Sets and operations on sets

- Normally sets viewed as mathematical structures.
- But set manipulations needed in many cases for design of algorithms.
- Simplest is to represent sets as arrays.
- $R[i]$ containing the name of set to which i belongs.
- *FIND*(i) requires 1 random access of array.
- But *UNION*(A, B, C) requires sequential scan of $R[.]$ and modifying each entry that is either A or B to C.
- $\therefore$ cost of one *UNION* is O(n). A sequence of n *UNION* is, thus, O(n^2).

Better implementations

- One way could be use linked list to represent sets.
- Then we can merge sets simply by concatenating the lists representing the sets.
- But *FIND*(i) would require each linked list to be scanned sequentially before set containing it can be identified.
- Another way is use the array representation, but to merge the smaller set into larger set for *UNION*.
- The merging requires maintaining internal names and external names for each set as well as size.
- External names are referred by *UNION* operation.
- Internal names maps sets to array $R[.]$. The names are from $1, 2, \ldots, n$.

Details of implementation

- Let $R[i]$ contain name of the set containing i which is internal name.
- For each set A (external name) construct a linked list *LIST*(A) which contains the elements in set A.
- *SIZE*[.] is the array such that *SIZE*$[A]$ denote the number of elements in A.
- Two array *INT_NAME*[.] and *EXT_NAME*[.] are to associate mapping of sets internally and externally.
- *INT_NAME*$[i]$ gives the internal name of set whose external name is i.
- *EXT_NAME*$[A]$ the set name which has A as internal name.

Example

	R	Next
1	2	3
2	3	4
3	2	5
4	3	8
5	2	7
6	1	0
7	2	0
8	3	0

LIST	SIZE	EXT_NAME
6	1	3
1	4	1
2	3	2

Set 1 = {1, 3, 5, 7}

Set 2 ={2, 4, 8}

Set 3 = {6}

	INT_NAME
1	2
2	3
3	1

Example contd.

- Total 8 elements are partitioned into 3 sets
- Set 1 consist of $\{1, 3, 5, 7\}$. Set 2 $= \{2, 4, 8\}$ and set 3 $= \{6\}$
- Internal name of set 1 is 2 that of set 2 is 3 and of set 3 is 1.
- So first and the only element of set 1 (internal name) is 6, size $= 1$.
- The first element of set 2 (internal name) is 1, size $= 3$.
- The first element of set 3 (internal name) is 2, size $= 4$.

Example contd.

```
algorithm UNION(I, J, K) {
    A = INT_NAME(I);
    B = INT_NAME(J);
    wlg (assume SIZE(A) ≤ SIZE(B)) {
        curr = LIST[A];
        while (curr ≠ 0) {
            R[curr] = B;
            prev = curr;
            curr = NEXT[curr];
        }
        NEXT[prev] = LIST[B];
        LIST[B] = LIST[A];
        SIZE[B] = SIZE[A] + SIZE[B];
        INT_NAME(K) = B;
        EXT_NAME(B) = K;
    }
}
```

Cost

- Cost found by partitioning or associating with the movement of each element from one set to a another.
- Every time an element moves it finds itself to be a member of the set at least twice as large of the set of which it was a member before *UNION*
- So no element can move more than $\log n$ time.
- So the total cost of $n-1$ *UNION*s is O($n \log n$).

Almost linear time algorithm

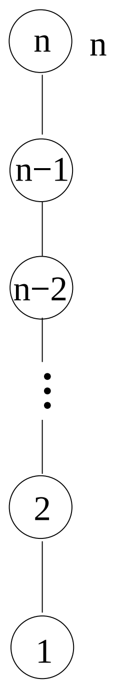
- Use a rooted tree to represent each set.
- The vertices of tree represent elements of the set.
- The name of the set is attached to the root.
- *UNION*(A, B, C) is executed by making the root r_A of tree T_A representing A a son of r_B of tree T_B representing B. Then changing r_B to r_C.
- This approach gives a cost of O(1) time to each *UNION*.
- But *FIND*(i) cost becomes proportional to the length of the path $i \stackrel{*}{\rightarrow} r$ where r is the root of the tree to which i belongs.

