Union/Find

Borrowed from Oberlin CS280, 2001

Union/Find

- Initial state:
  - A union/find structure begins with **n** elements, each considered to be a one element set
- Functions:
  - **Union(S_i,S_j)**: Takes any two sets and unions them into one
  - **Find**: Takes any element in the structure and returns the index (name?) of the set

Basic Notation

- The elements in the structure will be numbered from **0** to **n-1**
- Each set will be referred to by the number of one of the element it contains
  - Initially we have sets $\mathbf{S_1,S_2,\ldots,S_{n-1}}$
  - If we were to call **Union($\mathbf{S_2,S_4}$)**, these sets would be removed from the list, and the new set would now be called either $\mathbf{S_2}$ or $\mathbf{S_4}$

First Attempt

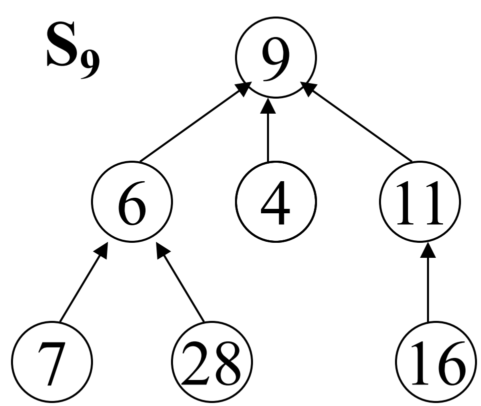
- A basic strategy might work like this:
  - Represent the Union/Find structure as an array **arr** of **n** elements
  - **arr[i]** contains the set number of element **i**
    - Initially, **arr[i]=i** (since each element **i** is in set S_i
  - **find(i)** just returns the value of **arr[i]**
  - To perform **Union(S_i,S_j)**:
    - For every **k** such that **arr[k]=S_j**, set **arr[k]=i**

Analysis

- The worst-case analysis of each method:
  - **Find(i)** takes **O(1)** time
  - **Union(S_i,S_j)** takes **Θ(n)** time
    - It will always have to look at every element of the list – so it can never better
- Note that the amortized analysis won't do any better
  - A sequence of **n Unions** will take **$\Theta(n^2)$** time

Visualization

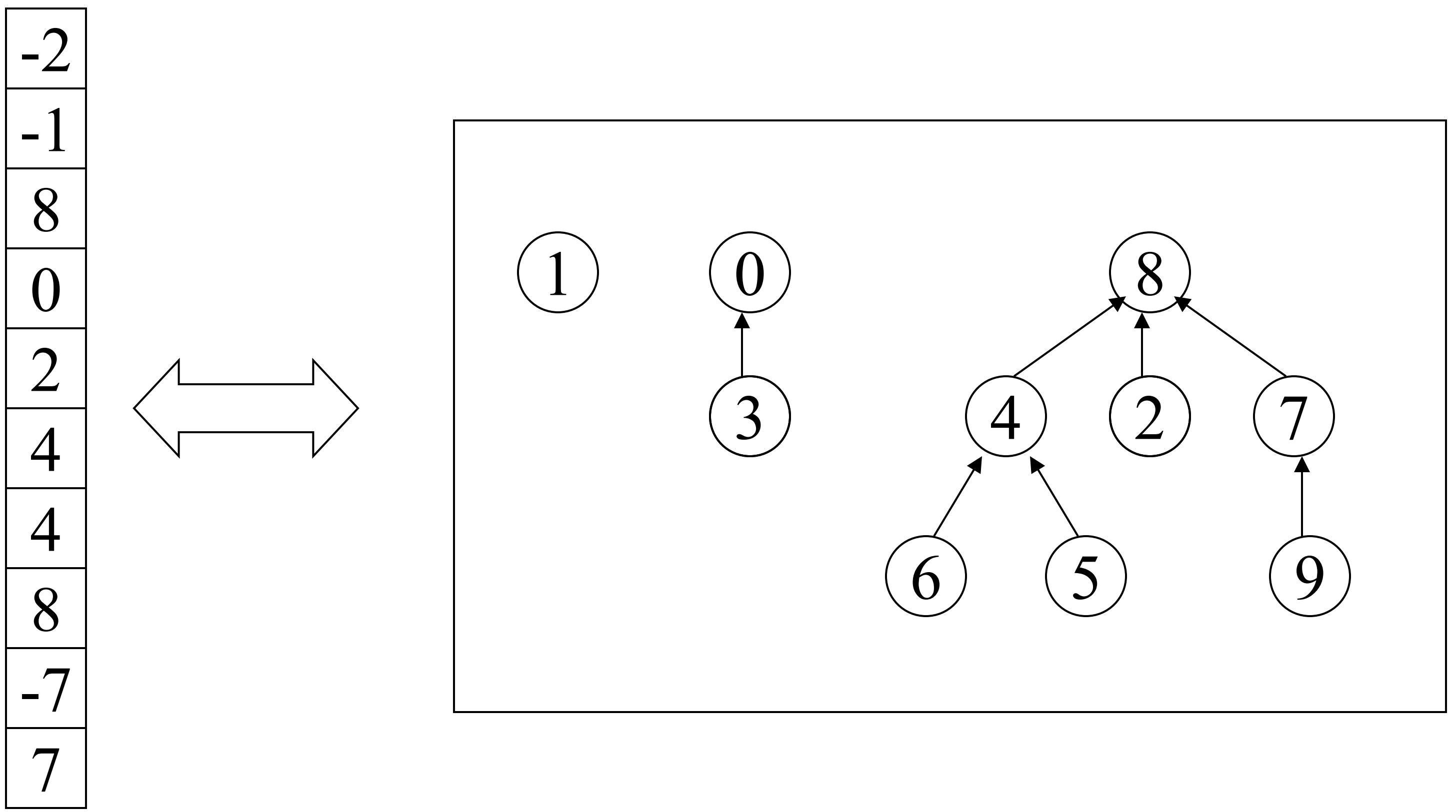
- In order to do better, we will start visualizing the structure as a forest:
  - We visualize each element as a node
  - A set will be visualized as a directed tree
    - Arrows will point from child to parent
  - The set will be referred to by its root



