

Overview

- Maintaining Disjoint Sets
- Complexity Analysis

Applications

- Equivalence of symbolic addresses (Fortran),
- Minimum spanning trees, and other combinatorial optimization problems,
- Special kind of sorting.

Disjoint Sets - Problem Formulation

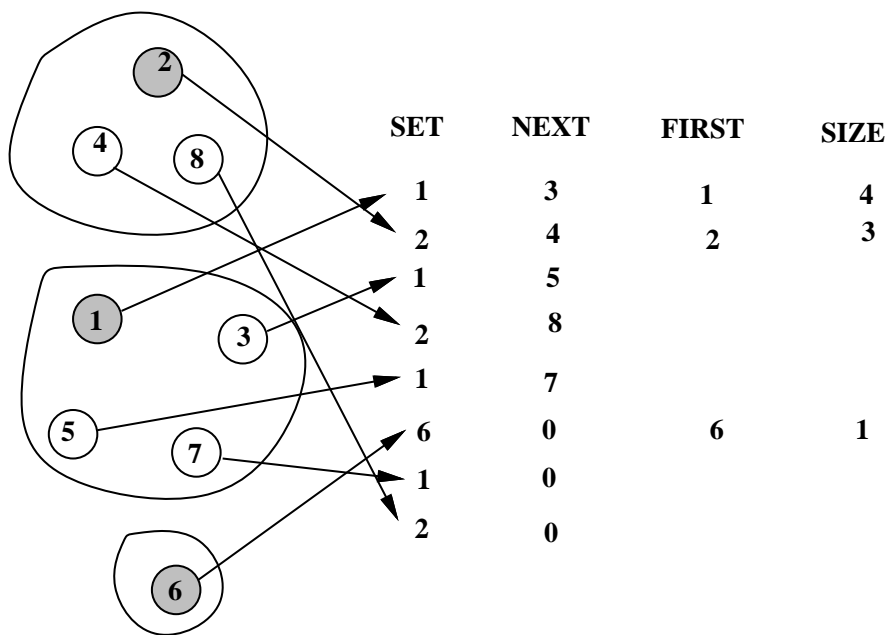
Data structure supporting the representation of disjoint sets.

- Sets are identified by unique representatives called *canonical elements*.
- Elements in sets are assumed to be integers between 1 and n . If not, appropriate pointers need to be maintained.
- Each element can be accessed in $O(1)$ time.
- Three operations must be available:
 - `makeSet(x)`: create a new set containing the single element x ,
 - `find(x)`: return the canonical element of the set containing x ,
 - `link(x,y)`: form a new set that is the union of the two sets whose canonical elements are x and y . A canonical element of the union is selected and returned. Old sets are destroyed.
- How to organize disjoint sets in order to be able to carry out:
 - n `makeSet`,
 - m `find`,
 - k `link`, $k \leq n - 1$,

in any feasible order as quickly as possibly.

