COL106
Data Structures and Algorithms

Subodh Sharma and Rahul Garg

Announcements

Exam Syllabus

- The syllabus is all that has been covered in the class
- The major and lab tests may not have questions from all the topics
- But you are expected to study all the topics covered in the classes

Exam Syllabus

- The syllabus is all that has been covered in the class
- The major and lab tests may not have questions from all the topics
- But you are expected to study all the topics covered in the classes
- Some of the topics introduced will be covered in more depth in subsequent courses
- Use this as a learning opportunity

Union-Find

Based on slides by: Goodrich and Tamasia, CSE 373 University of Washington, Kong Lwang, Xzou, Purdue University Indianapolis,

Applications

- Finding connected components in a Graph

- Example: A social network

- Given a set, S, of n people.

- A social network for S comprises a set E, of edges or ties between pairs of people (such as in a friendship network, like Facebook, or tagging network by twitter)

- A **connected component** in a network is a subset, T of S such that:
  - Every two people in T are connected through a sequence of friendship relationships
  - No one in T is friends with anyone outside of T.

Union-Find Operations

- A **partition** or **union-find** structure is a data structure supporting a collection of disjoint sets subject to the following operations:

- makeSet(e): Create a singleton set containing the element e and return the position storing e in this set

- union(A,B): Return the set A U B, naming the result "A" or "B"

- find(e): Return the set containing the element e

Connected Components Algorithm

Algorithm UFConnectedComponents(S, E):

    Input: A set, S, of n people and a set, E, of m pairs of people from S defining pairwise relationships

    Output: An identification, for each x in S, of the connected component containing x

    for each x in S **do**

        makeSet(x)

    for each (x, y) in E **do**

        if find(x) $\neq$ find(y) **then**

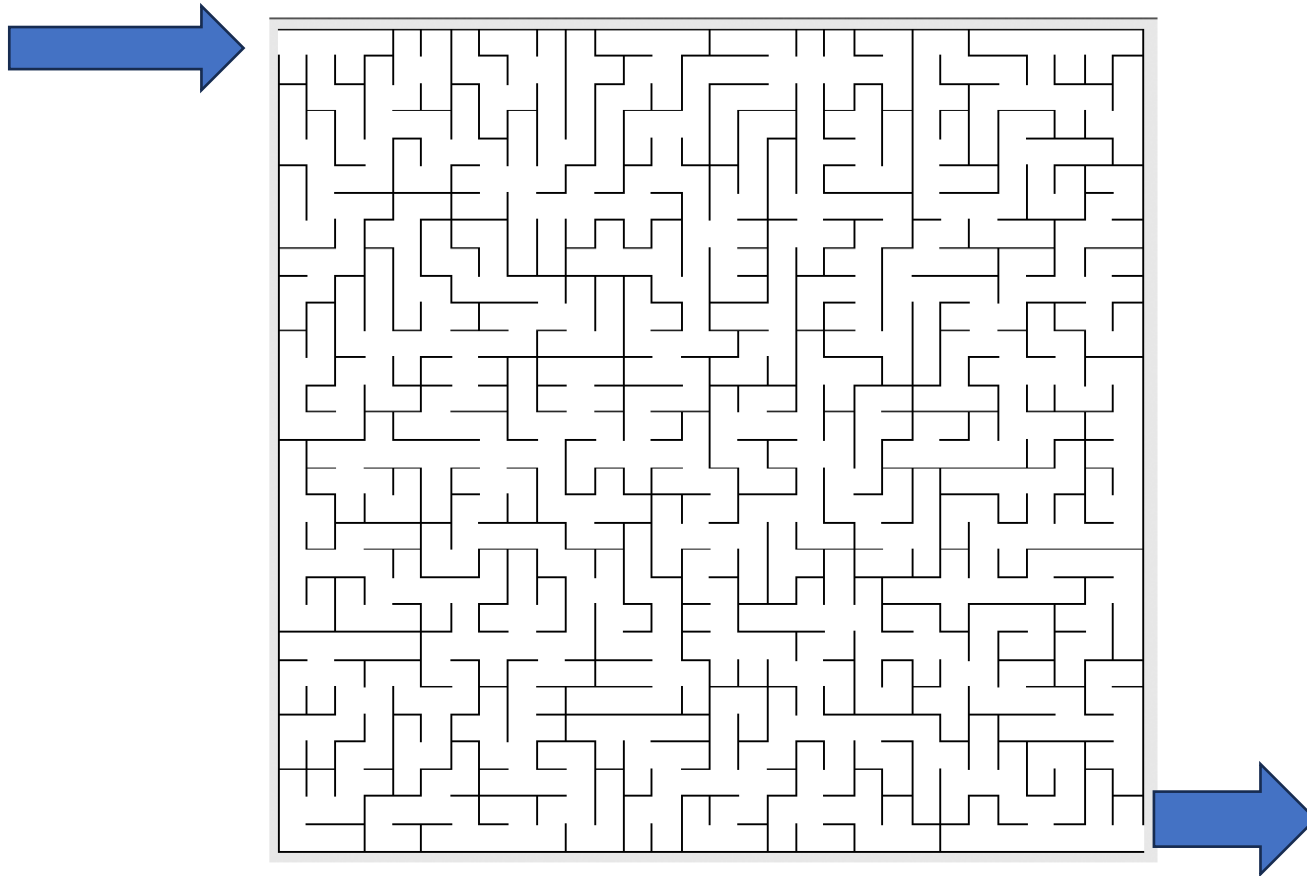   union(find(x), find(y))

