Tutorial 05 - Midterm, UDFS

CS2040S Semester 1 2023/2024

Set real display name

https://pollev.com/rezwanarefin430

Midterm Review

Midterm Question 1: Sorting Problem

- Given array of **n** integers **A[1...n]**. Find place of each **A[i]** in sorted order.
- Constraints:
  - **n ≤ 200000.**
  - **-100000 ≤ A[i] ≤ 99999.**
  - All the **A[i]** are distinct.
- Example: **A = [5, -1, 4]**. Output = **[3, 1, 2]**.
  - Let’s see the examples from Midterm.

Midterm Question 1: Sorting Problem

- **O(n log n)** Solution:
  - Sort **A** and store in **B**.
  - For each **A[i]**, binary search to find where it appears in **B**.

Midterm Question 1: Sorting Problem

- **O(n log n)** Solution:
  - Sort **A** and store in **B**.
  - For each **A[i]**, binary search to find where it appears in **B**.
- **O(n)** Solution:
  - Use counting sort to sort **A**.
    - Offset each element by **100000**. So $\mathbf{0 \leq A[i] < 200000}$.
    - Offsetting doesn't affect answer.
  - Note that you can deduce the output in the last step of counting sort.

Midterm Question 1: Sorting Problem

- **O(n log n)** Solution:
  - Sort **A** and store in **B**.
  - For each **A[i]**, binary search to find where it appears in **B**.
- **O(n)** Solution:
  - Use counting sort to sort **A**.
    - Offset each element by **100000**. So **$0 \leq A[i] < 200000$**.
    - Offsetting doesn't affect answer.
  - Note that you can deduce the output in the last step of counting sort.

Midterm Question 2: Stack with `findMin()`

- Implement a Stack than supports standard `peek()`, `push()`, `pop()`, and additionally `findMin()` operation all in O(1).

Midterm Question 2: Stack with `findMin()`

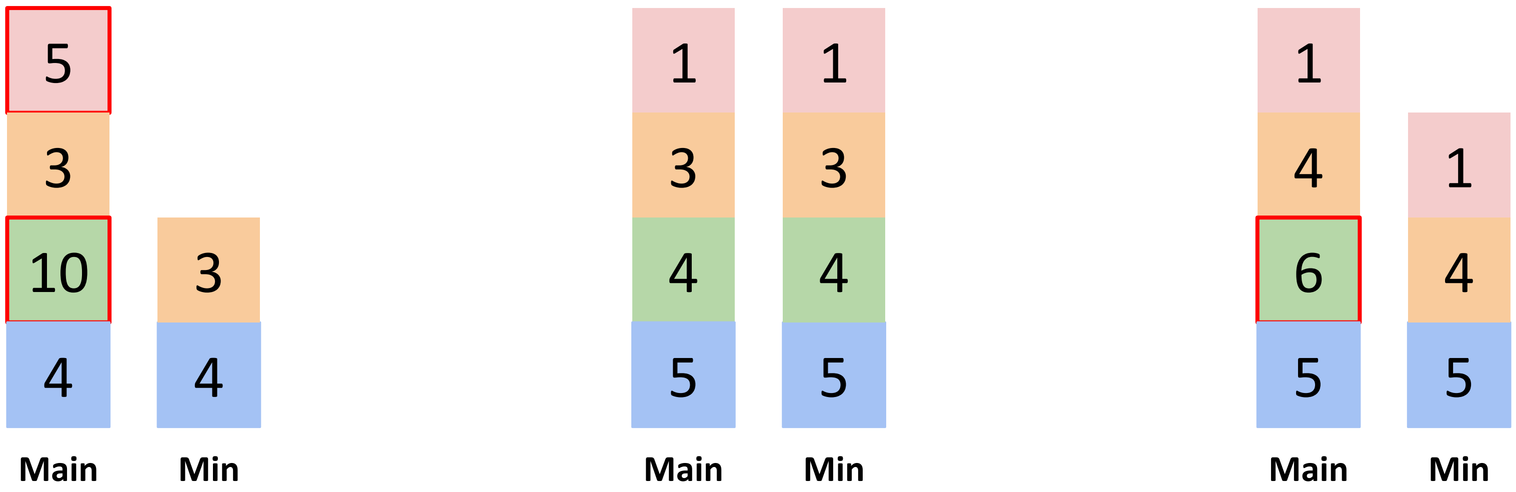
- Idea is to maintain a second stack that contains all the elements that may be returned by a future `findMin()` question.

