堆 与 并查集

堆 (HEAP)

并查集 (UNION-FIND DATA STRUCTURE)

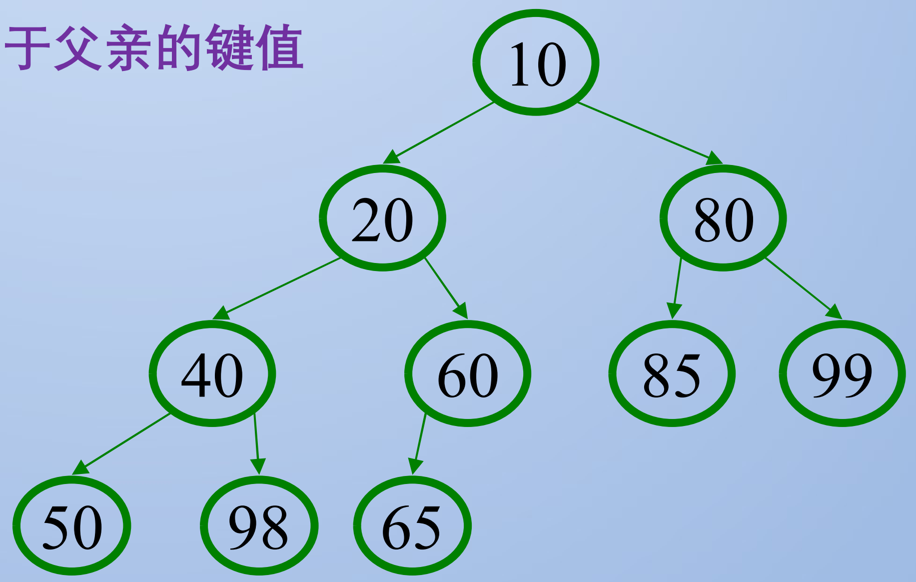
(DISJOINT-SET DATA STRUCTURE)

堆的抽象数据类型定义

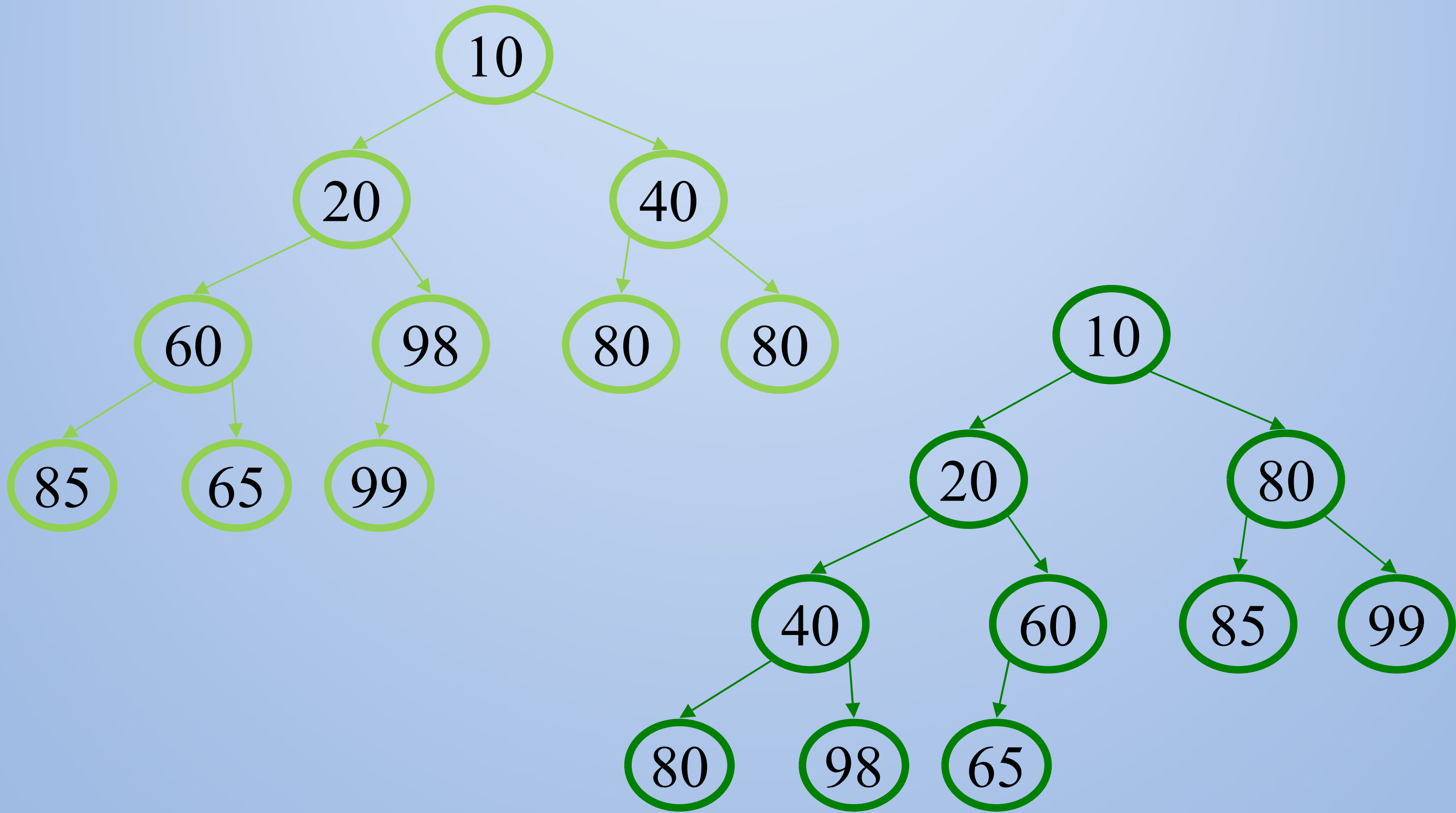
- 基本操作:
- insert(int val)　　插入一个键值
- void decrease_value(int i, int val)
  减小第i个结点的键值到val
- int delete_min()　删除最小键值的结点
- void delete(int i)　删除第i个结点
- Update_Value(int i, int val)
  修改第i个结点的键值为val