Implementation

- In actual implementation, we will still represent this with an array of **n** nodes
  - If element **i** is a “root node” (the smallest in the set containing **i**), then **arr[i]= -s**, where **s** is the size of that set
  - Otherwise, **arr[i]** is the index of **i**’s “parent”

Implementation and Visualization



New Methods

- A simple way to perform **U(S_i,S_j)**: Make S_i the parent of S_j
  - Implementation: set **arr[i]=arr[i]+arr[j]** and then set **arr[j]=i**
    - Note that S_k denotes the set with **k** as its root, both both **arr[i]** and **arr[j]** are negative – to this works
- A simple way to perform **find(i)**:
  - If **arr[i] < 0**, return S_i, else return the value **find(arr[i])**
  - Visually, we are searching up the tree

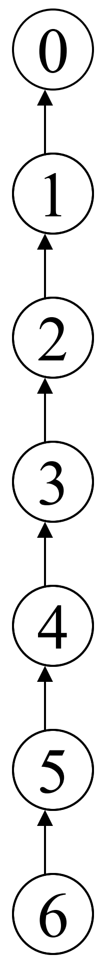
Analysis

- **Union(S_i,S_j)** can be done in two steps:
  - **O(1)** time
- **Find(i)** is dependent on the depth of **i**
  - Is there a bound on the height of any tree?
  - Consider the sequence of operations:

    Union(S_0,S_1), Union(S_1,S_2), …,

    Union(S_{n-2},S_{n-1}),

Result



Analysis

- Worst case:
  - **Union(S_i,S_j)** take **O(1)** time
  - **Find(i)** takes **O(n)** time
- Can we do better in an amortized analysis?
  - What is the maximum amount of time **n** operations could takes us?
  - Suppose we perform **n/2** unions followed by **n/2** finds
    - The **n/2** unions could give us one tree of height **n/2-1**
    - Thus the total time would be $\mathbf{2(n/2) + (n/2)(n/2) = O(n^2)}$
- This strategy doesn't really help

WeightedUnion

- Just a small change in the Union function will help:
  - **WeightedUnion(S_i,S_j)**: Make the root of the smaller tree the child of the root of the larger tree
  - Implementation:
    - If **-arr[i] < -arr[j]**, then set **arr[i]** to **arr[i]+arr[j]** and set **arr[j]**to **I**
    - Else, set **arr[j]** to **arr[i]+arr[j]** and set **arr[j]** to **i**

New Bound on **h**

- Lemma: Assume we start with a Union/Find structure where each set has **1** node, and perform a sequence of **WeightedUnion**s. Then any tree tree **T** of **m** nodes has a height no greater than $\lfloor \mathbf{\log_2 m} \rfloor$.
- We prove this by (strong) induction on **m**.

