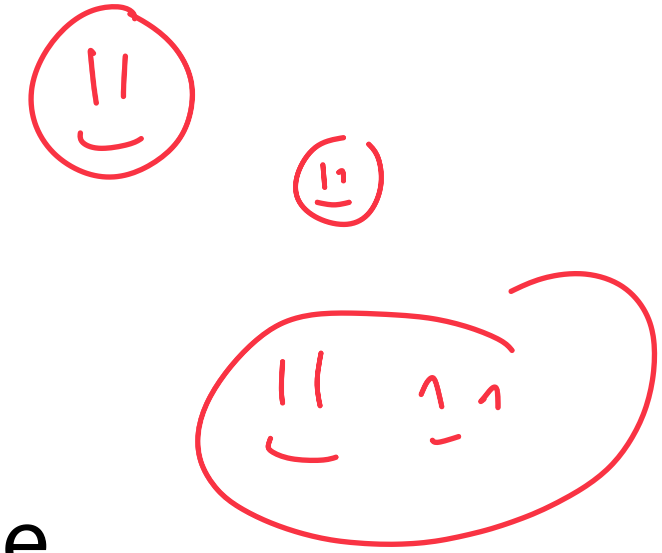
Data Structures

Extra Credit Project and Disjoint Sets

CS 225

Brad Solomon & G Carl Evans

October 16, 2023



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

November 1

Discuss extra credit project

Finish analyzing efficiency of minHeap

Introduce disjoint sets

Big Picture: Extra credit project

Do something that is of personal interest to you!

Want to do undergrad research? Find a foundational algorithm!

↳ Look at website of faculty!

↳ Get project ideas!

Want to go off into industry? Demonstrate knowledge with code!

↳ Do something cool!

Want extra credit points? Use one of the suggested algorithms!

↳ The structure of this project requires work

ECP Proposal

Approved By November 1

You are 'writing' your own assignment skeleton

1. Function I/O (in written proposal)
   ↳ Make a header file w/ comments

2. Tests (in Github repo)
   ↳ What correct alg does (Manually computed!)

3. Datasets (in Github repo)
   ↳ ≤ 4 datasets of different sizes
   ↳ Order of magnitude sizes

No details on code needed!

ECP Proposal

Approved By November 1

You dont need to know how to implement to propose a structure!

↳ dont start coding until approved project!

ECP Mid-Project Check-in

Meet by November 20

Meet with your mentor to confirm your algorithm works!

↳ Come to meeting once your algorithm is complete

↳ Benchmark after project is correct!

ECP Final Deliverables

Due December 6

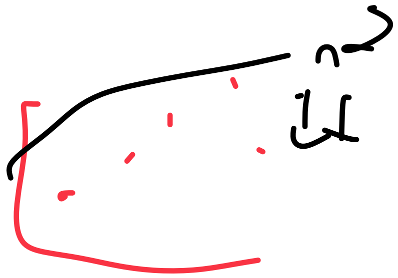
Prove your algorithm is correct and estimate runtime

1. Submit code base (GitHub repo)

2. Write a report that describes proof of correctness and efficiency

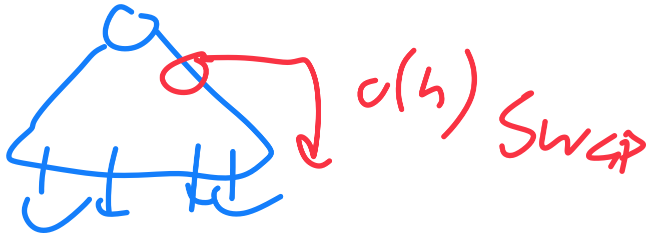
3. Present your work! Highlight what you did!

Benchmark datasets



Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is O(n)



Proof Strategy:

1. Call heapifyDown() on every non-leaf node

2. Every node we heapifyDown() has its height as worst case work.

Summing the total heights of every node is our worst case time!

Proving buildHeap Running Time

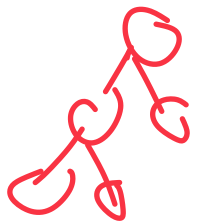
Theorem: The running time of buildHeap on array of size **n** is O(n)

$$S(h) = \overset{2}{\cancel{3}}{}^{h+1} - 2 - h$$

How can we relate **h** and **n**?