HEAP PROPERTY （堆性质）

- 完全二叉树；每个结点储存一个键值（value）。
- **堆性质（heap property）**
  - **孩子的键值不小于父亲的键值**
- 满足该性质称作最小堆。
- 最大堆相反。

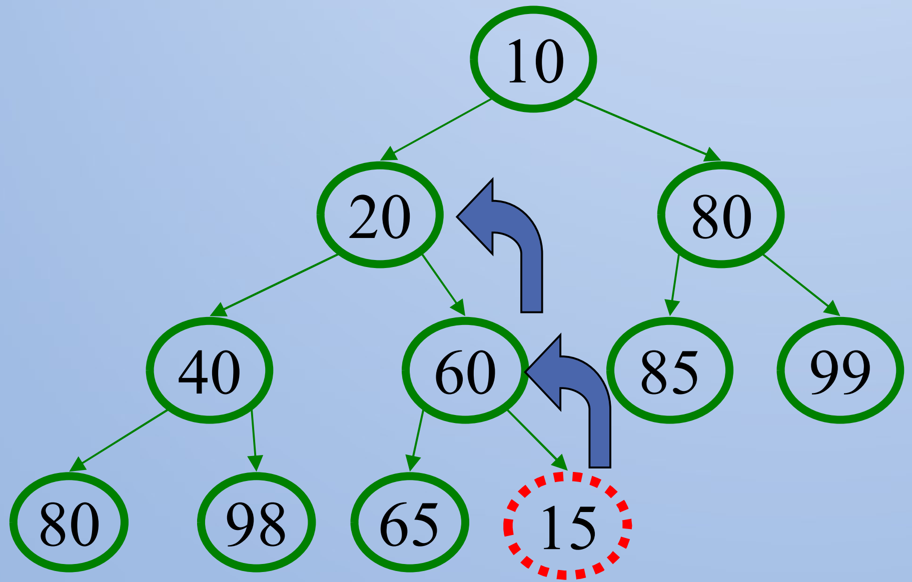


同样的键值，堆并不唯一



INSERT(INT VAL)

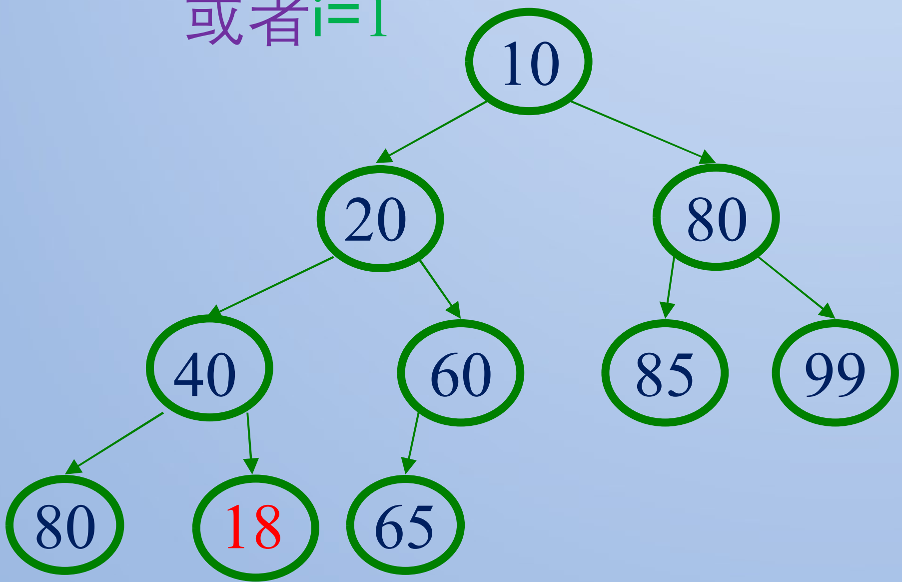
- 基本思想：
  - 先暂时放在最后的位置。
  - 不断往上调整。




```
void insert(int val) {
  int i = ++size;
  while (i > 1 &&
     val < Heap[i/2])
    Heap[i] = Heap[i/2];
    i /= 2;
  }
  Heap[i] = val;
}
```

DECREASE_VALUE(INT I, INT VAL)

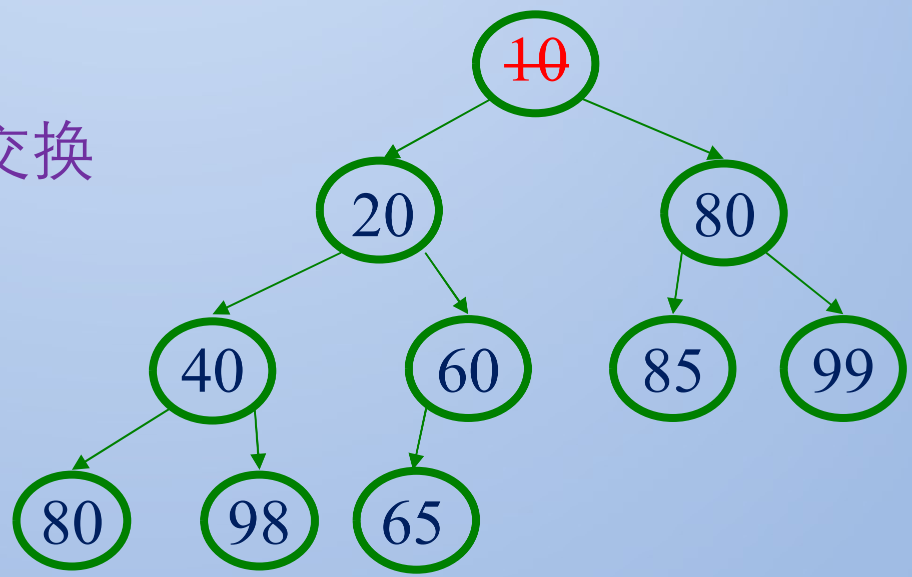
- 将Heap[i]赋值为val
- 然后不断向上调整
  直到Heap[i] ≥ Heap[i/2]
  或者i=1




```
void decrease_value(int i,
   int val) {
  while (i > 1 &&
     val < Heap[i/2])
    Heap[i] = Heap[i/2];
    i /= 2;
  }
  Heap[i] = val;
}
```