Midterm Question 2: Stack with `findMin()`

- Idea is to maintain a second stack that contains all the elements that may be returned by a future `findMin()` question.
- The second stack should not contain any element that will never be answer to a `findMin()` query.

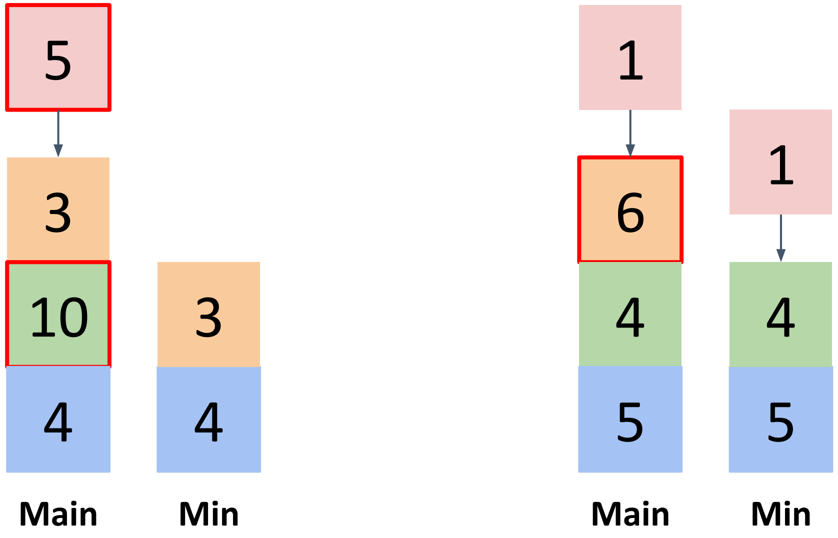
Midterm Question 2: Stack with `findMin()`

- Idea is to maintain a second stack that contains all the elements that may be returned by a future `findMin()` question.
- The second stack should not contain any element that will never be answer to a `findMin()` query.



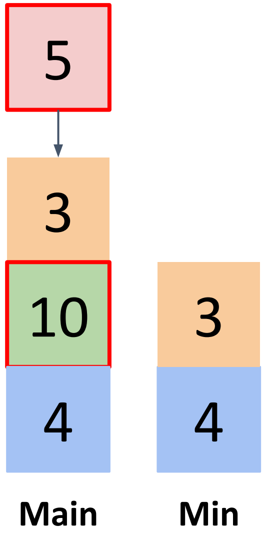
Midterm Question 2: Stack with `findMin()`

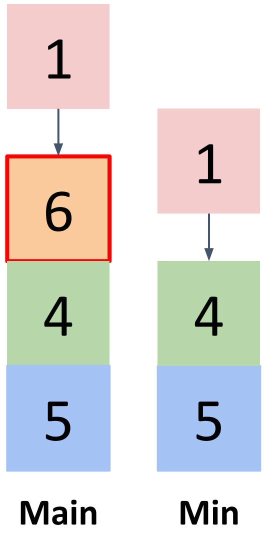
- Suppose X is being pushed in the Main stack.
- If X is bigger than top element in Min stack, it will never be answer of a findMin() query.
- Otherwise it will be, so we push it in Min stack as well.



Midterm Question 2: Stack with `findMin()`

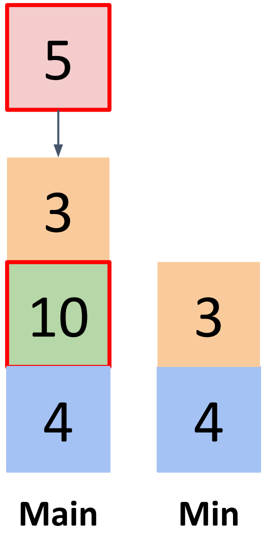
- Notice that top of the Min stack is the answer to `findMin()` now.
- What is the meaning of 2nd top element of Min stack?