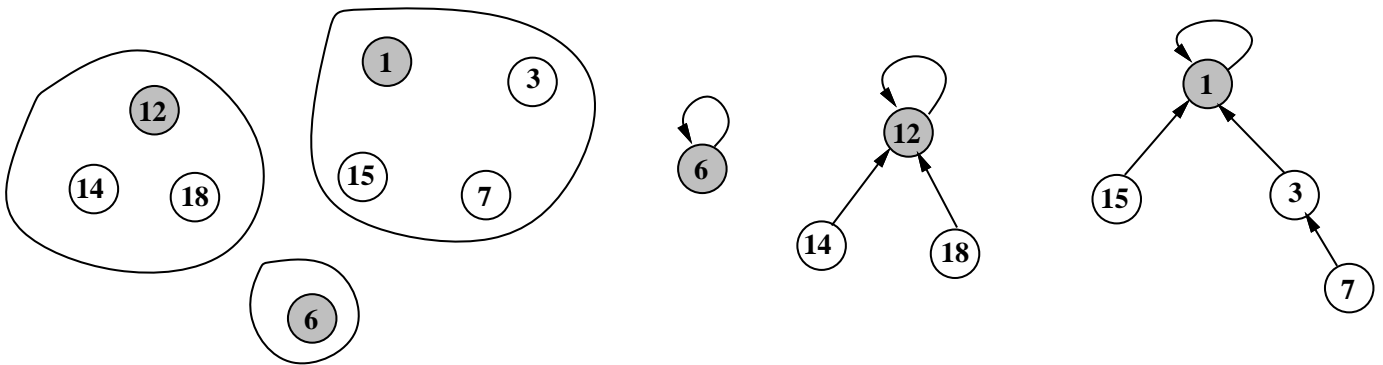
Vector Representation

- One element in a set represents the set itself.
- Four n -vectors can represent sets. Suppose element i belongs to set j .
 - `set(i) := j`,
 - `next(i) := pointer to the next element in j` ,
 - `first(j) := pointer to the first element in j` ,
 - `size(j) := size of the set j` .



Rooted Trees Representation

- nodes of a tree contain elements of a set,
- root contains the canonical element of a set,
- each node x has a pointer $p(x)$ to its parent, except for the root which points to itself.

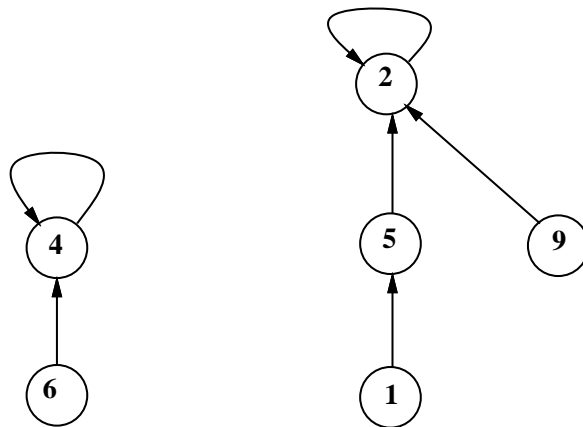


- The same set can be represented by many different trees

Rooted Trees Representation

- `makeSet(x)`: create one-node tree.
- `find(x)`: follow parent pointers from x to the root.
Can become $O(n)$ if not careful.
- `link(x,y)`: let y be the parent of x , and let y be the canonical element of the union set. $O(1)$ time.

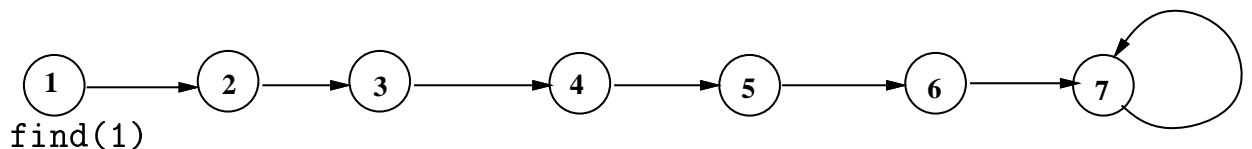
`find(6), link(4,2), find(6)`



What is wrong with this data structure? High trees.