DELETE_MIN

- 将堆中最后一个元素移动到根。
- 往下调整:
  - 在根的孩子中挑最小的与根的value比较
  - 如果比根小则与根交换
  - 继续往下。

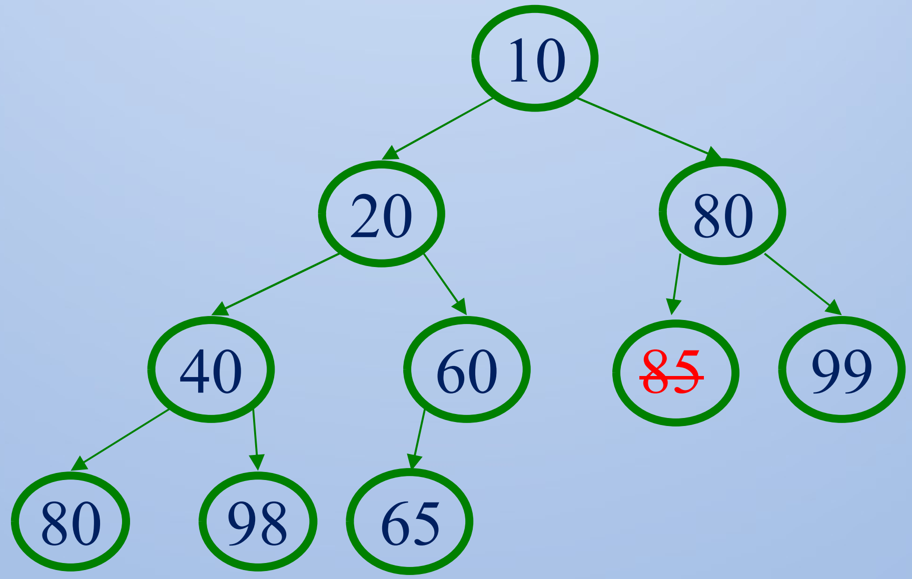


DELETE_MIN的实现

```
int delete_min() {
  int val = Heap[size--], ret = Heap[1];
  int i = 1, ch = 2;
  while (ch <= size)
    if (ch < size && Heap[ch+1] < Heap[ch])
        ch++;
    if (val <= Heap[ch]) break;
    Heap[i] = Heap[ch]; i = ch; ch += ch;
  }
  Heap[i] = val; return ret;
}
```

DELETE(INT I)

- 既有可能往下调整（如删除10，65要往下走）
- 也有可能往上调整（如删除85，65要往上走）



DELETE(INT I)的实现

```
void delete(int i) {
  int val = Heap[size--], ch = i * 2;
  while (ch <= size)
    if (ch < size && Heap[ch+1] < Heap[ch])
        ch++;
    if (val <= Heap[ch]) break;
    Heap[i] = Heap[ch]; i = ch; ch += ch;
  }
  while (i > 1 && val < Heap[i / 2]){
    Heap[i] = Heap[i / 2]; i /= 2;
  }
  Heap[i] = val;
}
```

UPDATE_VALUE(INT I, INT VAL)

- 类似于decrease_value(int i, int val)
- 但是既有可能往上调整（当Heap[i]被减小）也有可能往下调整（当Heap[i]被增加）
- 该函数的具体实现留作课后习题。
    - 请参照Delete(int i)的实现。

与线性表的 时间复杂度对比

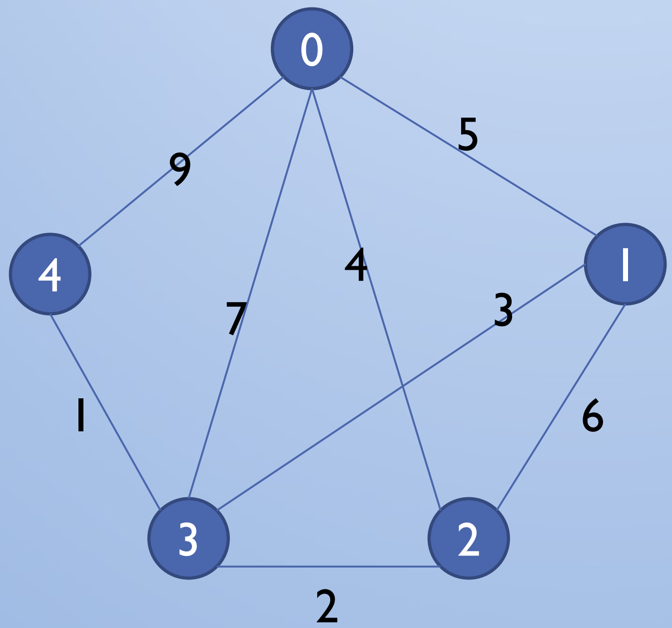
- 假如希望支持： 插入 + 查询最小元素。
  - 用无序线性表。 O(n) （查询要O(n)）
  - 用有序线性表。 O(n) （插入要O(n)）
  - 用堆。 两者都是 O(log(n))。
- 堆是一种优先队列（priority queue）。
  （后续slides将会介绍）

应用1：堆排序

- 【问题描述】给定a_1~a_n，将它们从小到大输出。
- 思路：
  - 建堆。
  - 执行n步骤：
    - 打印 delete_min();
- 时间复杂度为O(n log n)。
- 实际上，发明堆的人是为解决排序而发明的堆。

应用2：DIJKSTRA算法

- 【问题描述】求单源点0到其余各点的最短路径。
- F[1]…F[n]。 F[i]表示S到i的最短路径长度。



F[4]=?
0到4: 9
0到3到4: 长度为
7+1=8
0到2到3再到4:
长度为4+2+1=7
F[4]=7.