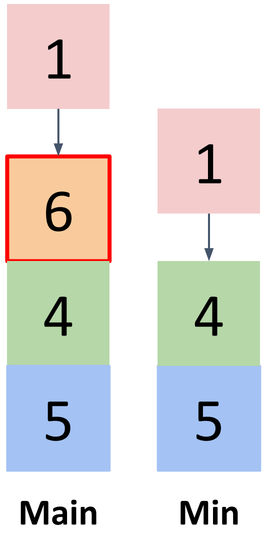




Midterm Question 2: Stack with `findMin()`

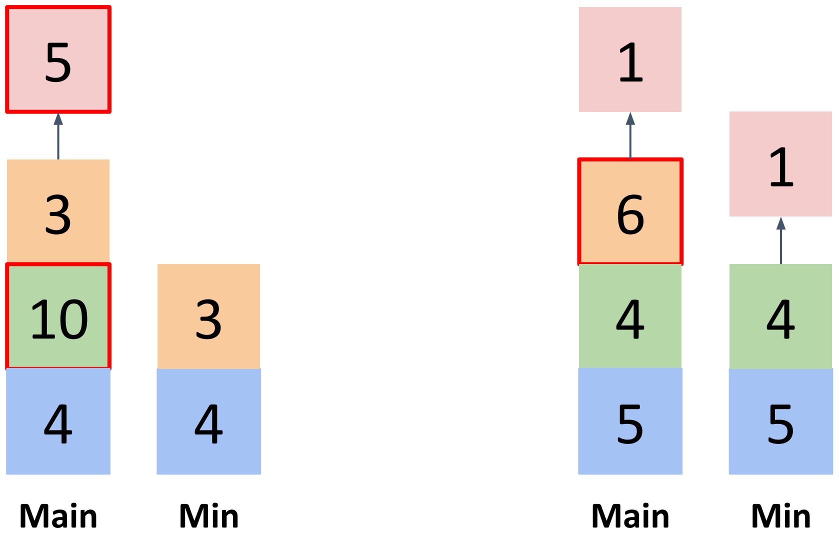
- Notice that top of the Min stack is the answer to `findMin()` now.
- What is the meaning of 2nd top element of Min stack?
    - Second minimum element.





Midterm Question 2: Stack with `findMin()`

- Therefore the popping algorithm should be as follows:
    - If top of Main stack and top of Min stack is same, pop them both.
    - Else pop only Main stack.



Midterm Question 3: Adding **n** integers

- Given **n** integers. Cost of adding **A** and **B** is **(A + B)**. Find minimum cost to sum all the integers.

Midterm Question 3: Adding **n** integers

- Given **n** integers. Cost of adding **A** and **B** is **(A + B)**. Find minimum cost to sum all the integers.
- Intuition is:
  - We will always perform exactly **n - 1** additions.

Midterm Question 3: Adding **n** integers

- Given **n** integers. Cost of adding **A** and **B** is **(A + B)**. Find minimum cost to sum all the integers.
- Intuition is:
    - We will always perform exactly **n - 1** additions.
    - We will try to minimize cost of each of the additions.
    - How to minimize cost of the current addition?

Midterm Question 3: Adding **n** integers

- Given **n** integers. Cost of adding **A** and **B** is **(A + B)**. Find minimum cost to sum all the integers.
- Intuition is:
  - We will always perform exactly **n - 1** additions.
  - We will try to minimize cost of each of the additions.
  - How to minimize cost of the current addition?
    - Take the smallest two A and B available and add them.

Midterm Question 3: Adding **n** integers

- Given **n** integers. Cost of adding **A** and **B** is **(A + B)**. Find minimum cost to sum all the integers.
- Intuition is:
  - We will always perform exactly **n - 1** additions.
  - We will try to minimize cost of each of the additions.
  - How to minimize cost of the current addition?
    - Take the smallest two A and B available and add them.
  - Techniques for proving that this eventually gives the smallest possible cost will be taught in later courses (eg. CS3230).
  - You can figure this out from doing the sample test cases.
    - Let’s see examples in the Midterm.