    for each x in S **do**

        Output "Person x belongs to connected component" find(x)

- The running time of this algorithm is O(t(n,n+m)), where t(j,k) is the time for k union-find operations starting from j singleton sets.

Maze Construction

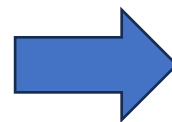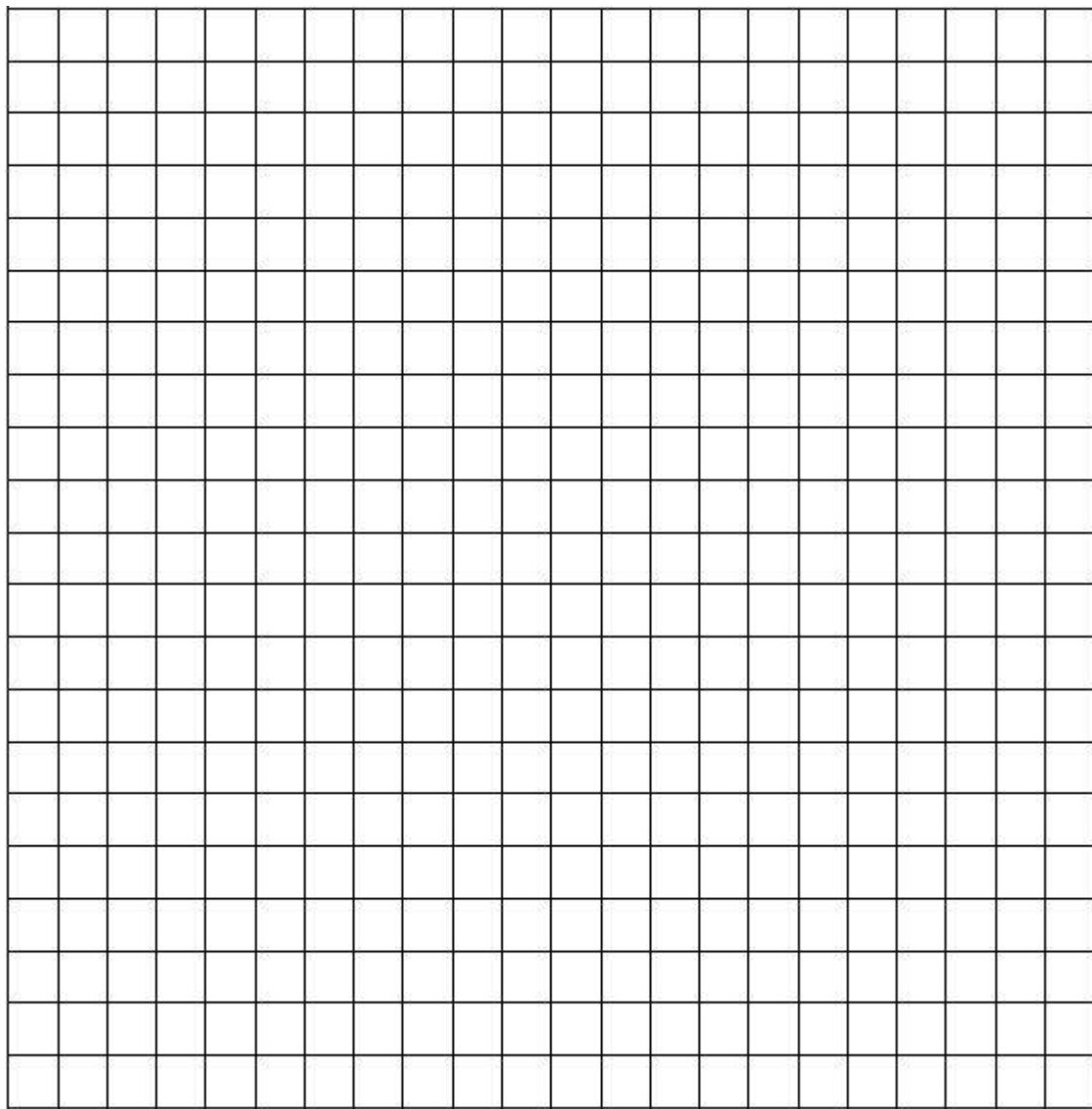- Problem: Construct a good maze that connects entry point to the exit point

How to Construct a Maze?