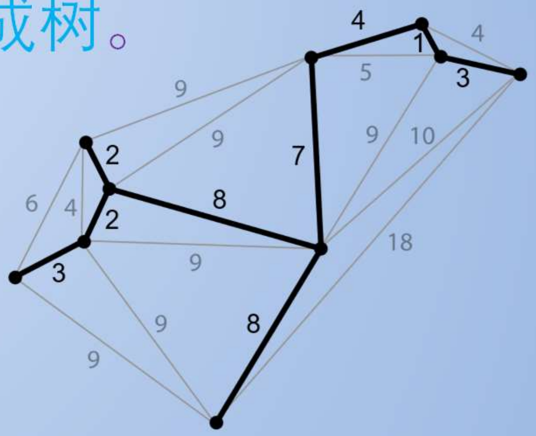
应用2： DIJKSTRA算法

算法描述

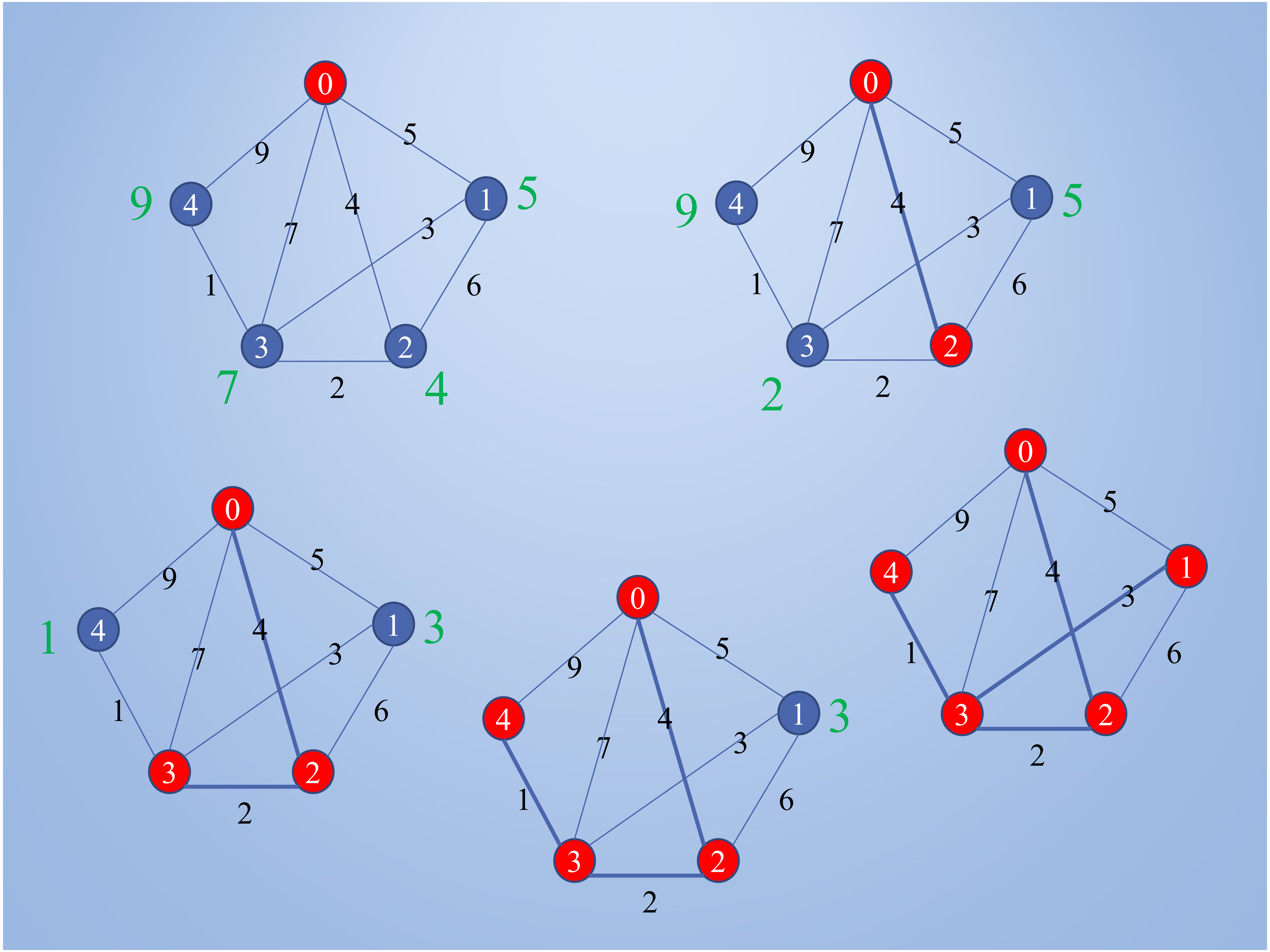
- F[i]初值设置为无穷大(i>0)；F[0]=0。
- 每次选一个F[i]最小的还未扩展的i，进行扩展：
  - 如果有一条边(i,j)，长度为L。
  - 令F[j] = min (F[j], F[i]+L)。
- 当所有点都被扩展完以后。F[i]中的值即为0→S的最短路径的长度。

应用3：PRIM算法

- Prim算法用来计算一个图的最小生成树。(后续课程将会讲授具体算法)



- 类似Dijkstra算法，也可以转化为：
  - Decrease_value();
  - insert();
  - delete_min();
- 用堆来做，复杂度为 O((|V|+|E|) log |V|)



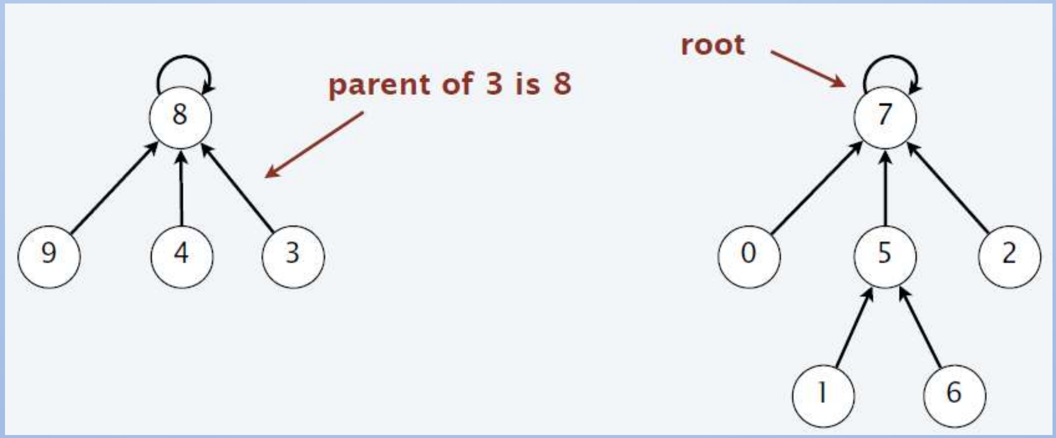
0 1 2 3 4
5 9 7 4 1 3 6 2
9 5 7 4
0 1 2 3 4
9 5 7 4 1 3 6 2
9 5 2
0 1 2 3 4
9 5 7 4 1 3 6 2
1 3
0 1 2 3 4
9 5 7 4 1 3 6 2
3
0 1 2 3 4
9 5 7 4 1 3 6 2

