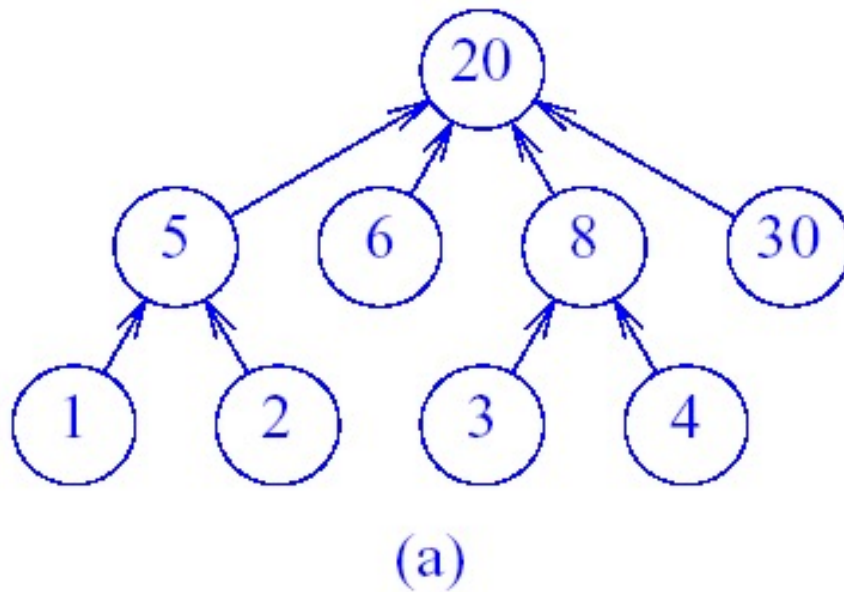


Figure 8.15 Tree representation of disjoint sets

集合的树表示

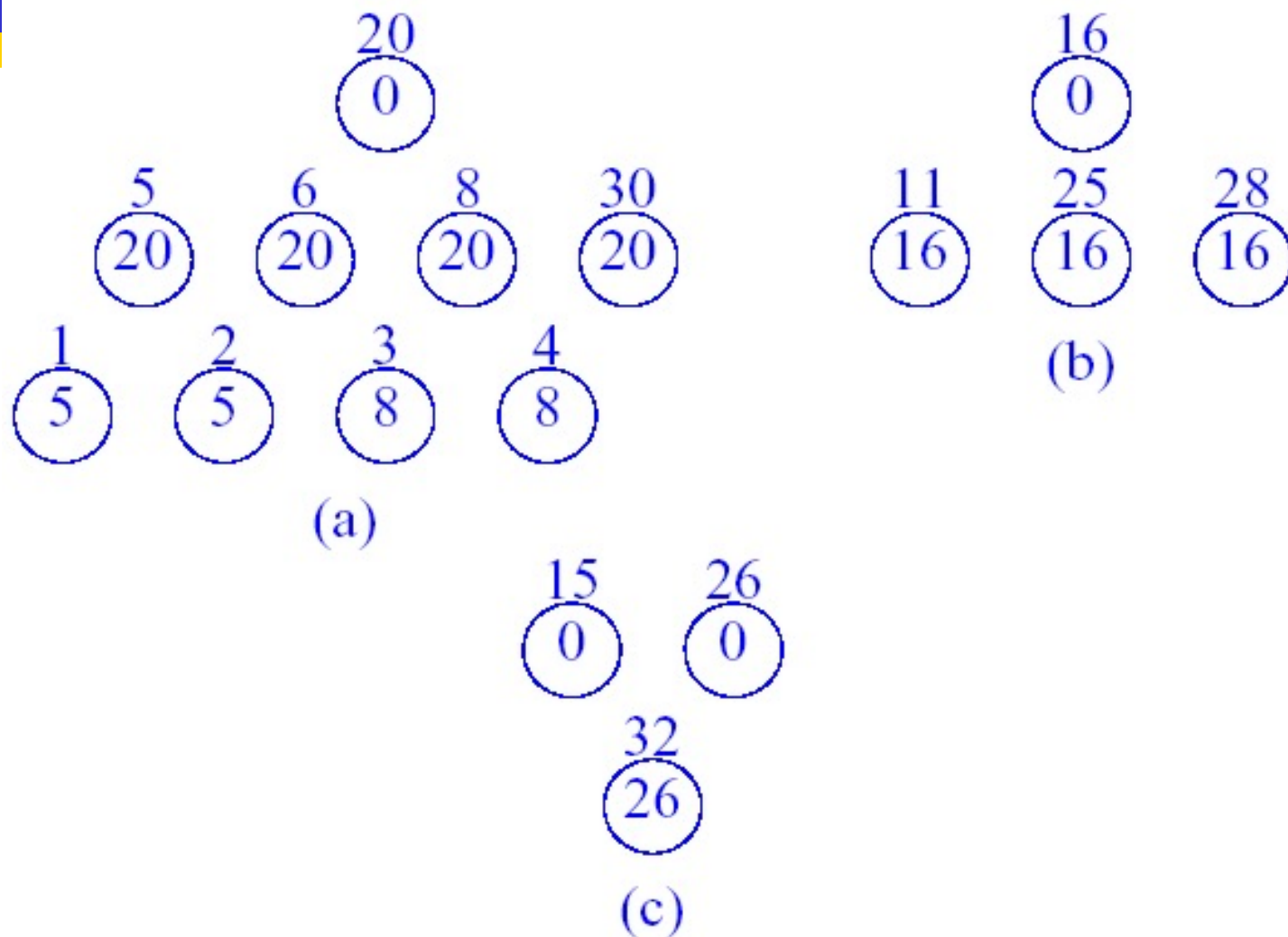


Figure 8.16 Representation of trees of figure 8.15

Union-Find算法

```
void Initialize(int n)
{
    // One element per setclass tree.
    parent = new int [n+1];
    for (int e = 1; e <= n; e++)
        parent[e] = 0;