Cost of FIND

- Suppose we have a sequence of n *UNION-FIND*s consisting of $n-1$ *UNION*s which precedes the *FIND*s.
- The sequence could be

  UNION(1, 2, 2)
  UNION(2, 3, 3)
  ⋮
  UNION(n-1, n, n)
  FIND(1)
  FIND(2)
  ⋮
  FIND(n)

Worst case scenario



- In the particular situation cost of *FIND*s will be O(n^2).

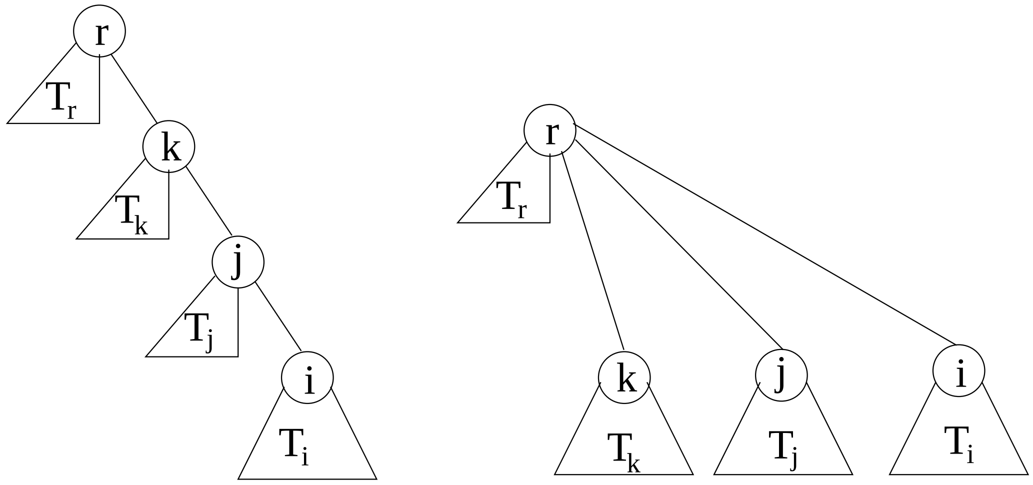
Improvements in UNION

- We can employ the technique of merging smaller sets into larger set to keep the tree balanced.
- If in executing *UNION* root of the tree with fewer vertices made the son of the root of the larger tree, then no tree in forest will have a height greater than or equal to h unless the tree has 2^h vertices.
  - Induction on h. For $h = 0$ it is true.
  - Assume hypothesis to be true for all values of $h \geq 1$.
  - T be tree of height h with fewest vertices.
  - T is the result of a *UNION* of two tree T_1 and T_2
  - Clearly T_1 is a tree of height $h - 1$ with no more vertices then T_2.
  - T_1 has at least 2^{h-1} vertices by induction.
  - T_2 has at least as many vertices as T_1 has.

Cost of FIND

- Modifying *UNION* execution we can reduce the execution of *FIND*s to O($n \log n$).
- So we need another approach in execution of *FIND*s.
- We employ path compression.
- Each time a *FIND*(i) is executed all the intermediate nodes on the tree path $r \xrightarrow{*} i$ and i are made children of r.

Path compression



FIND using path compression

algorithm *FIND*(i) {
    LIST $= \Phi$;
    v=ELEMENT[i];
    while (FATHER[v] $\neq$ 0) {
        add v to LIST;
        v=FATHER[v];
    }
    for (each w in LIST) FATHER[w]=v
}

UNION using trees

- Initially for each element i create a vertex v_i and set SIZE[v_i] $= 1$, NAME[v_i] $= i$ and FATHER[v_i] $= 0$.
- Create ROOT[.] with ROOT[i] pointing to v_i.
- Create ELEMENT[.] where ELEMENT[i] $= v_i$.

algorithm *UNION*(i, j, k) {
    // Assume WLG that size of T_{r_i}
    // is smaller than that of T_{r_j}
    LARGE = ROOT[j];
    SMALL = ROOT[i];
    FATHER[SMALL] = LARGE;
    SIZE[LARGE] = SIZE[LARGE]+SIZE[SMALL];
    NAME[LARGE] = k;
    ROOT[k] = LARGE;
}