$h = O(\log n)$

$h \leq \log(n)$



complete

$\downarrow$

$\log n$

How can we estimate running time?

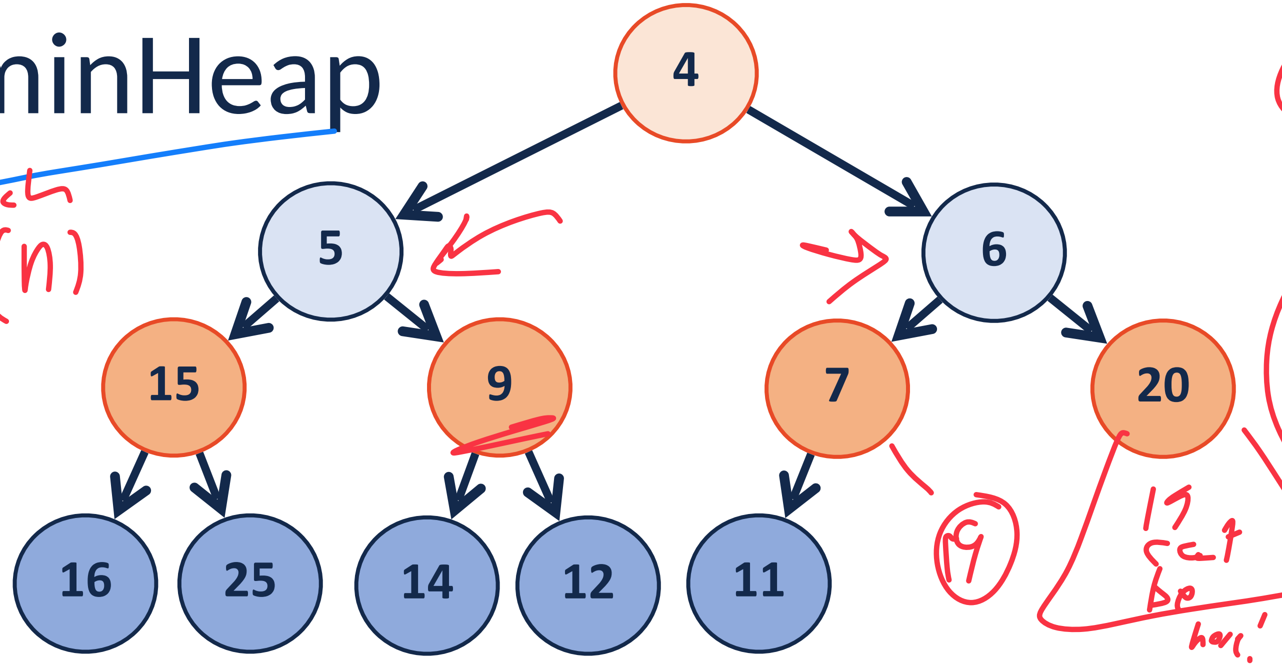
$$2^{\log(n)+1} - 2 - \log(n)$$

$$2 \cdot 2^{\log_2(n)} - 2 - \log(n)$$

$$2 \cdot n - \log(n) - 2 \approx O(n)$$

minHeap

search O(n)



1. **Construction** → O(n)
2. **Insert** → O(log n)
3. **RemoveMin** → O(log n)