}

int Find(int e)
{
    // Return root of tree containing i.
    while (parent[e])
        e = parent[e]; // move up one level
    return e;
}

void Union(int i, int j)
{
    // Combine trees with roots i and j.
    parent[j] = i;
}
```

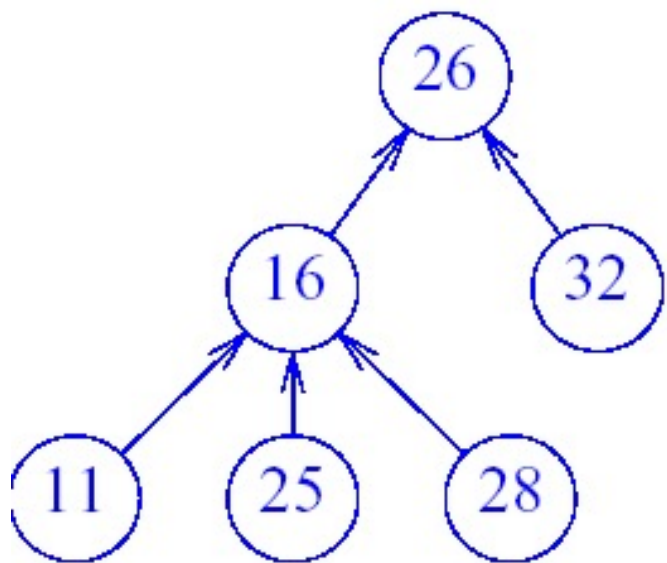
Program 8.15 Simple tree solution to union-find problem