Midterm Question 3: Adding **n** integers

- So our algorithm is this:
  - Find and remove two smallest integers **A** and **B** from the array.
  - **Cost += A + B.**
  - Insert **A + B** back into the array.

Midterm Question 3: Adding **n** integers

- So our algorithm is this:
  - Find and remove two smallest integers **A** and **B** from the array.
  - **Cost += A + B.**
  - Insert **A + B** back into the array.
- To make this fast, use a priority queue to store the array.
- Complexity: **O(n log n)**.

Questions?

Union Find

Union Find: The Problem

- We have **n** sets initially: **{1}, {2}, {3}, …, {n}**.
- Need to support these two type of queries:
    - Merge the sets containing **u** and **v**.
    - If **u** and **v** in the same set?

Union Find: Example

- Initial Set:
  - {1}, {2}, {3}, {4}, {5}
- Union(1, 3)
  - {1, 3}, {2}, {4}, {5}
- Union(2, 5)
  - {1, 3}, {2, 5}, {4}
- IsSameSet(2, 5) = False
- IsSameSet(1, 3) = True

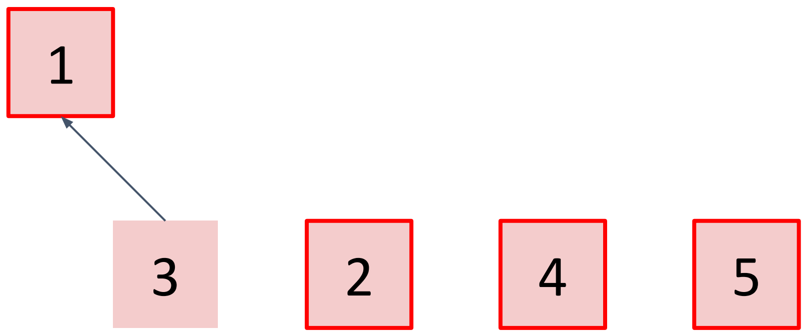
Union Find: The Data Structure

- We will represent each set as a tree.
  - Note that the structure of the tree has nothing to do with the semantics of the operations we are supporting.
- For our algorithm, we will also elect a leader in each set. The leader will be root of the tree.
- Then merging two sets is equivalent to attaching leader of one set as a child of leader of another set.

Union Find: Example

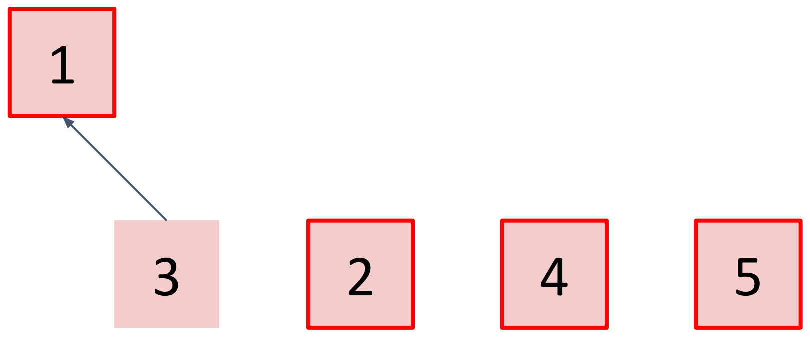
Initial State

Union(1, 3)