Just an array in storage

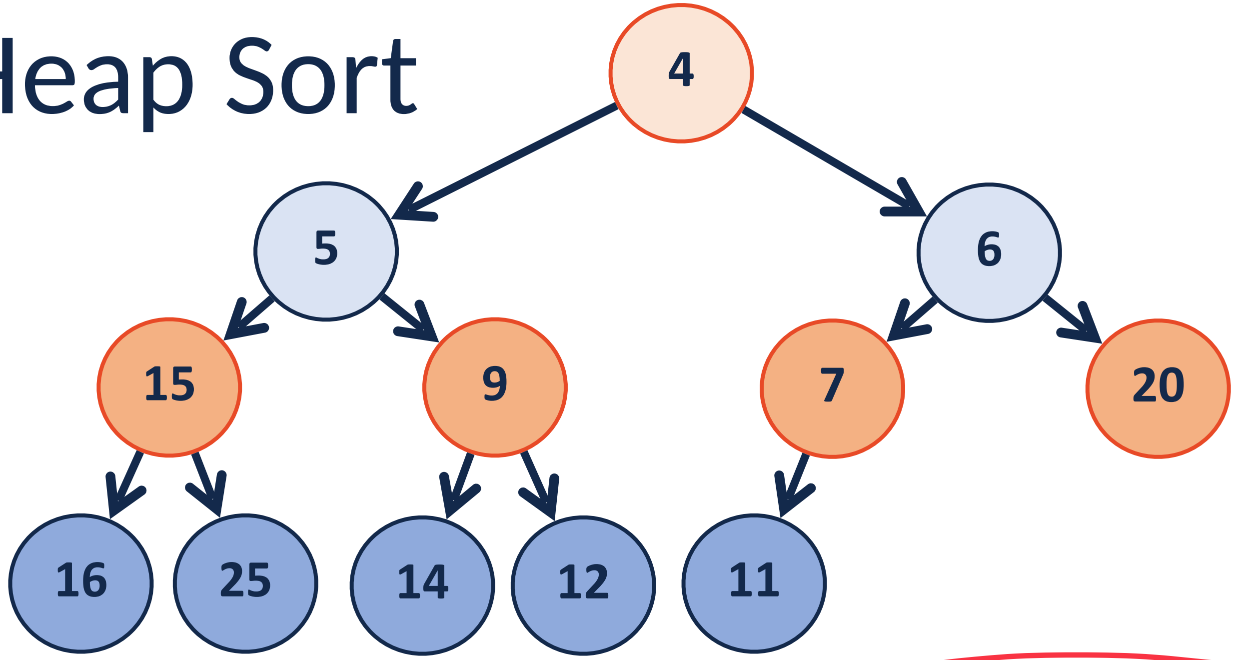
	4	5	6	15	9	7	20	16	25	14	12	11	19		

minHeap is a good example of tradeoffs:

Nearly optimal* for every function ⟷ Cost of access!

Not suitable for random access or find

Heap Sort



1. Construction → $O(n)$

2. Call removeMin() n times

   → $n \cdot O[\log n]$

3. Reverse list is our sorted list

1) swap w/ last
size--;