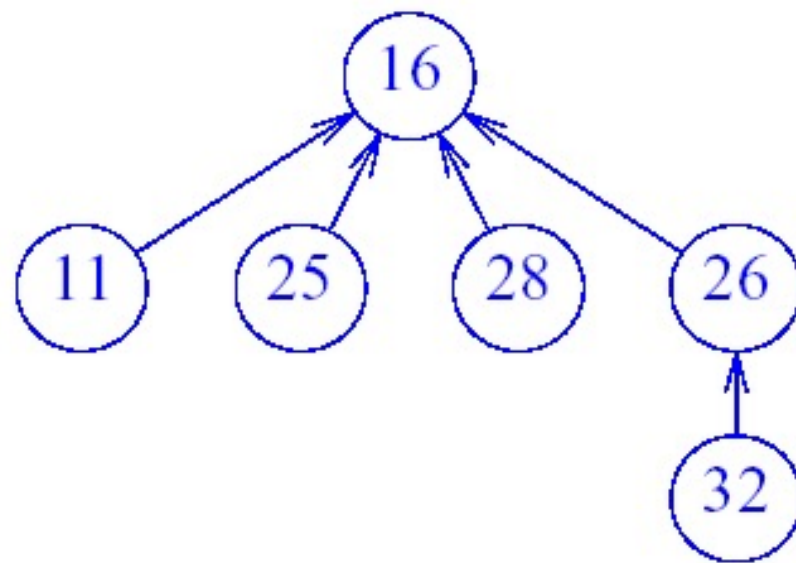
算法复杂性

- 假设要执行 u 次合并和 f 次查找。因为每次合并前都必须执行两次查找，因此可假设 $f > u$ 。
- 每次合并所需时间为 $\Theta(1)$ 。而每次查找所需时间由树的高度决定。在最坏情况下，有 m 个元素的树的高度为 m 。
- 若使用重量规则(高度规则)，合并和查找序列的代价(不包括初始化时间)为 $O(u + f \log u)$

图8.17 Union



(a)



(b)

Figure 8.17 Union

图8.18 Two sample trees

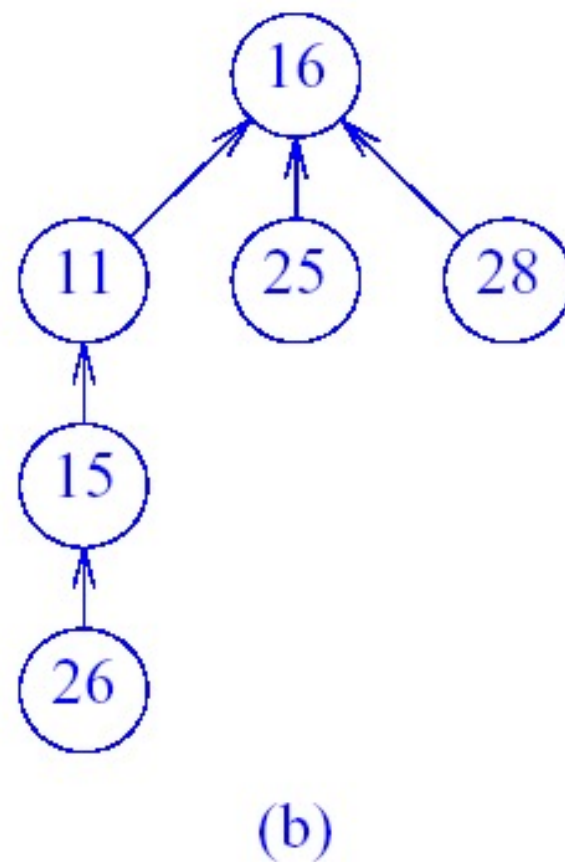
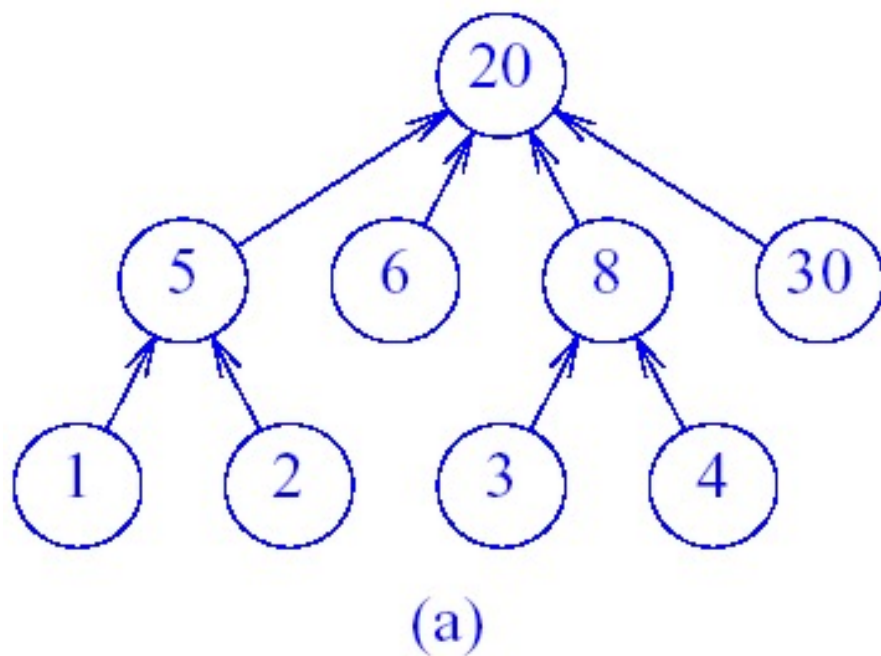