How to Construct a Maze?

- Hint: Model it as a connectivity problem in a Graph

How to Construct a Maze?

- Hint: Model it as a connectivity problem in a Graph

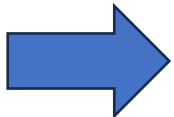- What are the vertices?

- What are the edges?

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
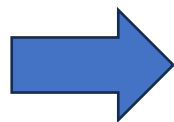
How to Construct a Maze?

- Vertices are cells
- Two vertices are connected by an edge if:
  - They are adjacent
  - The adjoining wall is not there
- Keep on adding the edges till vertex 1 (entry point) is connected to vertex 100 (exit point)
- Do not add edges within the same connected components
- Construct a spanning tree

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

A Maze Generator

Algorithm MazeGenerator(G, E):

    Input: A grid, G, consisting of n cells and a set, E, of m "walls," each of which divides two cells, x and y, such that the walls in E initially separate and isolate all the cells in G

    Output: A subset, R of E, such that removing the edges in R from E creates a maze defined on G by the remaining walls

    while R has fewer than $n - 1$ edges **do**

        Choose an edge, (x, y), in E uniformly at random from among those previously unchosen

        if find(x) $\neq$ find(y) **then**

            union(find(x), find(y))

            Add the edge (x, y) to R

    return R

Data Structures for Union Find

- Linked lists?

- Trees?

- Others?

Disjoint Sets with Linked Lists

- Approach 1: Create a linked list for each set.
    - last/first element is representative
    - cost of union? find?

- Approach 2: Create linked list for each set. Every element has a reference to its representative.
    - last/first element is representative
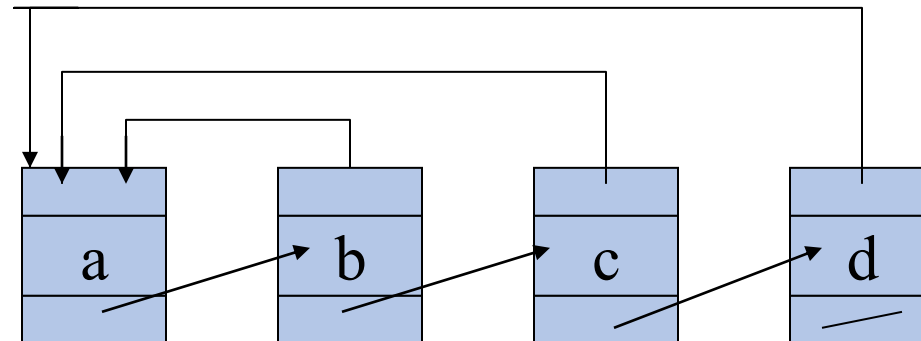   - cost of union? find?

Disjoint Sets with Linked Lists

- Approach 1: Create a linked list for each set.
  - last/first element is representative
  - cost of union? find?
- Union:

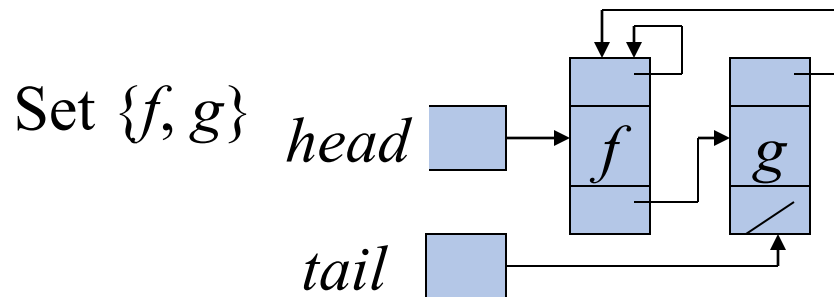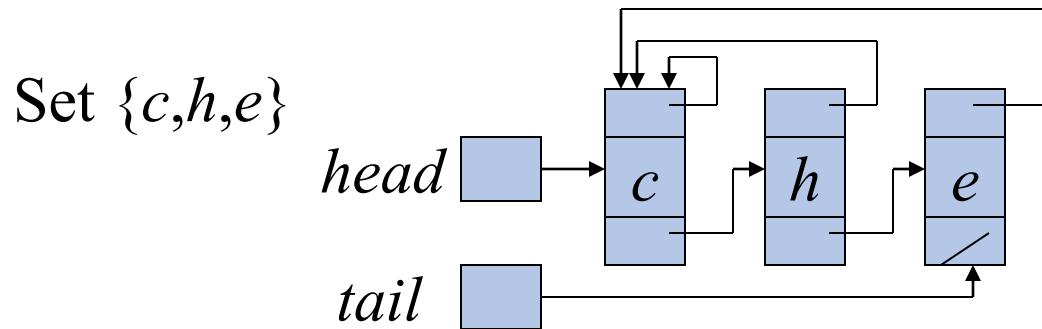# Disjoint Sets with Linked Lists