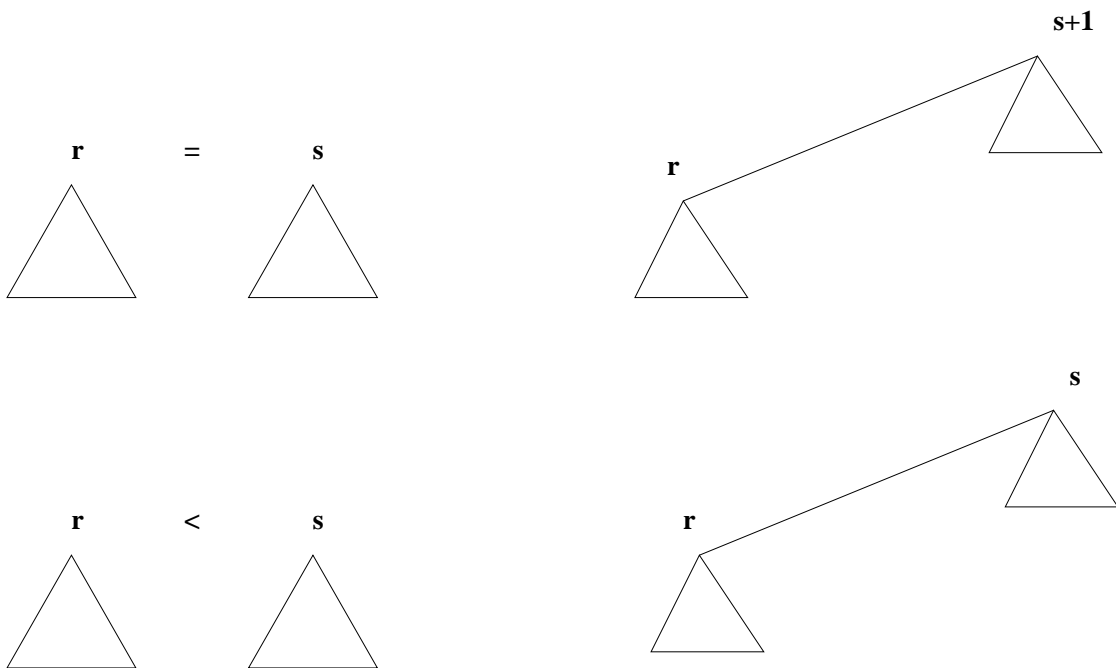
`makeSet(1), makeSet(2), ..., makeSet(n)`

`link(find(1),2), link(find(2),3), ..., link(find(n-1),n)`



Linking by Rank

- roots of one-element trees have rank 0,
- *linking by rank*: During the *link* operation, the root of the tree with higher rank is made the root of the union tree.
- if trees have the same rank, then the rank of the new root (chosen arbitrarily) is increased by 1.



Linking by Rank - Example

link(1,5), link(4,7), link(3,6), link(2,7),
 link(6,8), link(7,5), link(9,6), link(6,5)

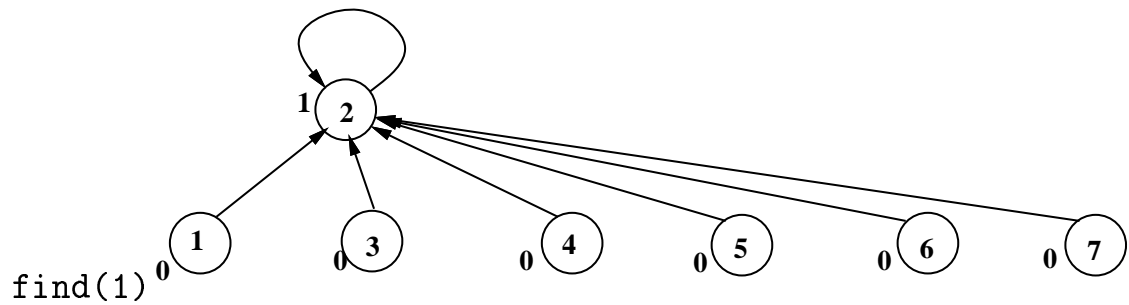
```

      5
     7  6
    1 2 4 9 3 8
  
```

Is it better? How good is it in fact?

makeset(1), makeset(2), ..., makeset(n)

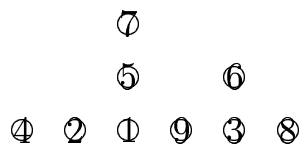
link(find(1),2), link(find(2),3), ..., link(find(n-1),n)



Linking by Weight

- *linking by weight*: During the link operation, the root of the tree with more nodes is made the root of the union tree.