Figure 8.18 Two sample trees



算法改进

- 使用加权规则
- 定义[重量规则] 若树 i 节点数少于树 j 节点数, 则将 j 作为 i 的父节点。否则, 将 i 作为 j 的父节点。
- 定义[高度规则] 若树 i 的高度小于树 j 的高度, 则将 j 作为 i 的父节点, 否则将 i 作为 j 的父节点。



加权规则(Weight rule)

```
void Initialize(int n)
{
    // One element per set/class/tree.
    root = new bool[n+1];
    parent = new int [n+1];
    for (int e = 1; e <= n; e++) {
        parent[e] = 1;
        root[e] = true;
    }
}

int Find(int e)
{
    // Return root of tree containing e.
    while (!root[e])
        e = parent[e]; // move up one level
    return e;
}
```



续

```
void Union(int i, int j)
{
    // Combine trees with roots i and j.
    // Use weighting rule.
    if (parent[i] < parent[j]) {
        // i becomes subtree of j
        parent[j] += parent[i];
        root[i] = false;
        parent[i] = j; }
    else { // j becomes subtree of i
        parent[i] += parent[j];
        root[j] = false;
        parent[j] = i; }
}
```

Program 8.16 Unioning with the weight rule

Weight rule lemma

- **Lemma 8.1** 假定从单元素集开始执行加权并.

设 t 是上述过程产生的有 p 个节点的一棵树,则 t 的深度不超过 $\lfloor \log p \rfloor + 1$

- 证明: 设 t_1, t_2 为产生 t 的Union序列中最后一次合并前的树. 设 t_1, t_2 的节点数为 p_1 和 p_2 , (均小于 p), 又设 $p_1 \leq p_2$, 对 p_1 和 p_2 用归纳假设:

$$d_1 \leq \lfloor \log p_1 \rfloor + 1 \quad d_2 \leq \lfloor \log p_2 \rfloor + 1$$

- t 的深度 $d = d_2$ 或 $d_1 + 1$, 无论何种情形都有

$$d \leq \lfloor \log p \rfloor + 1$$

$$(1 + \log p_1 = \log 2p_1 \leq \log(p_1 + p_2) = \log p)$$



优先队列问题

优先队列中元素出队列的顺序由元素的优先级决定。

- 例：假设我们对机器服务进行收费。每个用户每次使用机器所付费用都是相同的，但每个用户所需要服务时间都不同。为获得最大利润，假设只要有用户机器就不会空闲，我们可以把等待使用该机器的用户组织成一个最小优先队列，优先权即为用户所需服务时间。当一个新的用户需要使用机器时，将他/她的请求加入优先队列。一旦机器可用，则为需要最少服务时间（即具有最高优先权）的用户提供服务。
- 如果每个用户所需时间相同，但用户愿意支付的费用不同，则可以用支付费用作为优先权，一旦机器可用，所交费用最多的用户可最先得得到服务，这时就要选择最大优先队列。



第一种方案

■ 1) 采用无序线性表描述最大优先队列

- 假设有一个具有 n 个元素的优先队列，插入操作可以十分容易地在表的右端末尾执行，插入所需时间为 $\Theta(1)$ 。删除操作时必须查找优先权最大的元素，即在未排序的 n 个元素中查找具有最大优先权的元素，所以删除操作所需时间为 $\Theta(n)$