	4	5	6	15	9	7	20	16	25	14	12	11			

9 7 6 5 4

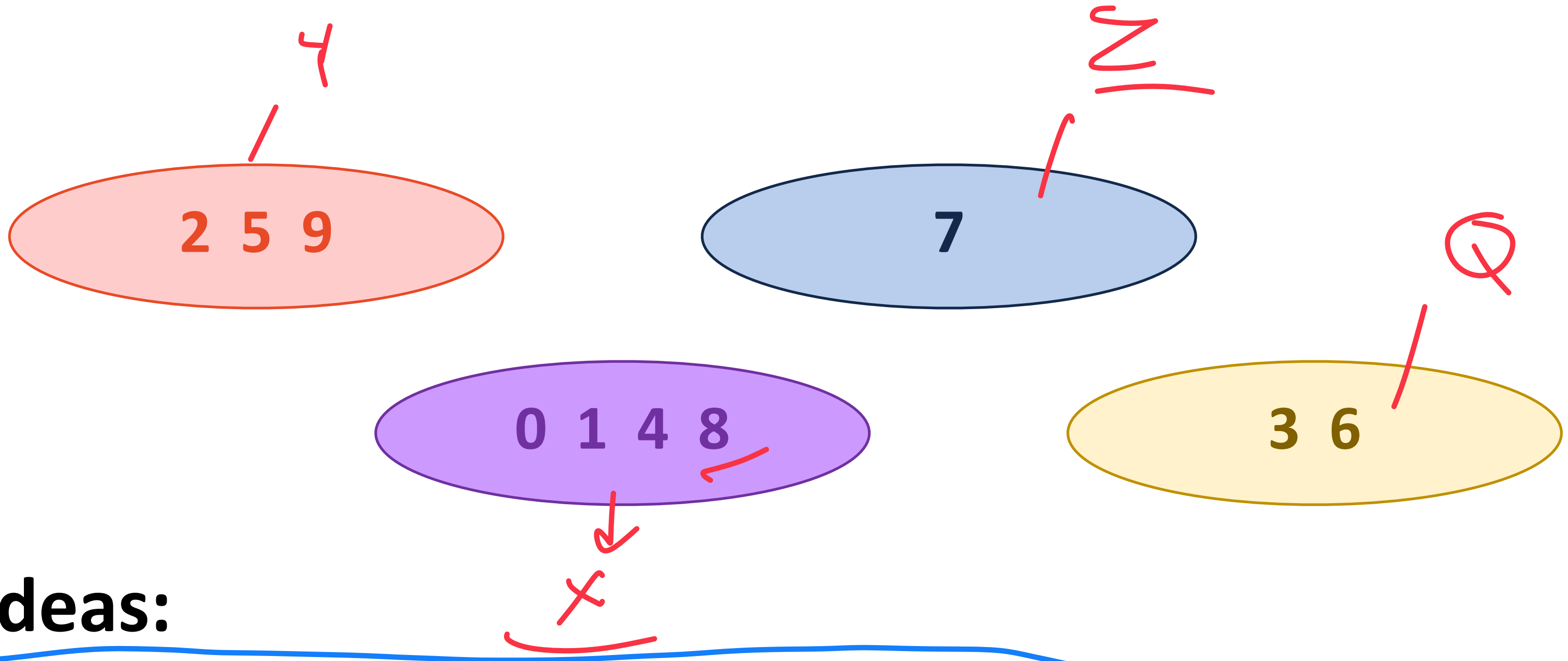
Running time? $O(n \log n)$

Disjoint Sets

Sort of a dictionary

keys = my #s

Value = their set



7: Z

0: X

1: X

8: X

4: X

Key Ideas:

- Each element exists in exactly one set.
- Every item in each set has the same representation
  - In other words: find(4) == find(8) == find(0) …
- Each set has a different representation
  - In other words: find(7) != find(4)

Disjoint Sets

pick a representative in each set

Dont care how

Each set is represented by a **canonical element** (internally defined)

2 5 9

7

0 1 4 8

3 6

0:1 4:1
1:1 8:1

telling me what set I am in

Operation:

```
find(4) == find(8)
```

1 == 1 ✓

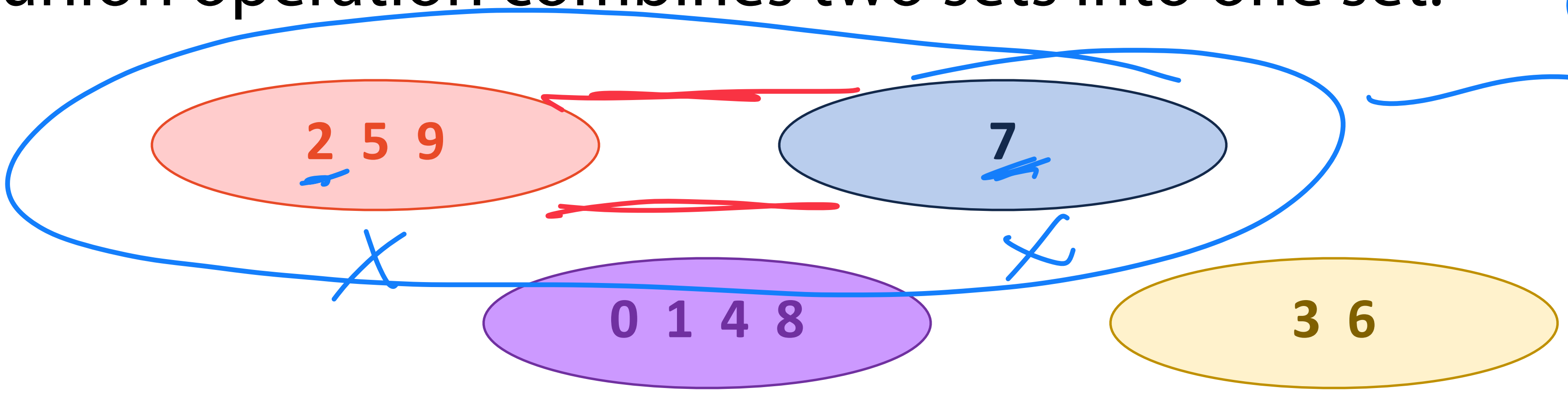
find(7) == find(6)

7 ✗ 6

Disjoint Sets

The union operation combines two sets into one set.