`link(1,5), link(4,7), link(3,6), link(2,7),
link(6,8), link(5,7), link(9,6), link(6,7)`



Some Basic Observations

- Once an item ceases to be the root of a tree, it never becomes a root again. Furthermore, its rank never changes.
- $r(x) \leq r(p(x))$ with the inequality strict if $p(x) \neq x$.
When linking, old root with higher rank becomes a new root; if both roots have the same rank, one of them becomes a new root and its rank is increased by one.
- The number of items $s(x)$ in a tree with root x is at least $2^{r(x)}$.

By induction on the number of link-operations.

- True before the first link.
- Assume that true before the i -th link (of items x and y). Let r_i denote the rank function just before the i -th link.
- if $r_i(x) < r_i(y)$, then after $\text{link}(x, y)$, y is the root, and $r_{i+1}(y) = r_i(y)$. Hence, $s_{i+1}(y) > s_i(y) \geq 2^{r_i(y)} = 2^{r_{i+1}(y)}$.
- if $r_i(x) > r_i(y)$, the symmetric situation arises.
- if $r_i(x) = r_i(y)$, then according to the hypothesis, the tree after $\text{link}(x, y)$ satisfies:

$$s_{i+1}(y) = s_i(x) + s_i(y) \geq 2^{r_i(x)} + 2^{r_i(y)} = 2^{r_i(y)+1} = 2^{r_{i+1}(y)}$$

since $r_{i+1}(y) = r_i(y) + 1$.

- $n \geq s_i(z) \geq 2^{r_i(z)}$ for all i , $0 \leq i < n$, implies $\log n \geq r_i(z)$. Since the rank is strictly increasing when going up the tree, no tree has height greater than $\lceil \log n \rceil$.
Conclusion: `find` requires $O(\log n)$ time.
- Overall complexity: $O(n + m \log n + n - 1) = O(n + m \log n)$. This bound is tight.

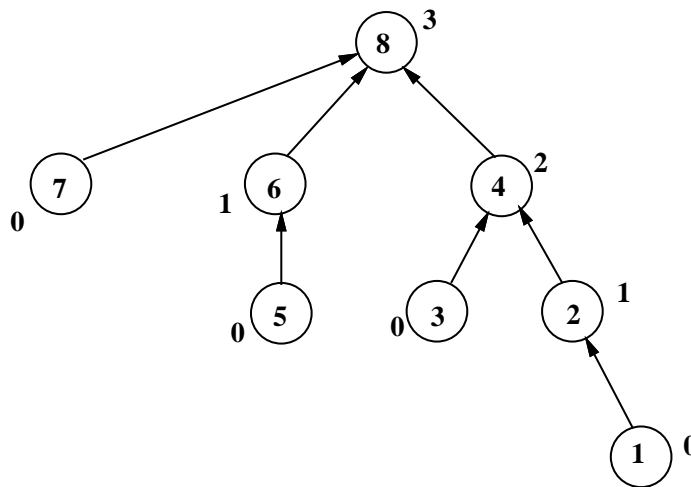
Linking by Rank or Size - Bound Tightness

- A *binomial tree* B_0 consists of a single node.
- A *binomial tree* B_i , $i > 0$, consists of two binomial trees B_{i-1} with root of one being the parent of the root of the other.
- B_i has size 2^i and height i .
- Let $n = 2^i$ for some $i \in \mathcal{N}$. Appropriate sequence of link by rank will result in a binomial tree B_i .
- One path in B_i has $i + 1$ nodes. find of bottom node takes $O(i) = O(\log n)$ time.

```

makeset(1), makeset(2), makeset(3), makeset(4),
makeset(5), makeset(6), makeset(7), makeset(8)
link(1,2), link(3,4), link(5,6), link(7,8)
link(2,4), link(6,8)
link(4,8)
m times find(1)

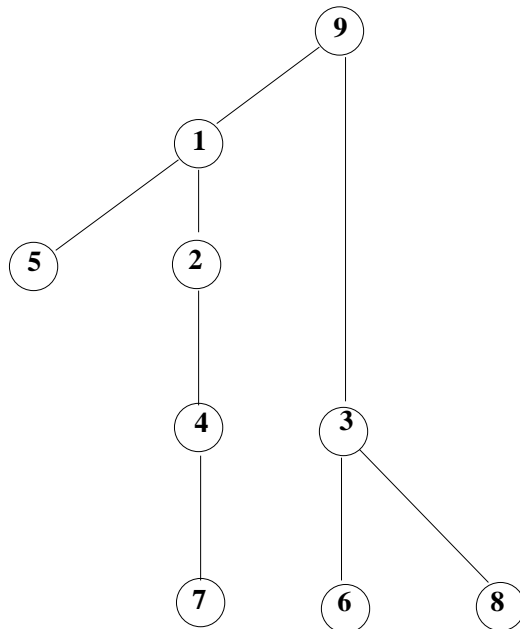
```