- Approach 1: Create a linked list for each set.
    - last/first element is representative
    - cost of union? find?
- Find:

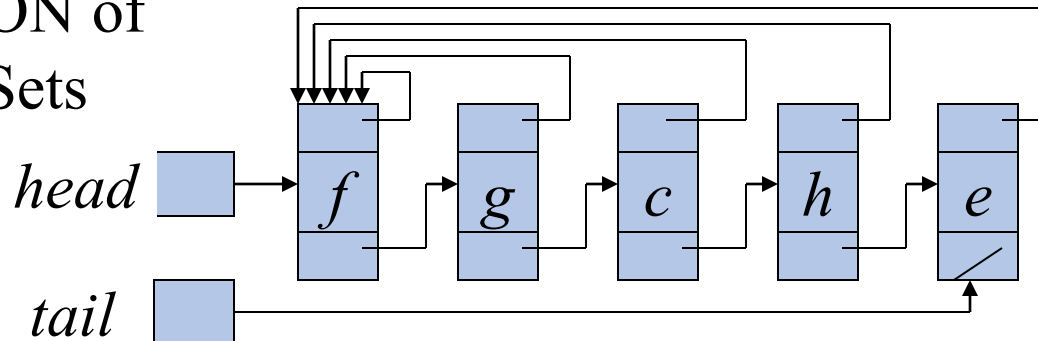# Disjoint Sets with Linked Lists

- Approach 2: Create linked list for each set. Every element has a reference to its representative.
  - Last/first element is representative
 - Cost of union? find?

Linked-lists for two sets

Set {*c*,*h*,*e*}

head → c → h → e

tail

Set {*f*, *g*}

head → f → g

tail

UNION of two Sets

head → f → g → c → h → e

tail

UNION Implementation

- A simple implementation: UNION(x,y) just appends x to the end of y, updates all back-to-representative pointers in x to the head of y.

- Each UNION takes time linear in the x's length.

- Suppose n MAKE-SET(x_i) operations ($O(1)$ each) followed by n-1 UNION
  - UNION(x_1, x_2), $O(1)$,
  - UNION(x_2, x_3), $O(2)$,
  - .....
  - UNION(x_{n-1}, x_n), $O(n$-1$)$

- The UNIONs cost $1+2+...+n$-1$=\Theta(n^2)$

- So $2n$-1 operations cost $\Theta(n^2)$, average $\Theta(n)$ each.

- Not good!! How to solve it ???

Linked-lists: Approach 2b

- For union, keep the identity of larger set
- Modify the elements of the smaller set to point to the larger set's ID
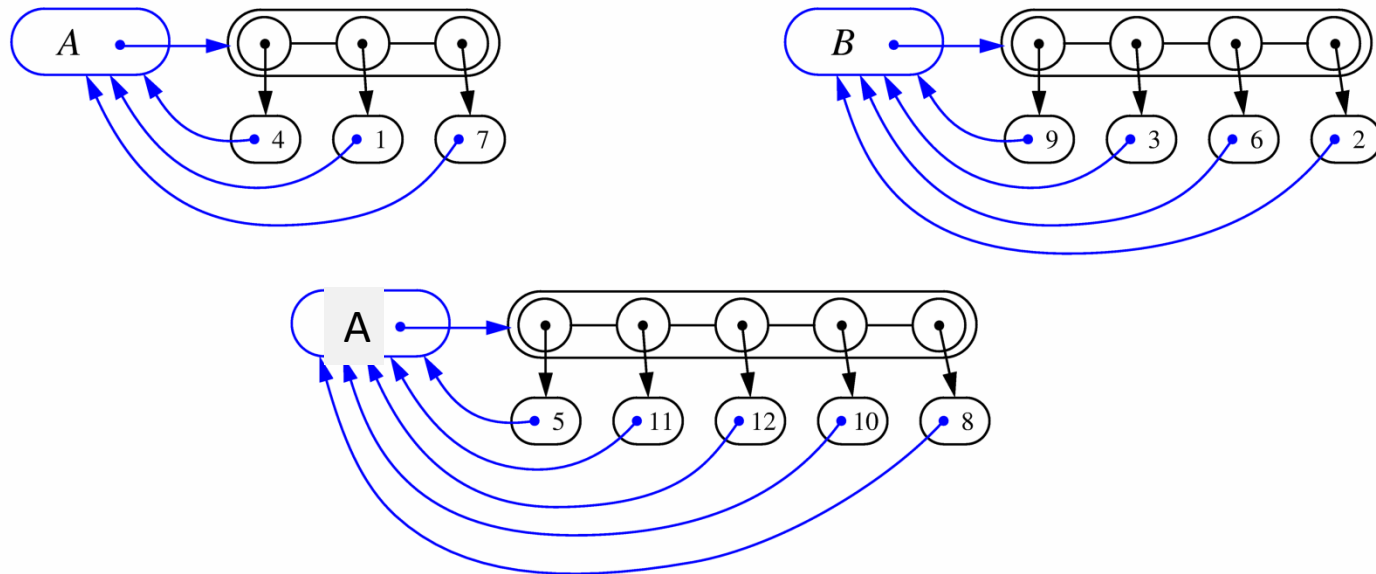
Analysis of Approach 2b