■ 2) 采用有序线性表

- 删除时间均为 $\Theta(1)$ ，插入操作所需时间为 $\Theta(n)$



第二种方案

- 可以利用堆数据结构来高效地实现优先队列
- 定义[最大树（最小树）] 每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树。
- 最大树不必是二叉树，最大树或最小树节点的子节点个数可以大于2。
- 定义[最大堆（最小堆）] 最大（最小）的完全二叉树。

Heaps

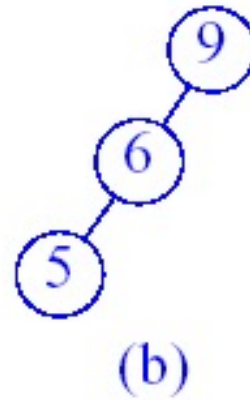
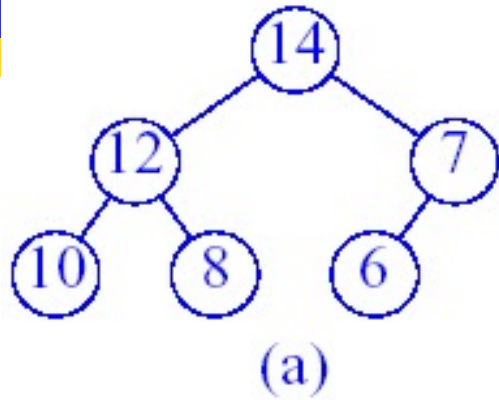


Figure 9.1 Max trees

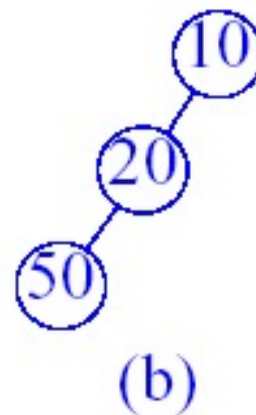
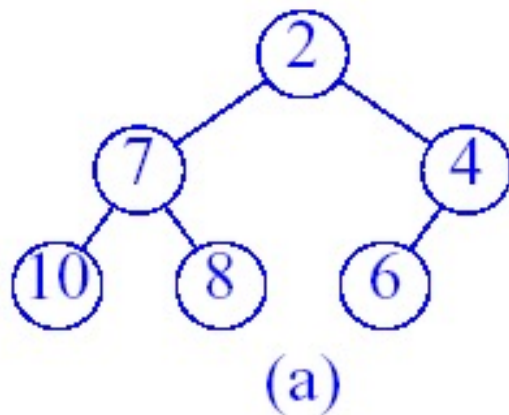
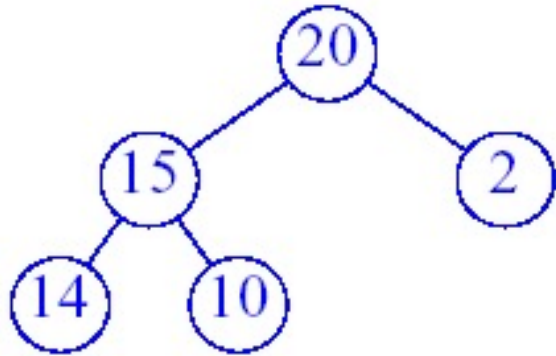
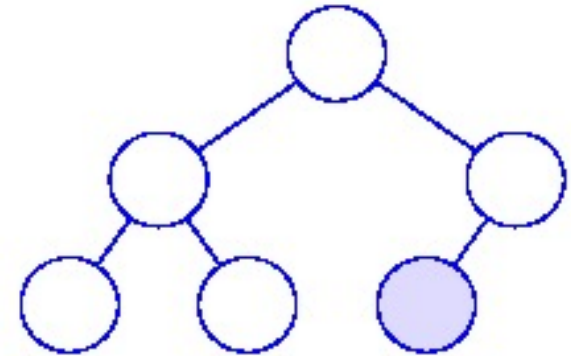