Path Compression

- *path compression*: During the `find` operation all traversed nodes are made direct successors (sons) of the root.

- `find(7)`

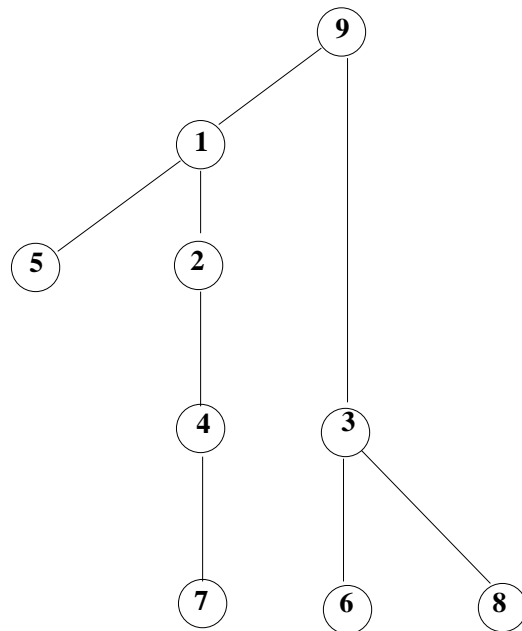


The path from 7 to the root has to be traversed twice.

Find with Halving

- *halving*: During the find every other node on the find path is made a child of its grandparent.