Only Set 2579 or union



Operation:

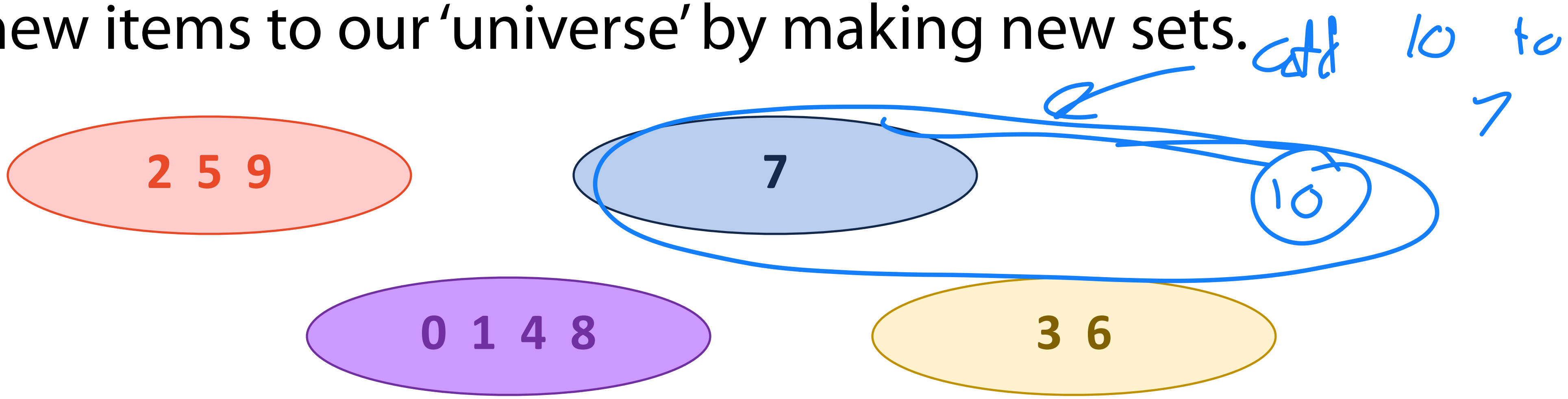
```
if find(2) != find(7){
   union( find(2), find(7) );
}
```

2 5 7 9

or

Disjoint Sets

We add new items to our 'universe' by making new sets.



Operation:

```
makeSet(10);
```

union (7, 10)

Disjoint Sets ADT

minimal functions we expect

Constructor

insert into sets

makeSet ( )

union ( )

makeSet

Find

Union

Disjoint Sets

How might we implement a disjoint set?

Dictionary[key] = set representation

0 1 4 8 / 7

B Tree (or any tree)

Dictionaries!

Tree (BST)

RB tree

Vector!