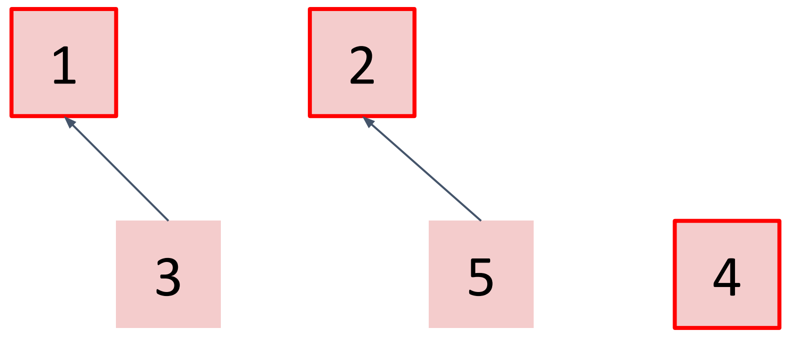


Union Find: Example

Previous State

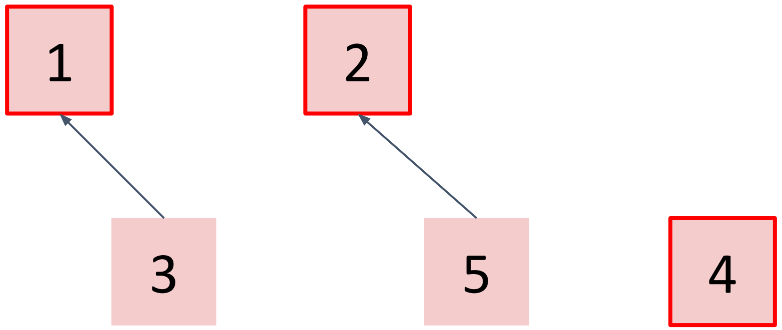


Union(2, 5)

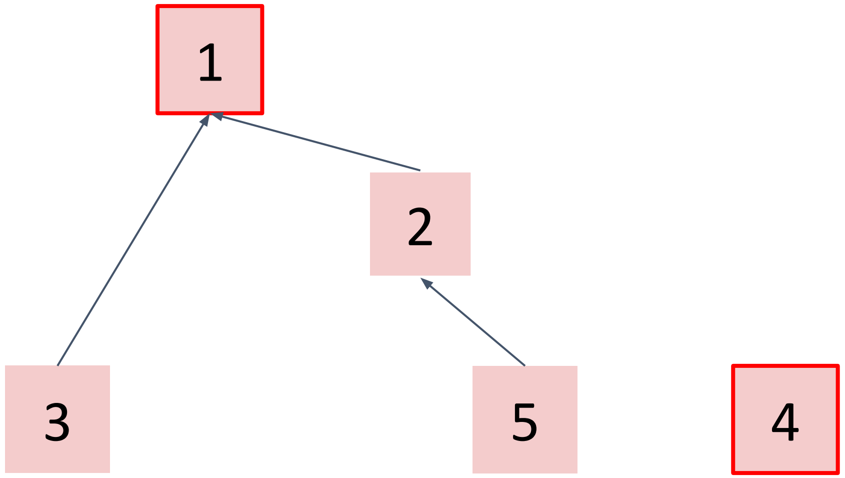


Union Find: Example

Previous State



Union(3, 5)



Union Find: `Find(u)`

- Finding the leader set is a critical operation.
- Let’s define `Find(u)` operation which returns the leader of u’s set.

Union Find: `Find(u)`

- Finding the leader set is a critical operation.
- Let's define `Find(u)` operation which returns the leader of u's set.
- Implementation:
  - Define `p[u]` = parent of u in the tree. If `p[u] = u`, then u itself is a leader.
  - Initially everyone is a leader, so `p[i] = i` for all `i`.

Union Find: `Find(u)`

- Finding the leader set is a critical operation.
- Let's define `Find(u)` operation which returns the leader of u's set.
- Implementation:
  - Define `p[u]` = parent of u in the tree. If `p[u] = u`, then u itself is a leader.
  - Initially everyone is a leader, so `p[i] = i` for all `i`.
  - When leader u becomes subordinate of `v`, we will change `p[v] = u`.
  - How to implement `Find(u)` using this `p[]` array?

Union Find: `Find(u)` Implementation

- Keep following p[u] pointers until we find a leader.

```
public static int Find(int u) {
	while (p[u] != u) u = p[u];
	return u;
}
```

Union Find: `IsSameSet(u,v)` Implementation

- How to check if two element are in the same set?