Figure 9.2 Min trees

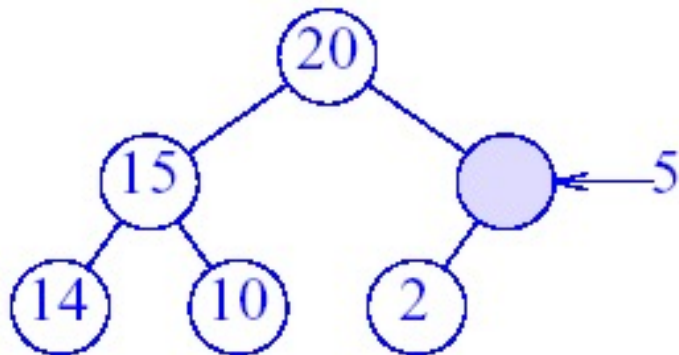
图9.3 Insertion into a max heap



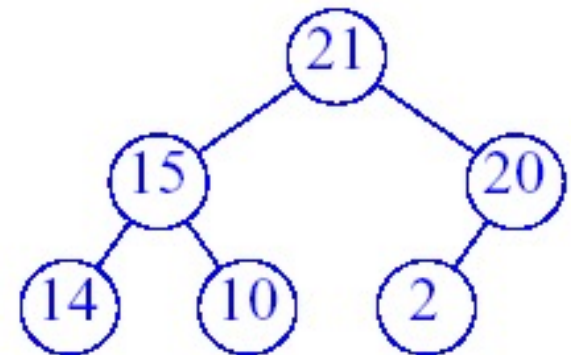
(a)



(b)



(c)



(d)

Figure 9.3 Insertion into a max heap

图9.4 Deletion from a max heap

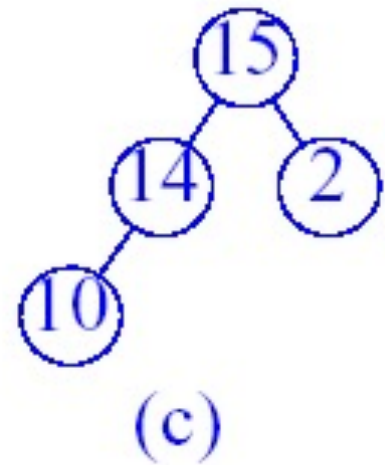
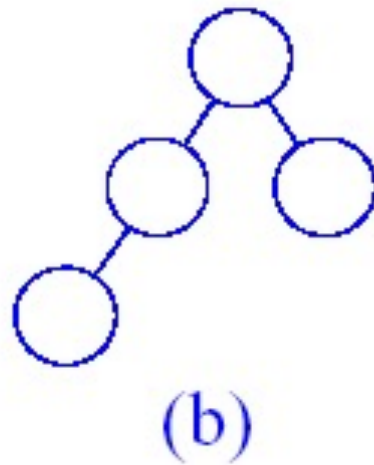
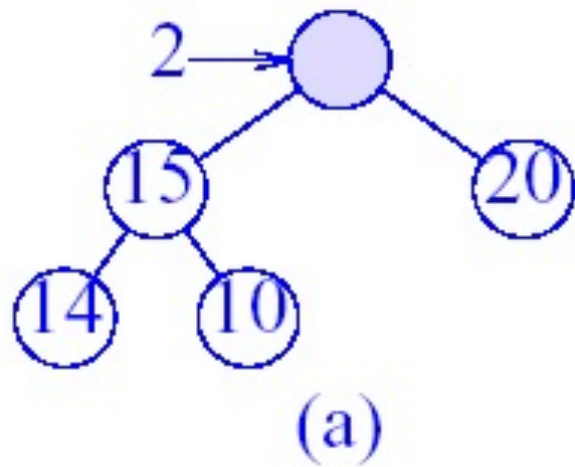