◆ When doing a union, always move elements from the smaller set to the larger set

- Each time an element is moved it goes to a set of size at least double its old set
- Thus, an element can be moved at most O(log n) times

◆ Total time needed to do n unions and m finds ??

Analysis of Approach 2b

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most O(log n) times
- Total time needed to do n unions and m finds is O(n log n + m).
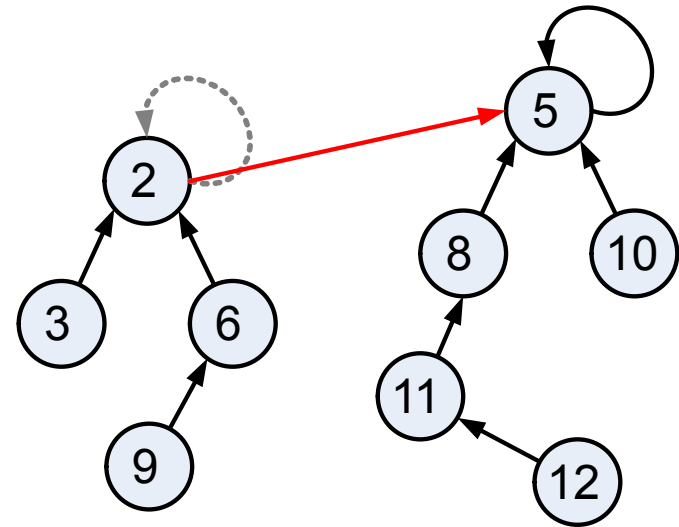
Tree-based Implementation

- Each element is stored in a node, which contains a pointer to parent
- A node v whose parent pointer points back to v is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
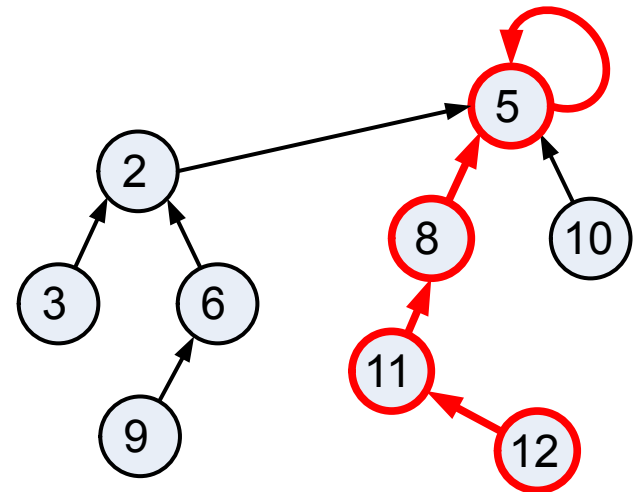- For example: The sets "1", "2", and "5":
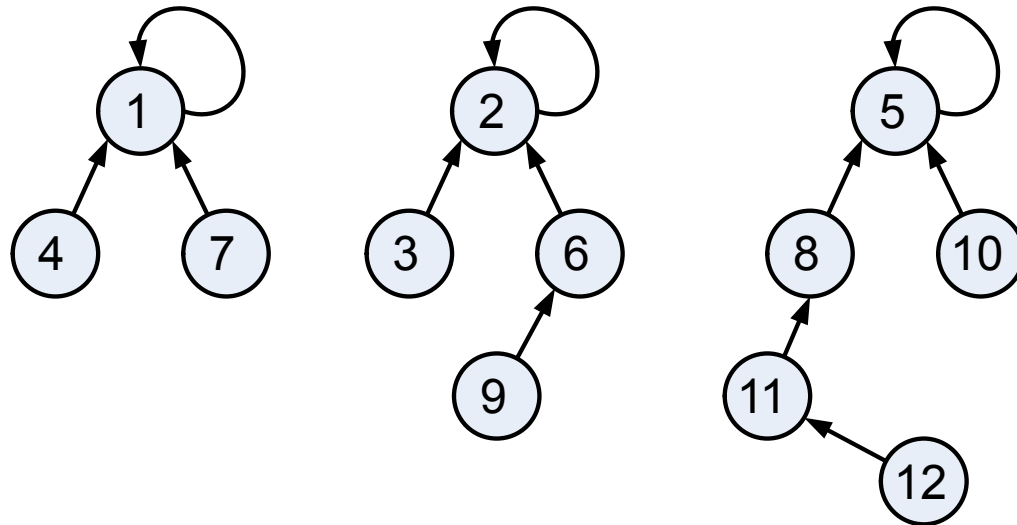
Union-Find Operations