Union Find: `IsSameSet(u,v)` Implementation

- How to check if two element are in the same set?
- They must have the same leader.

```
public static boolean IsSameSet(int u, int v) {
	return Find(u) == Find(v);
}
```

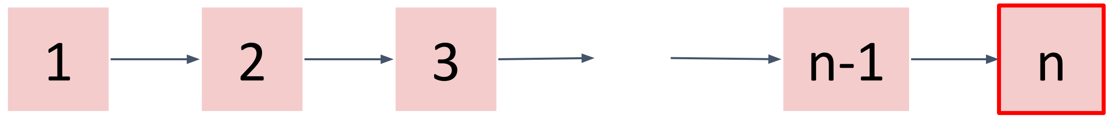
Union Find: `Union(u,v)` Implementation

- Find leaders of set containing u and v.
- If they are the same, then nothing to do.
- Otherwise, attach one of them as child of another.

```
public static void Union(int u, int v) {
	u = Find(u);
	v = Find(v);
	if (u == v) return;
	p[u] = v;
}
```

Union Find: Complexity

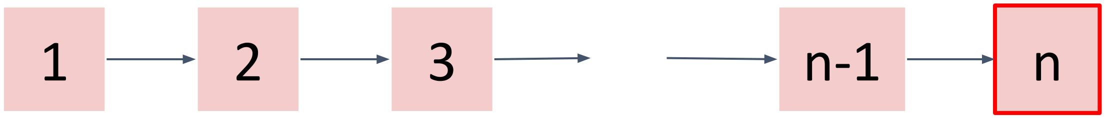
- Current implementation complexity is O(n).
- Suppose we do Union(1, 2), Union(2, 3), Union(3, 4), …, Union(n-1, n).
- Then we will have the following tree:



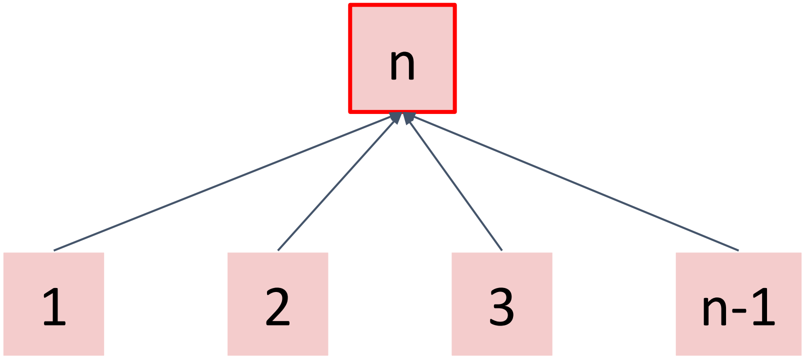
- Now if we do IsSameSet(1, 2), both the calls Find(1) and Find(2) will take O(n) to find the leader.

Union Find: Path Compression

- Note that we don't care about the structure of the tree, as long as all elements in same set are in the same tree.
- So, whenever we call Find(u), we will compress the path from u to root.



- Find(1) call should return n and change the structure as follows:



Union Find: Path Compression Implementation

```
public static int Find(int u) {
	if (p[u] == u) {
		return u;
	} else {
		p[u] = Find(p[u]);
		return p[u];
	}
}
```

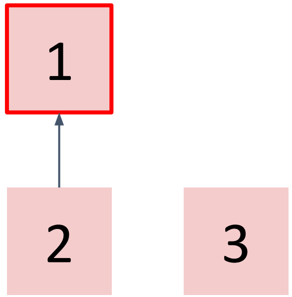
- Complexity is still worst case **O(n)**.
- But it can be proven than over **n** calls to Find, the total complexity cannot exceed **O(n log n)**.
- So we say that the complexity is amortized **O(log n)**.

Union Find: Union by Rank

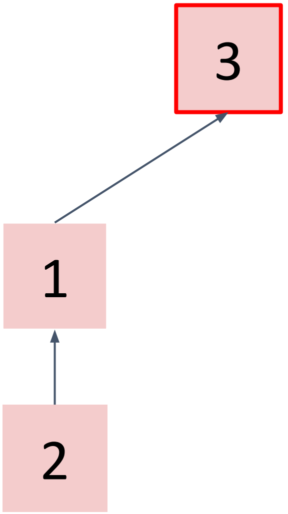
- (Assume we don’t have path compression)
- In Union(u, v), we blindly attached u’s leader as child of v’s leader.
- If u’s tree has height 5, and v’s tree has height 4:
  - Attaching u’s tree to v’s tree results in a tree of height 6.
  - Attaching v’s tree to u’s tree results in tree of height 5.
- Attaching lower height tree to higher height tree results in a lower height.