Figure 9.4 Deletion from a max heap

图9.5 Initializing a max heap

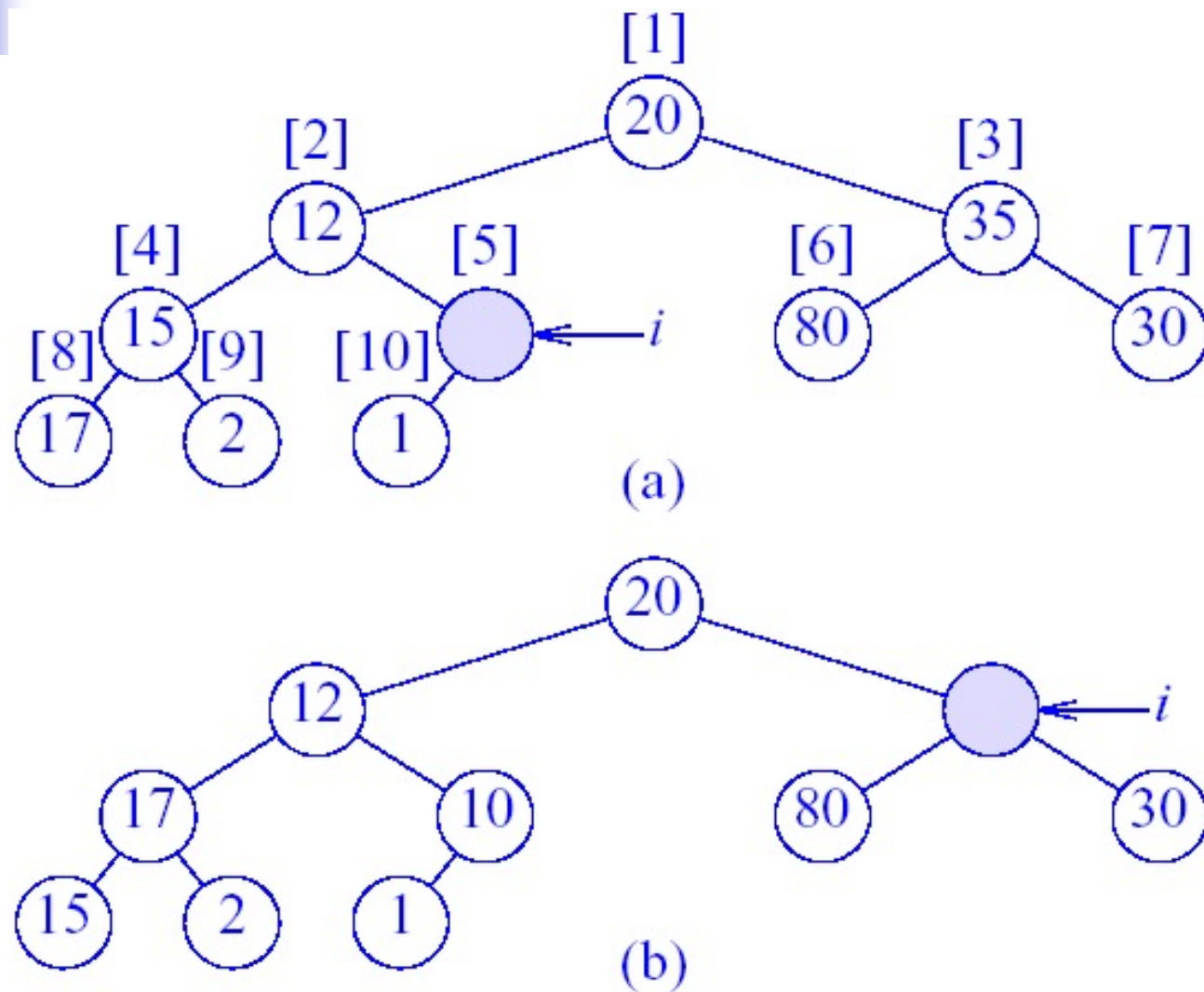


Figure 9.5 Initializing a max heap

图9.5 Initializing a max heap(续)