- To do a union, simply make the root of one tree point to the root of the other

- To do a find, follow set-name pointers from the starting node until reaching a node whose parent pointer refers back to itself
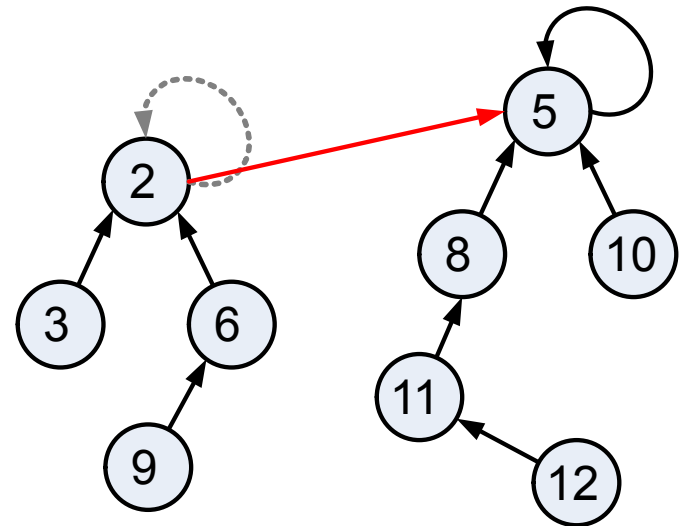
Union-Find Heuristic 1a

- **Union by size**:
  - Maintain a variable size at every node
  - When performing a union, make the root of smaller tree point to the root of the larger
  - Update the size of the root

- Implies O(n log n) time for performing n union-find operations:
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most O(log n) pointers for any find.
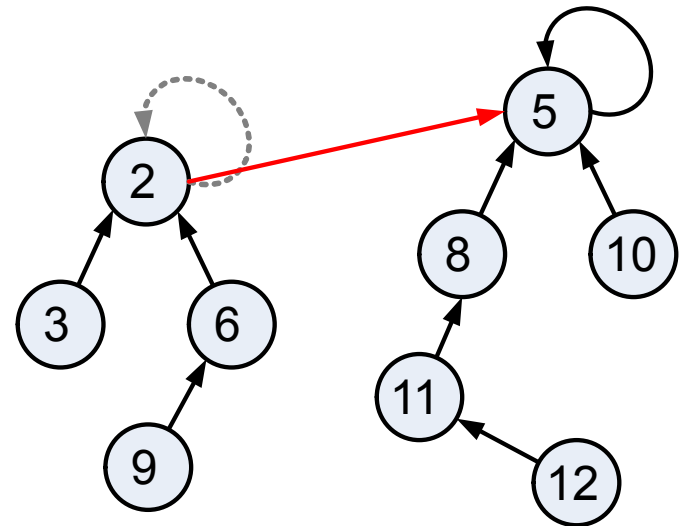
Union-Find Heuristic 1b

- **Union by rank**:
  - Maintain a rank at every node initialized to zero
  - When performing a union, make the root of smaller-rank tree point to the root of the larger rank three
  - If rank of both the trees are the same, add one to the rank of the root