Union Find: Union by Rank

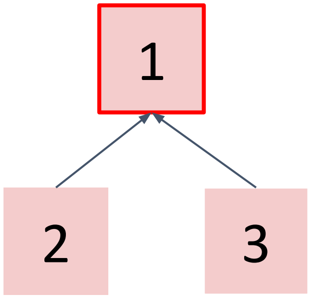
Union(2, 3)



Attach 1 to 3



Attach 1 to 3



Union Find: Union by Rank Implementation

- Define rank[u] = height of subtree rooted at u.
  - Initially everyone has rank 0.
- When merging two trees, only the leader’s rank may change. Update accordingly.

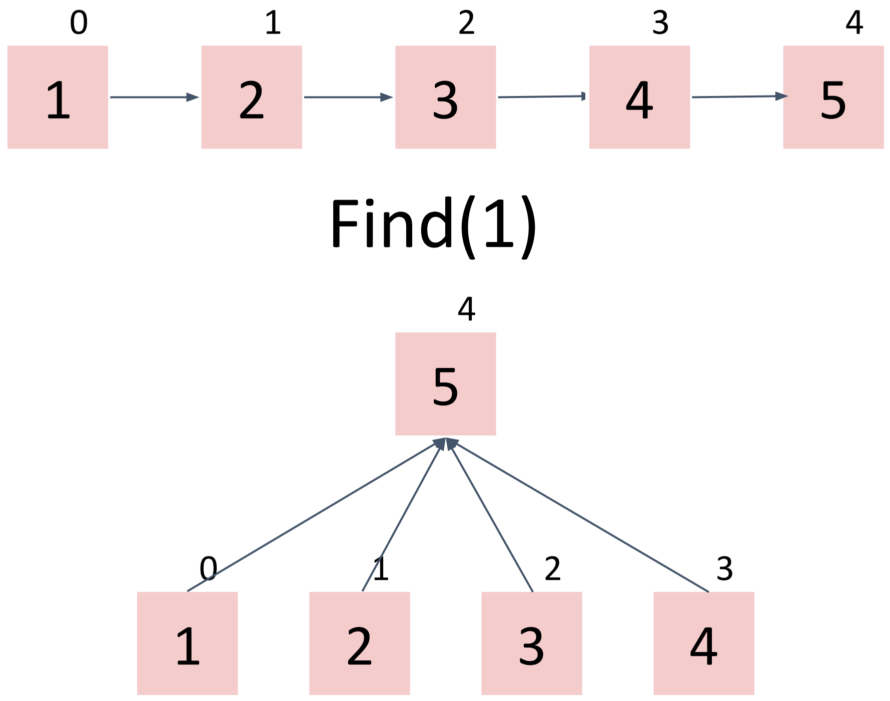
```
public static void Union(int u, int v) {
    u = Find(u);
    v = Find(v);
    if (u == v) return;
    if (rank[u] > rank[v]) swap(u, v); // ensure rank[u] <= rank[u]
    p[u] = v; // v is the new leader
    if (rank[u] == rank[v]) ++rank[v];  // rank increases only if before merging both tree had same rank
}
```

Union Find: Union by Rank Complexity

- With only Union by Rank, every operation is *worst case* O(log n).
- If Union by Rank and Path Compression both are used, then every operation is amortized **O(α(n))** where **α** is the inverse ackermann function.
  - Ackermann function is VERY fast growing function.
  - It inverse for any realistic value of **n** is **< 4**.
- Note that here amortized **O(1)** is special. The total cost over **n** operation is **O(n).** Normal amortized complexity would allow one single operation to be **O(n)**. But here Union by Rank guarantees that one single operation doesn't exceed **O(log n)**.