Proof

- Base case: If **m=1**, then this is clearly true
- Assumption: Assume it is true for all trees of size **m-1** or less
- Proof: Let **T** be a tree of **m** nodes created by a sequence of **WeightedUnion**s. Consider the last union: **Union(S_j,S_k)**. Assume S_j is the smaller tree. If S_j has **a** nodes, then S_k has **m-a** nodes, and $1 \le a \le m/2$.

Proof (continued)

- The height of **T** is either:
  - The height of $\mathbf{T_k}$
  - One more than the height of $\mathbf{T_j}$
- Since $\mathbf{a \le m\text{-}a \le m\text{-}1}$, the assumptions applies to both $\mathbf{T_k}$ and $\mathbf{T_j}$
  - If **T** has the height of $\mathbf{T_k}$, then

    $$\mathbf{h \le \lfloor \log_2(m\text{-}a) \rfloor \le \lfloor \log_2 m \rfloor}$$

  - If **T** is one greater than the height of $\mathbf{T_j}$:

    $$\mathbf{h \le \lfloor \log_2 a \rfloor + 1 \le \lfloor \log_2 m/2) \rfloor + 1 \le \lfloor \log_2 m \rfloor}$$

Conclusion

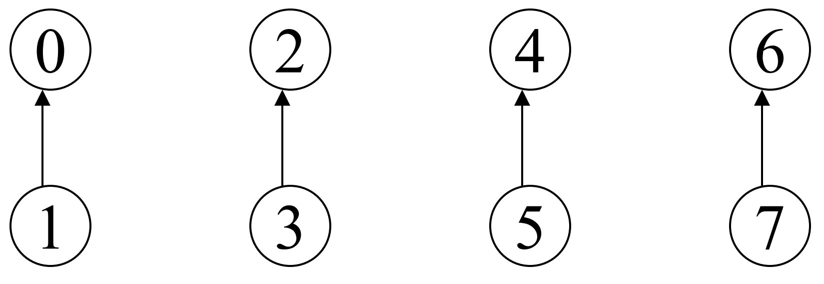
- Hence the height of each tree is bound by $\lfloor \mathbf{log_2 m} \rfloor \mathbf{+1}$
- Question: Is this *tight*?
  - Is there a sequence of operations that result in a tree of exactly this height (for any **m**)
  - Yes: "pair them off"

    Union(S_0,S_1), Union(S_2,S_3), Union(S_4,S_5), Union(S_6,S_7), Union(S_4,S_6), Union(S_0,S_4),