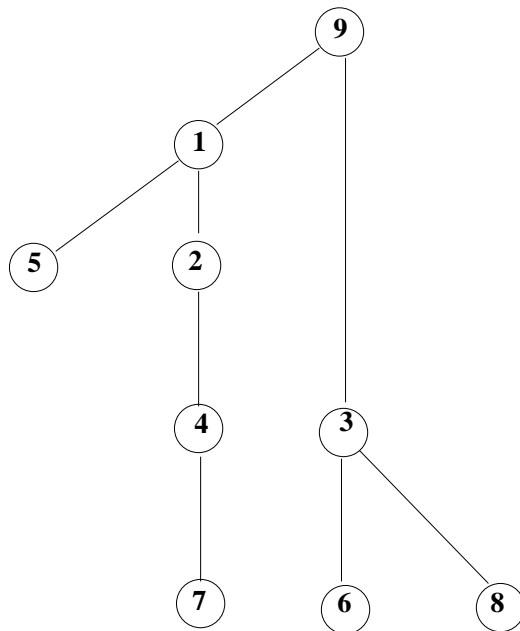
- `find(7)`



Find with Splitting

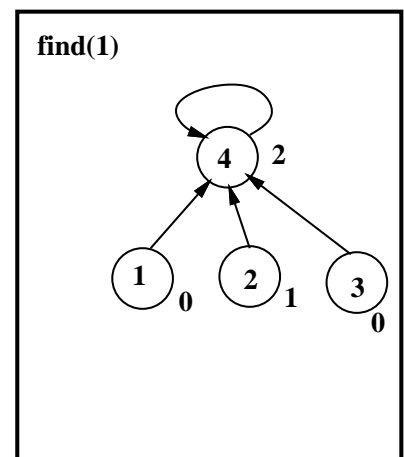
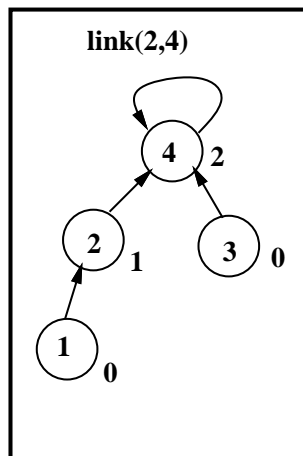
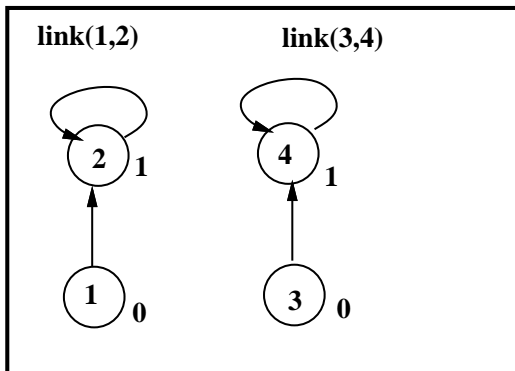
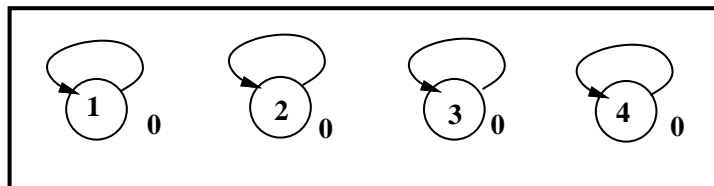
- *splitting*: During the find operation every node on the find path is made a child of its grandparent.

- `find(7)`



Linking by Rank+Compressions - Basics

- Each node has initially rank 0.
- As long as a node is a root, its rank either remains unchanged or it grows (by 1 at a time).
- When a node seizes to be a root, its rank remains unchanged.
- $r(x) < r(p(x))$ unless x is a root.



Linking by Rank+Compressions - Basics

- Without path compressions the rank of a node indicates its height.
- With path compressions the rank of a node is an upper bound on its height. Is it possible that the rank of some node becomes greater than $\log n$?
- No. Path compressions do not change ranks.
- There are no more than $n/2^r$ nodes that will receive rank r anytime during the entire sequence of operations.
 - When a node x receives rank r (x becomes a root), mark x and all its descendants using a label L_r . Since the rank of x is r , at least 2^r nodes are marked. Neither x nor its descendants have been marked by L_r before.
 - If the rank of x changes, it grows. Ranks of descendants of x remain unchanged. Furthermore, all future ancestors of x will have rank greater than r . Hence, neither x nor its descendants can be marked more than once by L_r (when x or any of its descendants changes a father, the rank of the new father is greater than r).
 - Number of nodes which receives the rank r throughout the entire sequence of operations is therefore at most $n/2^r$. Otherwise, more than $2^r n/2^r = n$ nodes would be labeled by L_r , a contradiction.

Ackermann's Function

$$A(i, j) = \begin{cases} 2^j & i = 1, j \geq 1 \\ A(i-1, 2), & i \geq 2, j = 1 \\ A(i-1, A(i, j-1)), & i, j \geq 2 \end{cases}$$

- $A(1, 1) = 2, A(1, 2) = 4, A(1, 3) = 8, A(1, 4) = 16, \dots$
- $A(2, 1) = A(1, 2) = 4,$
- $A(3, 1) = A(2, 2) = A(1, A(2, 1)) = A(1, 4) = 16,$
- $A(4, 1) = A(3, 2) = A(2, A(3, 1)) = A(2, 16) = A(1, A(2, 15)) = \dots$
- $A(2, 3) = A(1, A(2, 2)) = A(1, 16) = 2^{16} = 65536,$
- $A(3, 2) = A(2, A(3, 1)) = A(2, 16) = \dots$

Ackermann's function grows very quickly.

Inverse of Ackermann's Function

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$$

- For all practical purposes $\alpha(m, n)$ is not larger than 4. For instance $\alpha(m, n) \leq 3$ for $n < 2^{16}$.