We use small #s here for example

Make another dictionary

"MyBook" : 0

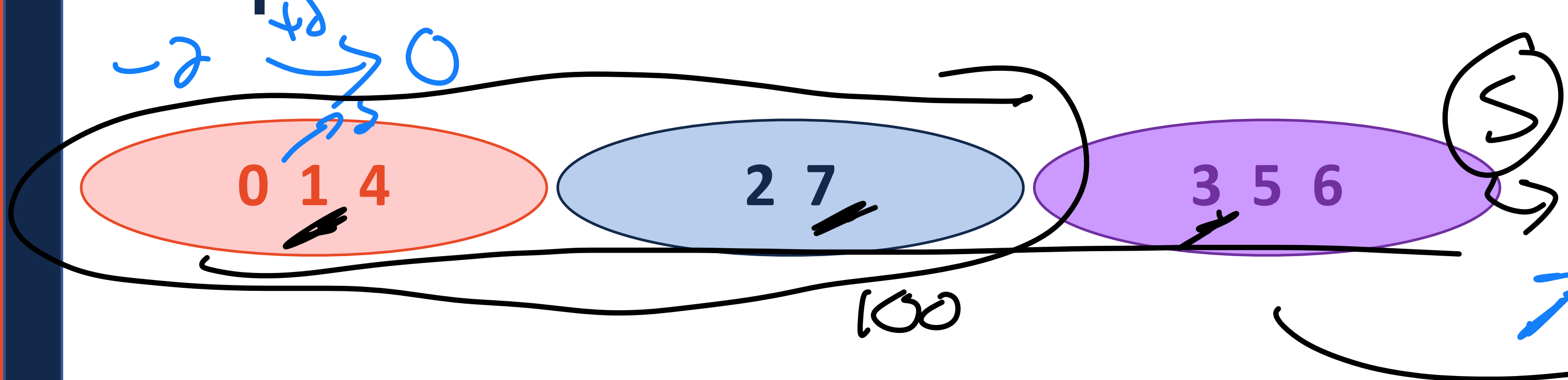
"YourBook" : 1

2 : 9

Implementation #1

Allocate memory for key max

String key as index



S

0	1	2	3	4	5	6	7
1	2	~~1~~ 1	3	1	3	3	~~7~~ 1

100

Makeset → array

Find(k): S[k] ← that's my set $O(1)$

How to make!

1) Allocate array of length = largest key
↳ 7 so array = size 7

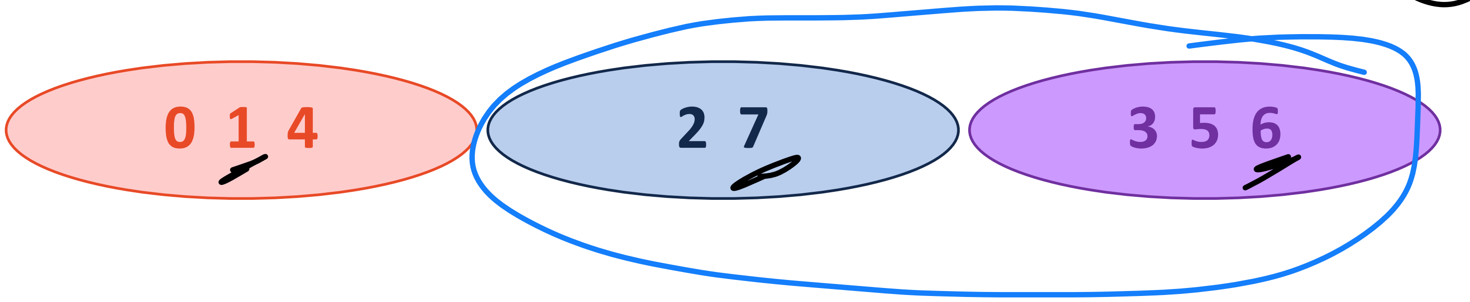
2) Each item stored at index
↳ what set 4 is in S[4]

(1 7)