- Implies O(n log n) time for performing n union-find operations:
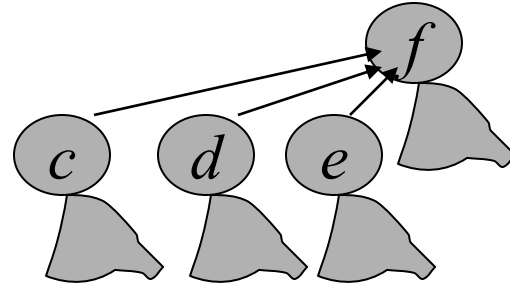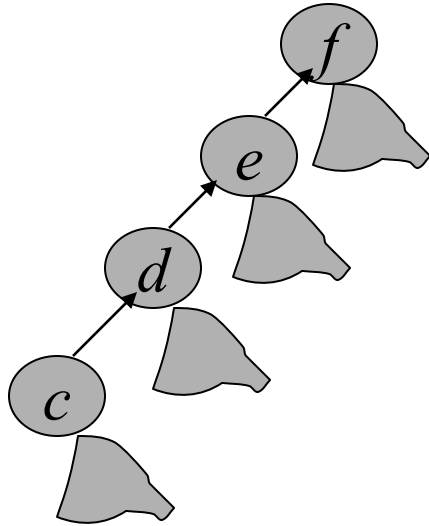 - Why?

Union-Find Heuristic 2

- **Path compression**:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root
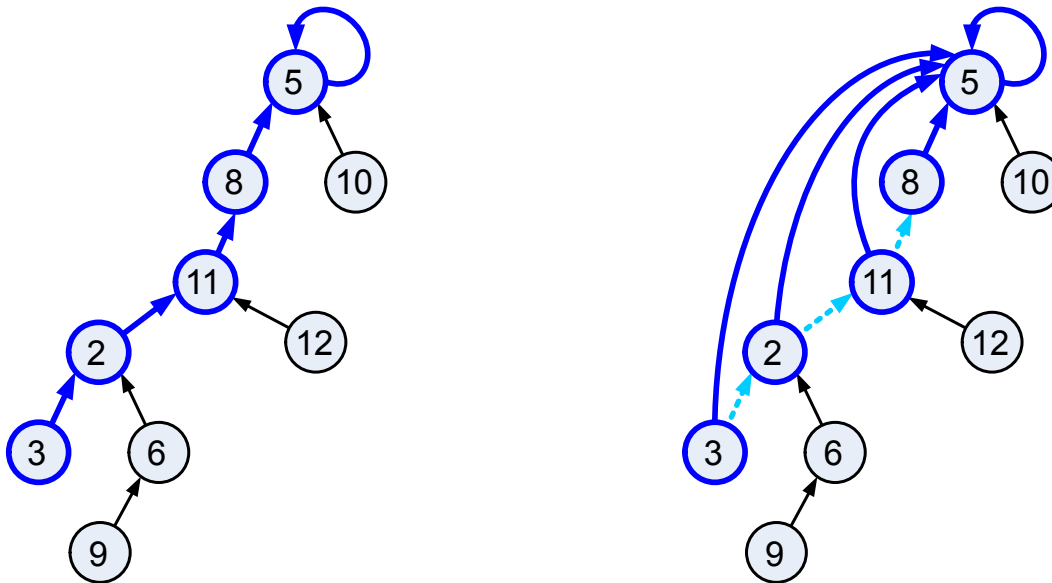
- Implies a fast "almost linear" time for n union-find operations.

Path Compression

Algorithm for Disjoint-Set Forest

MAKE-SET(x)
1. $p[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

UNION(x,y)
1. LINK(FIND-SET(x),FIND-SET(y))

LINK(x,y)
1. **if** $rank[x] > rank[y]$
2. **then** $p[y] \leftarrow x$
3. **else** $p[x] \leftarrow y$
4.   **if** $rank[x] = rank[y]$
5.   **then** $rank[y]++$

FIND-SET(x)
1. **if** $x \neq p[x]$
2.   **then** $p[x] \leftarrow$ FIND-SET($p[x]$)
3. **return** $p[x]$

Worst case running time for m MAKE-SET, UNION, FIND-SET operations is:
$O(m\alpha(n))$ where $\alpha(n) \leq 4$. So nearly linear in m.

Analysis of Union by Rank with Path Compression (by amortized analysis)

- Discuss the following:
  - A very quickly growing function and its very slowly growing inverse
  - Properties of Ranks
  - Proving time bound of $O(m\alpha(n))$ where $\alpha(n)$ is a very slowly growing function.

A Fun Exercise

A Fun Exercise

What is the largest number you can imagine represented using less than 100 characters?

A Fun Exercise

What is the largest number you can imagine represented using less than 100 characters?

Start thinking algorithmically

A Fun Exercise

What is the largest number you can imagine represented using less than 100 characters?