并查集

- 并查集(Union-find data structure)也叫做 Disjoint-set data structure
- 它的研究开始于1960年代。
- 并查集应用很广
  - Kruskal's algorithm.
  - Connected components.
  - Computing LCAs in trees.
  - Etc. (equivalence class)

并查集基本结构

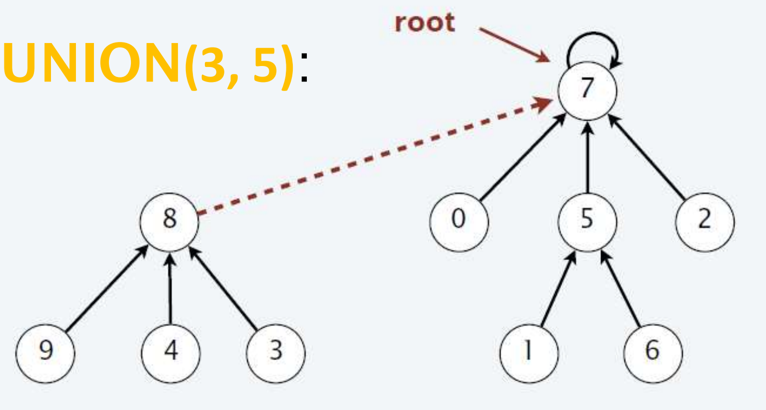
- Parent-link representation.
  - Represent each set as a tree of elements
  - Each element has a parent pointer in the tree.
  - The root serves as the canonical element
    (and it points to itself).



并查集三种基本操作

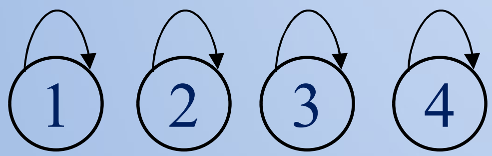
- **Make-Set(x):** build a tree with a single node x
- **FIND(*x*):** find the root of the tree containing *x*.
- **UNION(*x, y*):** merge trees containing *x* and *y* (by making one root point to the other).

UNION(3, 5):



举例

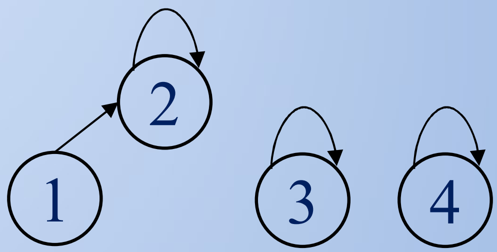
- Make-Set(1)
- Make-Set(2)
- Make-Set(3)
- Make-Set(4)



1

2

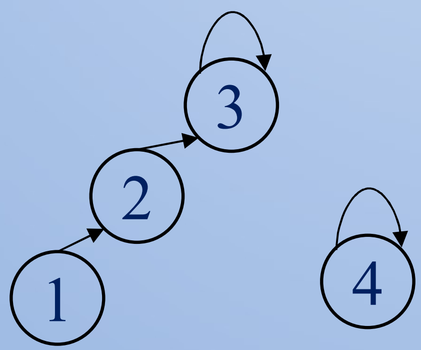
Union(1,2)



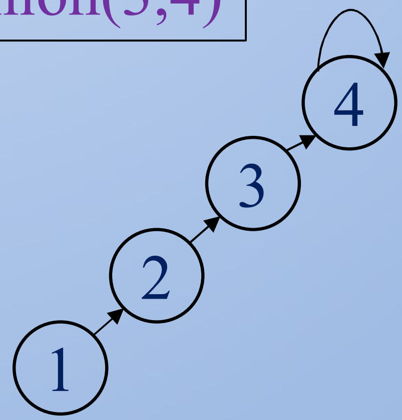
3

4

Union(2,3)



Union(3,4)



PSEUDO CODE

- **Make set(x)**

  parent(x)←x;

- **Find(x)**

  while (x!= parent(x))

  x ← parent(x);

  return x;

- **Union(x, y)**

  r←Find(x);

  s←Find(y);

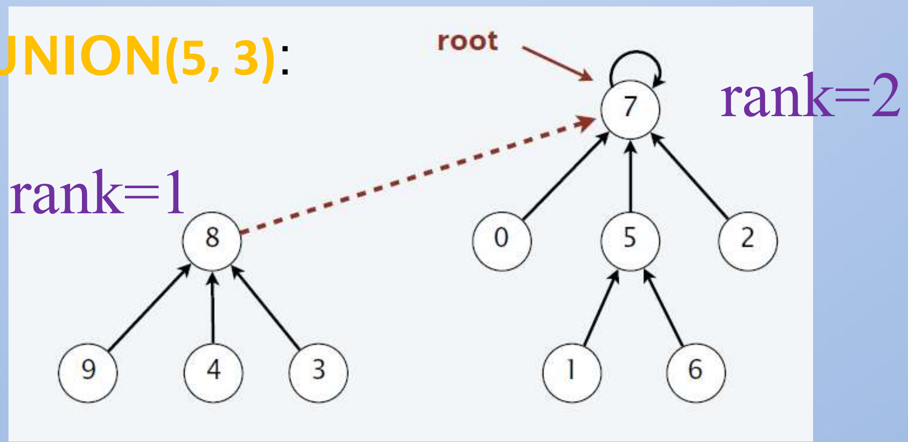
  parent(r) ← s;

a UNION or FIND operation can take $\Theta(n)$ time in the worst case, where n is the number of elements

LINK-BY-RANK

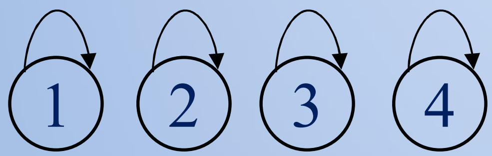
- Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