Union Find: Union by Rank + Path Compression

- After Path Compression, the rank values may *not* correspond to height of the current structure. They will represent height of the tree assuming the tree was never compressed.



Tutorial Questions

Question 3: `NumDisjointSets()`

- How to quickly query number of disjoint sets?

Question 3: NumDisjointSets()

- How to quickly query number of disjoint sets?
- Keep a counter. Initially there are **n** disjoint sets.
- Whenever two sets are merged, decrease the counter by 1.
- NumDisjoinSets just returns that counter in O(1).

Question 4: `SizeOfSet(u)`

- How to augment the data structure to query size of set containing u efficiently?

Question 4: `SizeOfSet(u)`

- How to augment the data structure to query size of set containing u efficiently?
- Similar to rank, define `size[u]` = size of set containing u.
    - This value will only be correct for leaders.
- Initially everyone is a leader, and `size[i] = 1` for all `i`.
- When attaching leader u to leader v, do `size[v] += size[u]`.
- `SizeOfSet(u)` should return `size[Find(u)]`.

Break

Attendance

Questions

https://visualgo.net/training?diff=Medium&n=5&tl=5&module=ufds

PS4 Discussion

PS4A: /swaptosort

- Given a reverse sorted array **A.**
- You have a list of **(a, b)** pairs.
- Pair **(a, b)** means that you are allowed to **swap(A[a], A[b])**.
- Can you sort the array using only allowed operations?

PS4A: /swaptosort

- Think like the midterm question.
- Where should **A[i]** go in the final array?

PS4A: /swaptosort

- Think like the midterm question.
- Where should **A[i]** go in the final array?
    - **n - i + 1**.
- If you need to swap indices a and b, there must be a sequence of valid swaps: **(a, x), (x, y), (y, z), …., (*, w), (w, b)**.
- Which data structure we've learnt let us query if a and b are reachable like this?

PS4B: /kaploeb

- A simulation question.
- Use a hash table to keep track of timings, since the ids are too big.
- Parse the floating point number carefully.

Hand-On

Hands-On: /speedrun

- Given some tasks, **i-**th of them needs the time period **[l_i, r_i]**.
- Two tasks cannot be active at the same time.
  - Cannot do both **[1, 3]** and **[2, 5]**.
  - Can do both **[1, 3]** and **[3, 5]**.
- Can you do at least **G** tasks?

Hands-On: /speedrun

- Should you choose the first task you can start?
  - No! What if it takes too long?
  - Example: [1, 24000], [2, 2], [3, 3], [4, 4], ….
- But you can prove that choosing the task that ends first always lets you finish maximum number of tasks.
  - If choosing the task **T** that ends first was not optimal, then there is another more optimal solution
  - But in that optimal solution, you can ignore the first task and do **T** instead.
  - So choosing **T** first is also as optimal as the other solution.

Hands-On: /annoyedcoworkers

- You have **c** coworkers.
- Coworker **i** is initially has $\mathbf{a_i}$ annoyance level.
- If you ask coworker **i** a question, the annoyance level increases by $\mathbf{d_i}$.
- You need to ask **h** questions.
- Minimize the maximum annoyance level of any coworker after asking **h** questions.

Hands-On: /annoyedcoworkers

- Should you ask the least annoyed person first?

Hands-On: /annoyedcoworkers

- Should you ask the least annoyed person first?
  - No! What if his temper is infinite?
  - Remember you are trying to minimize the resulting max annoyance level.

Hands-On: /annoyedcoworkers

- Should you ask the least annoyed person first?
  - No! What if his temper is infinite?
  - Remember you are trying to minimize the resulting max annoyance level.
- Choose the person whose resulting annoyance level will be the smallest.

Hands-On: /annoyedcoworkers

- Should you ask the least annoyed person first?
  - No! What if his temper is infinite?
  - Remember you are trying to minimize the resulting max annoyance level.
- Choose the person whose resulting annoyance level will be the smallest.
- Use a priority queue to store the persons, sorted by their resulting annoyance level if you were to ask them a question.

Thank You!

Anonymous Feedback:
https://forms.gle/MkETeXdUT53Vhh896