Start thinking algorithmically and non-linearly

Largest number you can imagine

Ackermann Function

Let

$$A_0(x) = x + 1$$

$$A_{i+1}(x) = A_i^{(x)}(x)$$

where, f(k) denotes k applications of the function f to x

$$f^{(1)}(x) = f(x)$$

$$f^{(k)}(x) = f\left(f^{(k-1)}(x)\right)$$

$$A_9(99)$$

4

10

17

13

21

7

4 + 10+17+13+21+7

Total: 72 characters

Ackermann Function

The version of the Ackermann function we use is based on an indexed function, A_i, which is defined as follows, for integers $x \geq 0$ and $i > 0$:

$$A_0(x) = x + 1$$
$$A_{i+1}(x) = A_i^{(x)}(x),$$

where $f^{(k)}$ denotes the k-fold composition of the function f with itself. That is,

$$f^{(0)}(x) = x$$
$$f^{(k)}(x) = f(f^{(k-1)}(x)).$$

So, in other words, $A_{i+1}(x)$ involves making x applications of the A_i function on itself, starting with x. This indexed function actually defines a progression of functions, with each function growing much faster than the previous one:

- $A_0(x) = x + 1$, which is the increment-by-one function
- $A_1(x) = 2x$, which is the multiply-by-two function
- $A_2(x) = x2^x \geq 2^x$, which is the power-of-two function
- $A_3(x) \geq 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ (with x number of 2's), which is the tower-of-twos function
- $A_4(x)$ is greater than or equal to the tower-of-tower-of-twos function
- and so on.

Ackermann Function

Let

$$A_0(x) = x + 1$$

$$A_{i+1}(x) = A_i^{(x)}(x)$$

$A_1(x) = A_0^{(x)}(x)$

$$=$$

Ackermann Function

Let

$$A_0(x) = x + 1$$

$$A_{i+1}(x) = A_i^{(x)}(x)$$

$A_1(x) = A_0^{(x)}(x)$

$\qquad = 2x$

Ackermann Function

Let

$$A_0(x) = x + 1$$

$$A_{i+1}(x) = A_i^{(x)}(x)$$

$A_2(x) = A_1^{(x)}(x)$

$\quad = A_1^{(x-1)}(A_1(x))$

$\quad = A_1^{(x-1)}(2x)$

$\quad = A_1^{(x-2)}(2^2x)$

$\quad = x2^x$

Ackermann Function

Let

$$A_0(x) = x + 1$$

$$A_{i+1}(x) = A_i^{(x)}(x)$$

$A_2(x) = A_1^{(x)}(x)$

$\quad = A_1^{(x-1)}(A_1(x))$

$\quad = A_1^{(x-1)}(2x)$

$\quad = A_1^{(x-2)}(2^2x)$

$\quad = x2^x$

$\quad \geq 2^x$

Ackermann Function

Let

$$A_0(x) = x + 1$$

$$A_{i+1}(x) = A_i^{(x)}(x)$$

$A_3(x) = A_2^{(x)}(x)$

$\quad = A_2^{(x-1)}(A_2(x))$

$\quad \geq A_2^{(x-1)}(2^x)$

$\quad \geq$

$\quad \geq$

Ackermann Function

The version of the Ackermann function we use is based on an indexed function, A_i, which is defined as follows, for integers $x \geq 0$ and $i > 0$:

$$A_0(x) = x + 1$$
$$A_{i+1}(x) = A_i^{(x)}(x),$$

where $f^{(k)}$ denotes the k-fold composition of the function f with itself. That is,

$$f^{(0)}(x) = x$$
$$f^{(k)}(x) = f(f^{(k-1)}(x)).$$

So, in other words, $A_{i+1}(x)$ involves making x applications of the A_i function on itself, starting with x. This indexed function actually defines a progression of functions, with each function growing much faster than the previous one:

- $A_0(x) = x + 1$, which is the increment-by-one function
- $A_1(x) = 2x$, which is the multiply-by-two function
- $A_2(x) = x2^x \geq 2^x$, which is the power-of-two function
- $A_3(x) \geq 2^{2^{\cdot^{\cdot^2}}}$ (with x number of 2's), which is the tower-of-twos function
- $A_4(x)$ is greater than or equal to the tower-of-tower-of-twos function
- and so on.

Ackermann Function

We then define the **Ackermann function** as

$$A(x) = A_x(2),$$

which is an incredibly fast-growing function.

- To get some perspective, note that $A(3) = 2048$ and $A(4)$ is greater than or equal to a tower of 2048 twos, which is much larger than the number of subatomic particles in the universe.

Likewise, its inverse, which is pronounced "alpha of n",

$$\alpha(n) = \min\{x: A(x) \geq n\},$$

is an incredibly slow-growing function. Even though $\alpha(n)$ is indeed growing as n goes to infinity, for all practical purposes, $\alpha(n) \leq 4$.

Thank You