Union(k_1, k_2): Replace one sets canonical rep w/ other

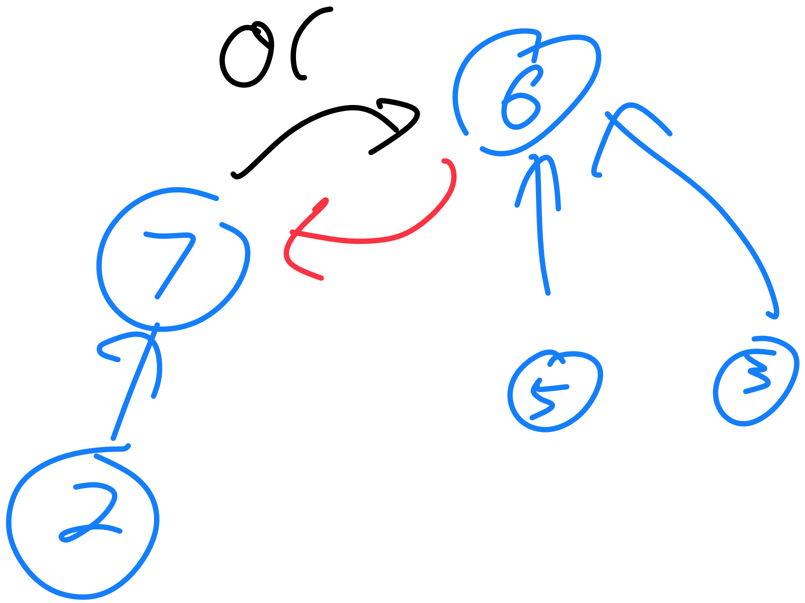
↳ $O(n)$

Implementation #2 → Same idea but store canonical element as -1



0	1	2	3	4	5	6	7
1	-1	7	6	1	6	-1	-1

either 7 -1
-1 6

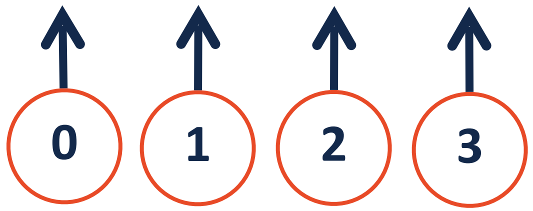


Find(k):

6, 7

Union(k_1, k_2):

UpTrees



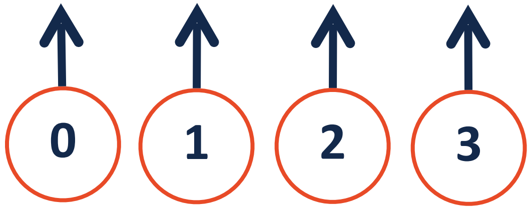
0	1	2	3

0	1	2	3

0	1	2	3

0	1	2	3

UpTrees



0	1	2	3

0	1	2	3

0	1	2	3

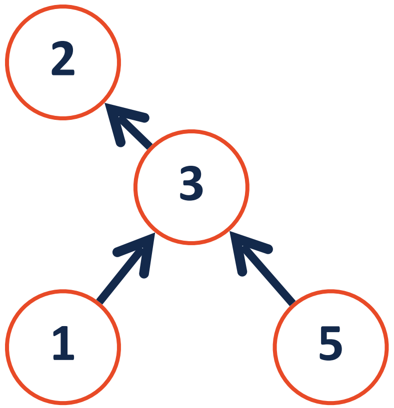
0	1	2	3

Disjoint Sets Representation

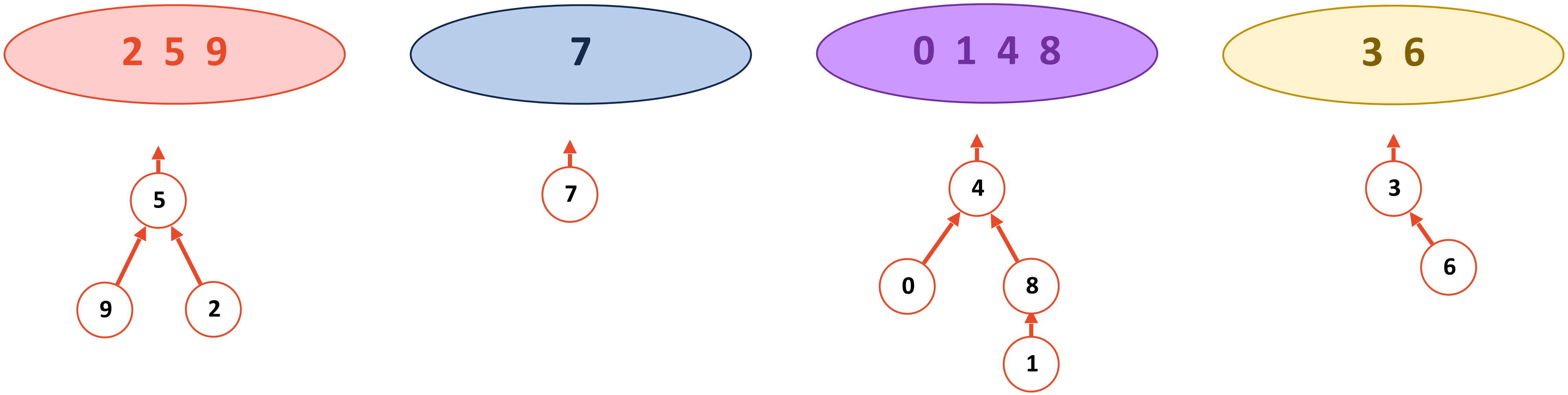
We can represent a disjoint set as an array where the key is the index