UNION(5, 3):



举例（LINK-BY-RANK）

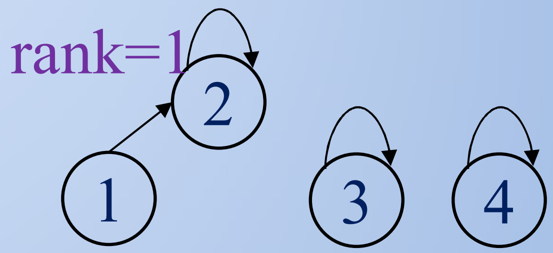
- Make-Set(1)
- Make-Set(2)
- Make-Set(3)
- Make-Set(4)



1

2

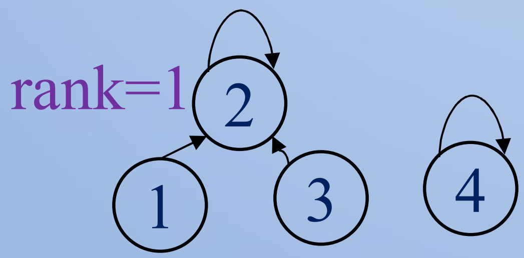
Union(1,2)



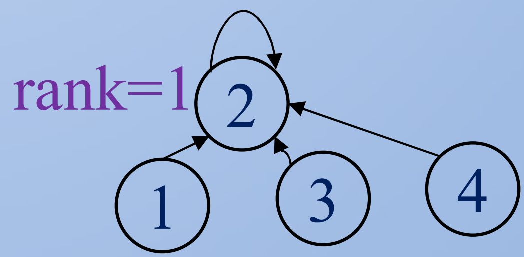
3

4

Union(2,3)



Union(3,4)

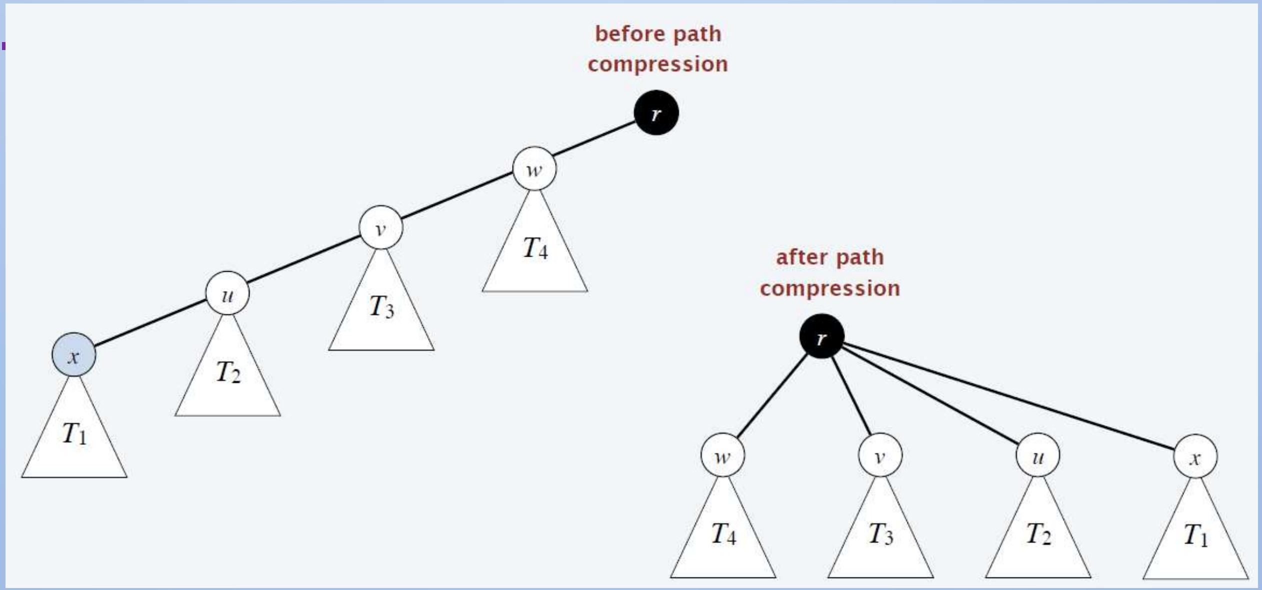


LINK-BY-RANK (PSEUDO CODE)

- **UNION(x, y)**
  - $r \leftarrow$ FIND(x). $s \leftarrow$ FIND(y).
  - **if** ($r = s$) **RETURN**.
  - **if** ($rank[r] > rank[s]$) $parent[s] \leftarrow r$.
  - **else** if ($rank[r] < rank[s]$) $parent[r] \leftarrow s$.
  - **else** {
  -    $parent[r] \leftarrow s$. $rank[s]$ ++.
  - }

PATH COMPRESSION

- When finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



PATH COMPRESSION(PSEUDO CODE)

- When finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.

FIND(*x*)
  IF (*x* != *parent*[*x*])
    parent[*x*] ← FIND(*parent*[*x*]).
  RETURN *parent*[*x*].

the new FIND changes the tree structure

BOUNDS ON EFFICIENCY (I) (***)

定理. Using link-by-rank, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements.

定理 (Tarjan–van Leeuwen 1984) Starting from an empty data structure, path compression with naïve linking performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND on a set of n elements in $O(m \log n)$ time.

BOUNDS ON EFFICIENCY (II) (****)

- 定理. Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.
- log*n 表示 对n取反复对数时多少步会<=1。
  - log*1=0. log*2=1. log*4 = 2. log*16=3. log* ($65536=2^{16}$)=4. log*(2^{65536})=5.
  - 2^{65536}大于宇宙原子个数; 所以可认为log*n非常小

BOUNDS ON EFFICIENCY (II) (****)

- 定理. Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

  对这个定理的证明感兴趣的同学，请阅读 https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf
  大约需要120min理解.