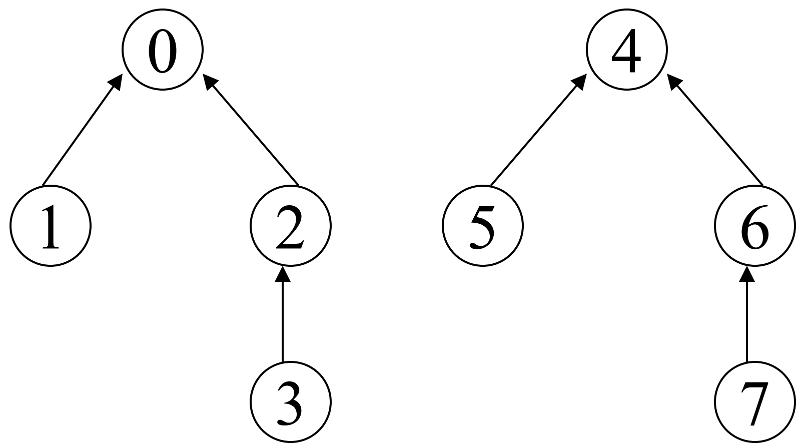
Example

0 1 2 3 4 5 6 7

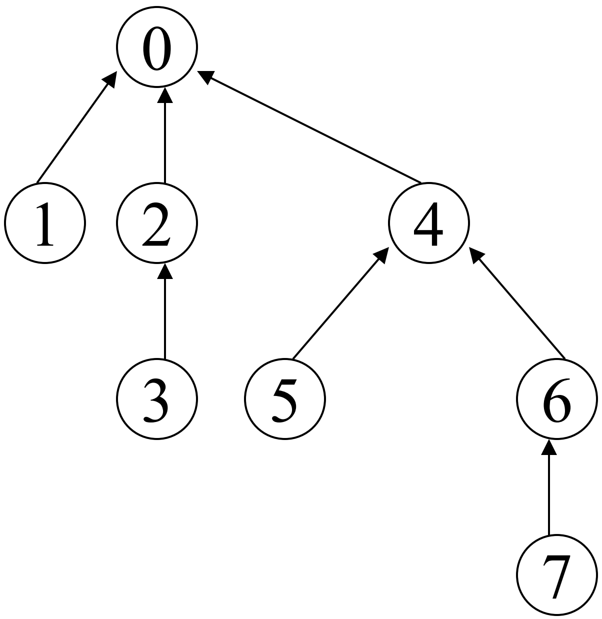
Example



Example



Example



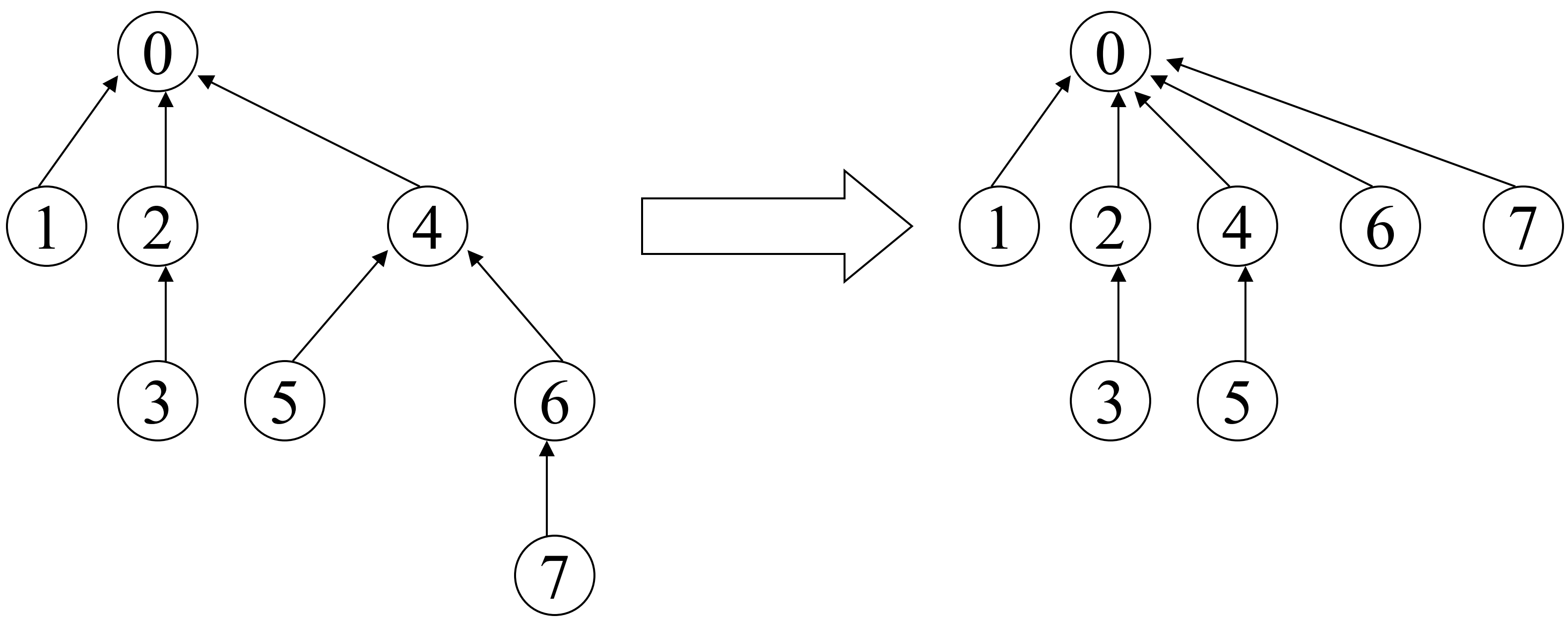
Analysis

- Worst case:
  - **Union** is still **O(1)**
  - **Find** is now **O(log n)**
- Amortized case:
  - A "worst amortized case" can be achieved if we perform **n/2** unions and **n/2** finds
  - Take **O(n log n)** time
- Conclusion: This is better, *but we can improve it further*

CollapsingFind

- Suppose that **Find(i)** also restructured the tree as it explored it
  - Specifically: we can reset every node encountered on the path to point directly to the root node
  - This will require two passes over the path, but the next **Find** involving any of these nodes will be faster

Find(7)



Summary

- Union/Find structures can be made very efficient (in an amortized sense)
  - Use **WeightedUnion**
  - Use **ColapsingFind**
- While a sequence of **n** operations does not technically run in linear time, you can't get much closer