Iterated Logarithm Function

$$\log^{(i)} n = \begin{cases} n & \text{if } i = 0, \\ \log(\log^{(i-1)} n) & \text{if } i > 0 \text{ and } \log^{(i-1)} n > 0, \\ \text{undefined} & \text{if } i > 0 \text{ and } \log^{(i-1)} n \leq 0 \text{ or } \log^{(i-1)} n \text{ undefined} \end{cases}$$

$$\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\}$$

- $\log^* 2 = 1,$
- $\log^* 4 = 2,$
- $\log^* 16 = 3,$
- $\log^* 65536 = 4,$
- $\log^* 2^{65536} = 5.$

Number of atoms in the universe is approx. $10^{80} < 2^{65536}.$

More Definitions

$$B(j) = \begin{cases} -1 & \text{if } j = -1, \\ 1 & \text{if } j = 0, \\ 2 & \text{if } j = 1, \\ 4 & \text{if } j = 2, \\ 16 & \text{if } j = 3, \\ 65536 & \text{if } j = 4, \\ 2^{2^{j-1}} & \text{if } j \geq 5, j - 1 \text{ times} \end{cases}$$

More generally, $B(j) = 2^{B(j-1)}$ for $j > 0$.

$$\text{block}(j) = [B(j-1) + 1..B(j)], j = 0, 1, \dots, \log^* n - 1$$

```

block(0) = [0..1]
block(1) = [2..2]
block(2) = [3..4]
block(3) = [5..16]
block(4) = [17..65536]

```

Worst-Case Time Complexity

- n `makeSet` requires $O(n)$ time.
- $n - 1$ `link` requires $O(n)$ time.
- Suppose that `find(x0)` is about to be carried out. Let

$$x_0, x_1, x_2, x_3, \dots, x_l$$

denote the path to the root.

- Divide nodes on the path in two groups:
 - Group A: nodes with ancestor's rank in the next block. In addition: child of the root.
 - Group B: remaining nodes.
- there are $\log^* n$ blocks. Processing of nodes of type A during each `find` takes at most $O(\log^* n + 1)$ time. There are m `find`. Processing of nodes of type A takes in total $O(m \log^* n)$ time.
- How many nodes in group B is processed during the m `find`?

Worst-Case Time Complexity (cont.)

- $N(j)$: number of nodes with ranks in $\text{block}(j)$.

$$N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$$

- For $j = 0$, we have $B(j-1) + 1 = 0$, $B(j) = 1$. Hence,

$$N(0) \leq \frac{n}{2^0} + \frac{n}{2^1} = \frac{3n}{2} = \frac{3n}{2B(0)}$$

- For $j \geq 1$, we have

$$\begin{aligned} N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} < \\ &\frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} = \\ &\frac{n}{2^{B(j-1)}} = \frac{n}{B(j)} < \frac{3n}{2B(j)} \end{aligned}$$

- Suppose that a group B node is in block $B(j)$. During each find involving this node, its rank is unchanged. Hence, it remains in the block $B(j)$. But its father changes. After at most

$$B(j) - B(j-1) - 1$$

find, the father will belong to another block. Consequently, our node will become of type A.

- Let $P(n)$ denote the total number of type B nodes encountered during the m find.

$$\begin{aligned} P(n) &\leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} (B(j) - B(j-1) - 1) \leq \\ &\sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j) = \frac{3}{2} n \log^* n \end{aligned}$$

Algorithms

Disjoint Sets

Summary

| | makeset | find | link | total |
|-------------------|---------|-------------|--------|-----------------------|
| vector | $O(1)$ | $O(1)$ | $O(n)$ | $O(m + n \log n)$ |
| tree | $O(1)$ | $O(n)$ | $O(1)$ | $O(mn)$ |
| link by rank | $O(1)$ | $O(\log n)$ | $O(1)$ | $O(n + m \log n)$ |
| with compressions | $O(1)$ | $O(\log n)$ | $O(1)$ | $O((m + n) \log^* n)$ |