BOUNDS ON EFFICIENCY (III) (*****)

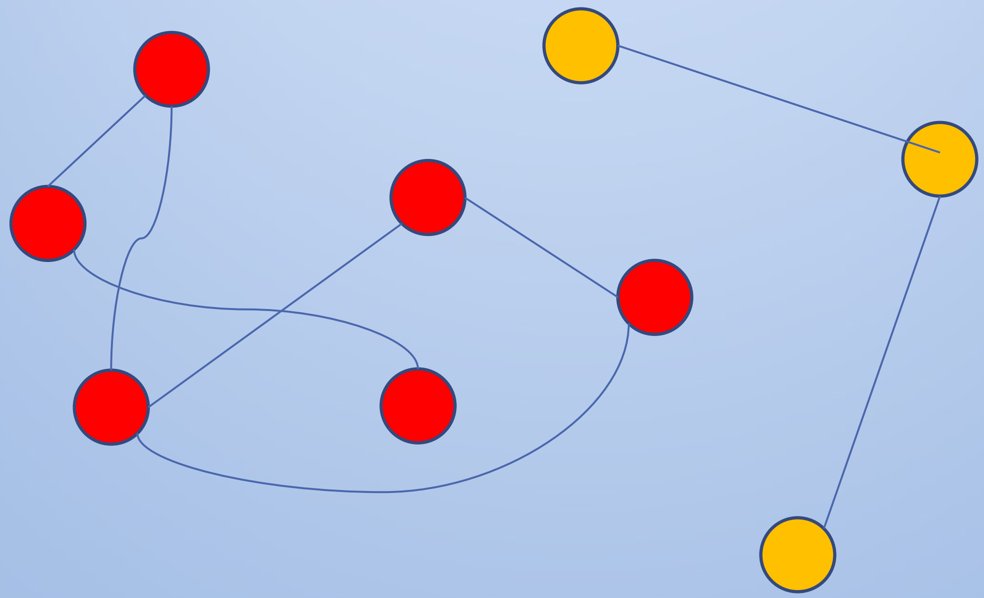
- 定理. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, link-by- { size, rank } combined with { path compression, path splitting, path halving } performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\,\alpha(m, n))$ time.
  - link-by-size 类似于link-by-rank
  - path splitting / halving 类似于path-compression
  - $\alpha(m, n)$是反Ackermann函数 。 （一般认为<4)

TIGHT LOWERBOUND(*****)

- 定理. [Fredman-Saks 1989] In the worst case, any CELL-PROBE(log n) algorithm requires $\Omega(m\,\alpha(m, n))$ time to perform an intermixed sequence of m MAKE-SET, UNION, and FIND on a set of n elements.
  - Cell-probe model. [Yao 1981] Count only number of words of memory accessed; all other operations are free.

应用1　连通分量（联通分块）

- 连通分块：彼此连通的最大的顶点集合。



应用1　连通分量

for (e=(x,y) 为G中的一条边) Union(x, y)。

for (int x 为G 中一个顶点)

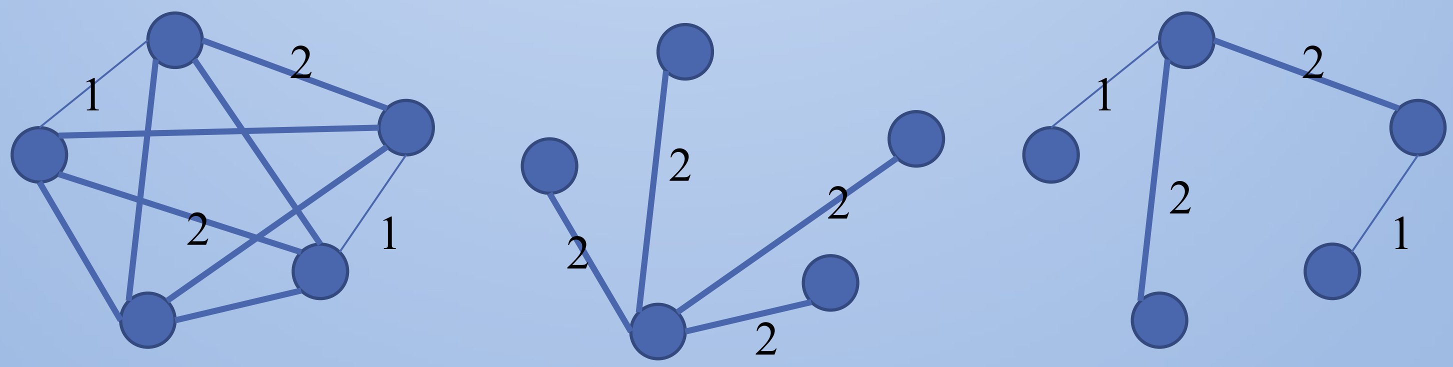
　将x加入到List[Find(x)]末尾;

for (int x 为G中一个顶点） 打印List[x]。

- 时间复杂度 (m+n) log*(n)。 (or, O(m+n) α(*m*, *n*))
  - m为G中边的数目。n为G中顶点数目。

应用2 KRUSKAL最小生成树算法

- 【问题描述】 给定一个图G=(V,E)；每条边e有一个权值w_e。求G的最小生成树。
  - 生成树是指G的连通的、且仅有|V|-1条边的子图。
  - 最小生成树是指权值最小的一个生成树。
  - 一棵生成树的权值定义为这棵树所有边的权值之和。



应用2 KRUSKAL最小生成树算法

- 算法描述:
  - 将G中的边按权值从小到大排序。
  - 令F←Ø。
  - 按权值**从小到大**取出e=(x,y)
    - 若x,y在F中不连通。则F←F+{e}。
  - 如果F中有|V|-1条边。则F是最小生成树——输出F；否则输出“G不存在最小生成树”。
- 算法的正确性将在以后的课程中进行讲解。

REFERENCE

- http://www.cs.princeton.edu/~wayne/kleinberg-tardos 等

上机练习题

- 堆：
  - LeetCode692 前K个高频单词
  - LeetCode295 数据流的中位数
- 并查集：
  - LeetCode684 冗余连接
  - LeetCode547 朋友圈
  - https://www.luogu.com.cn/problem/P1196

相关阅读: FIBNACCI堆(****)

- 两个**优点**:
  - 能够支持堆合并
  - **Decrease_value**的均摊复杂度降到了O(1)。
- **缺点**: 复杂很多（相比二叉堆）。

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[17]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[17][18]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[17][19]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$