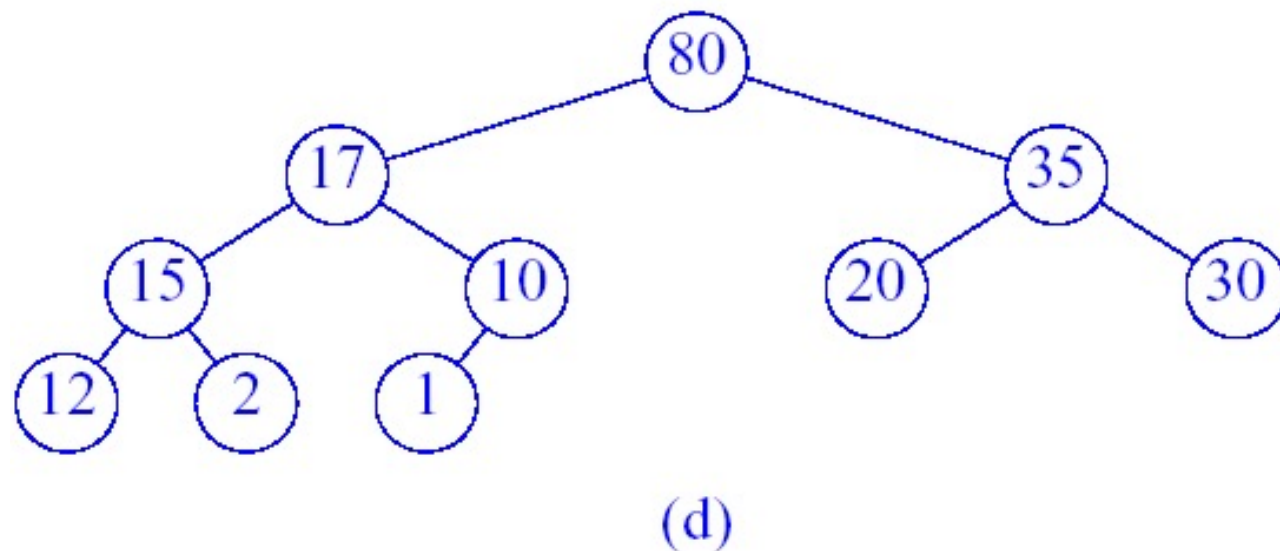
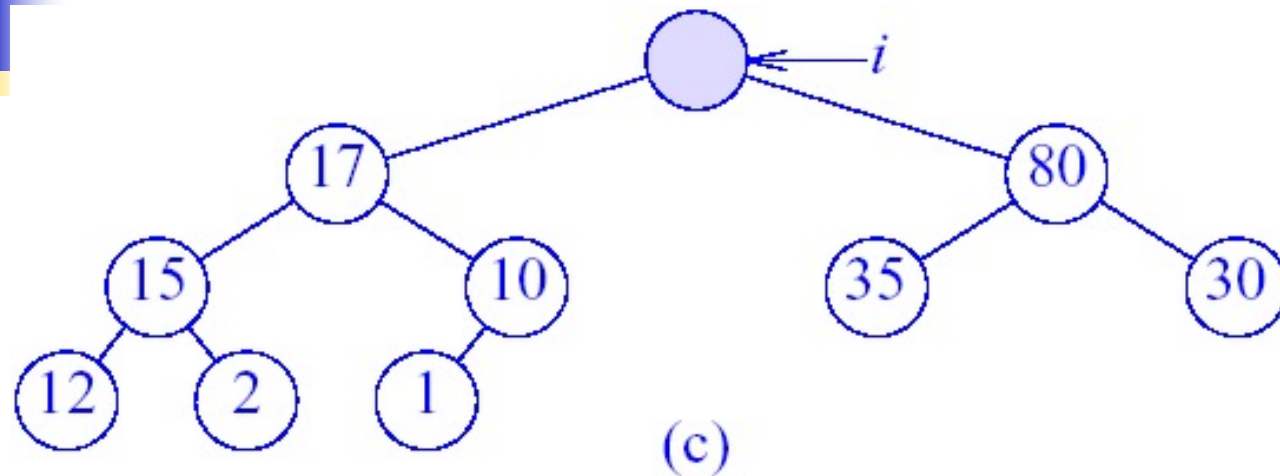


Figure 9.5 Initializing a max heap (Continuation)



堆排序分析

- n 个节点的完全二叉树的深度为 $\lceil \log(n+1) \rceil$

$$k = \lfloor \log n \rfloor + 1$$

- 堆插入和删除需 $\Theta(\log n)$ 即
- 堆排序时间为 $O(n \log n)$
- 初始化: $O(n \log n)$