相关阅读：左式堆/二项式堆(***)

- 左式堆(Leftist heap)
- 二项式堆(Binomial heap)
- 请感兴趣的同学wiki

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[17]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[17][18]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[17][19]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$

FIBONACCI堆的价值

- 在利用Fibonacci堆以后。
  - Dijkstra算法和Prim算法的时间复杂度变为：

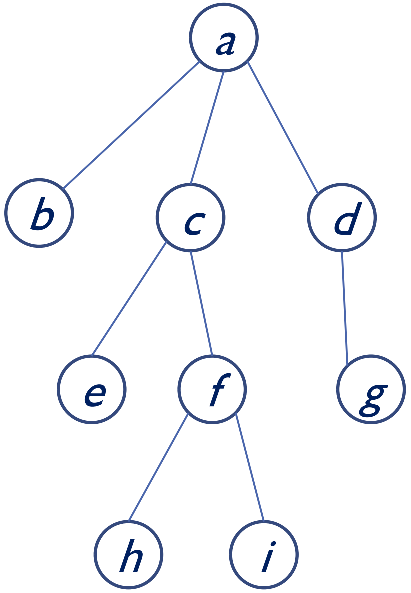
    $O(|E| + |V| \log |V|)$。

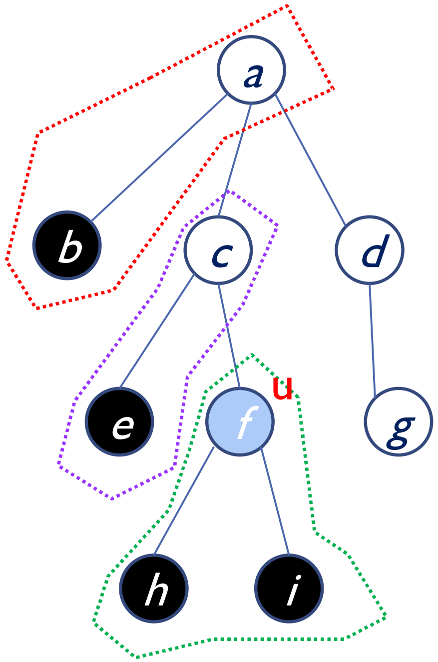
- Open problem:
  - $O(|E|+|V|)$ ？
  - (very difficult problem)

并查集应用3 OFFLINE-NCA

```
• function TarjanOLCA(u) is
•     MakeSet(u); u.ancestor = u;
•     for each v in u.children do
•         TarjanOLCA(v);
•         Union(u, v);  Find(u).ancestor := u;
•     u.color := black;
•     for each v such that {u, v} in P do
•         if v.color == black then
•             printf( "LCA %d %d = %d\n", u,v, Find(v).ancestor);
```

举例



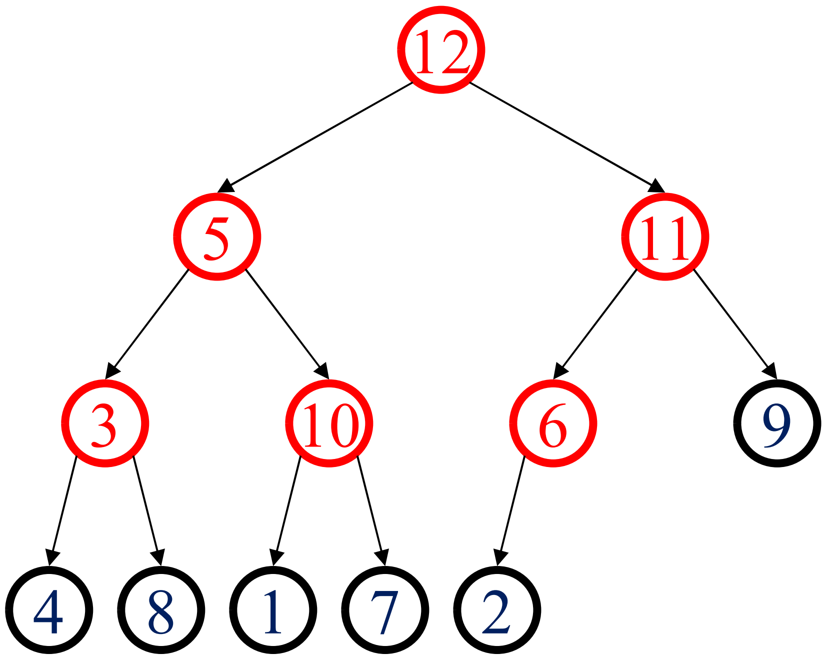


假设我们当前已经访问完了b,e,h,I，当前u=f。
{a,b}在一个set中。它们的祖先为a。
{c,e}在一个set中，它们的祖先为c。
{b,f,i}在一个set中，它们的祖先为f。

堆的快速建立方法（*）

n次插入的建立方法O(nlogn)。可改进!

12	5	11	3	10	6	9	4	8	1	7	2




```
void buildHeap() {
   for (int i = Size/2;
      i > 0; i-- )
      往下调整( i );
}
```

复杂度分析。

为了简单起见。假定n=2^k-1。（满二叉树）

总的运行时间为：

$$\sum_{i=1}^{k-1} 2^{k-i-1} i = 2^{k-1} \sum_{i=1}^{k-1} 2^{-i} i$$

$$= \frac{n+1}{2}\left(\frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 3 + \cdots + \frac{1}{2^{k-1}} * (k-1)\right)$$

$$< \frac{n+1}{2}\left\{\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) + \left(\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots\right) + \cdots\right\}$$

$$< \frac{n+1}{2}\left\{1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right\} < n + 1 = O(n)$$

练习题：中位数的计算

- 【问题描述】给定一个集合S（初始为空集）。
  你被要求支持三种操作：
  - *向S中新增一个元素；*
  - *从S中删除某个元素；*
  - *回答S中的中位数是多少。*
- **目标**：在O(log n)时间内完成每一种操作。
- **解法简述**：构建一个最大堆（保存较小的$\lfloor n/2 \rfloor$个数）
  构建一个最小堆（保存较大的$\lceil n/2 \rceil$个数）。
- 注：不考虑删除操作时，稍微容易一点。