The values inside the array stores our sets as a pseudo-tree (UpTree)

- The value **-1** is our representative element (the root)
- All other set members store the index to a parent of the UpTree



Disjoint Sets



0	1	2	3	4	5	6	7	8	9

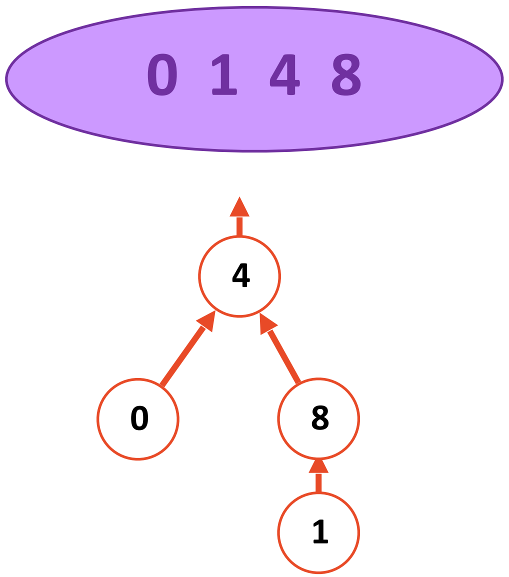
Disjoint Sets Find

Find(1)

```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else { return find( s[i] ); }
}
```

Running time?

What is ideal UpTree?

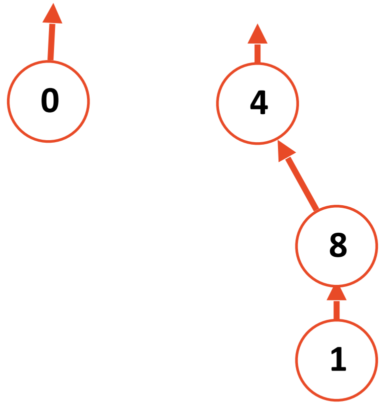


0	1	2	3	4	5	6	7	8	9
4	8			-1				4	

Disjoint Sets Union

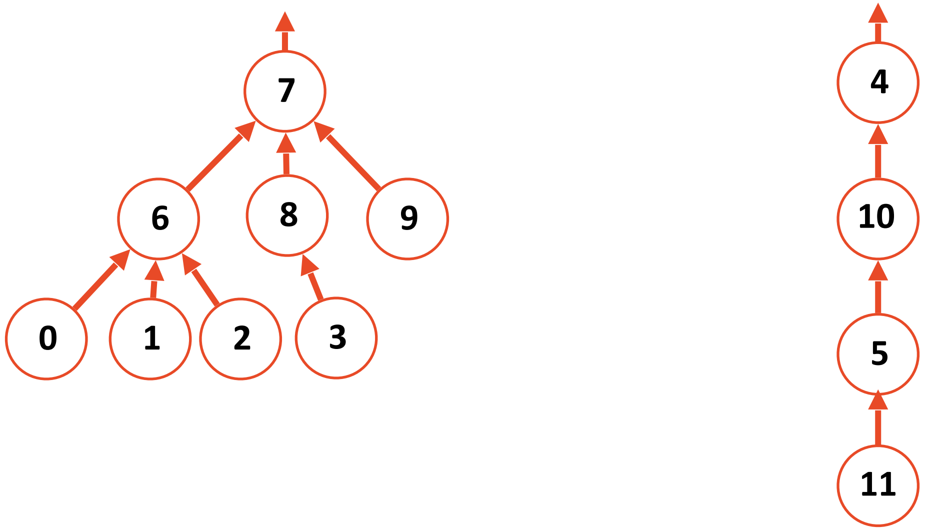
Union(0, 4)

```
int DisjointSets::union(int r1, int r2) {
}
```



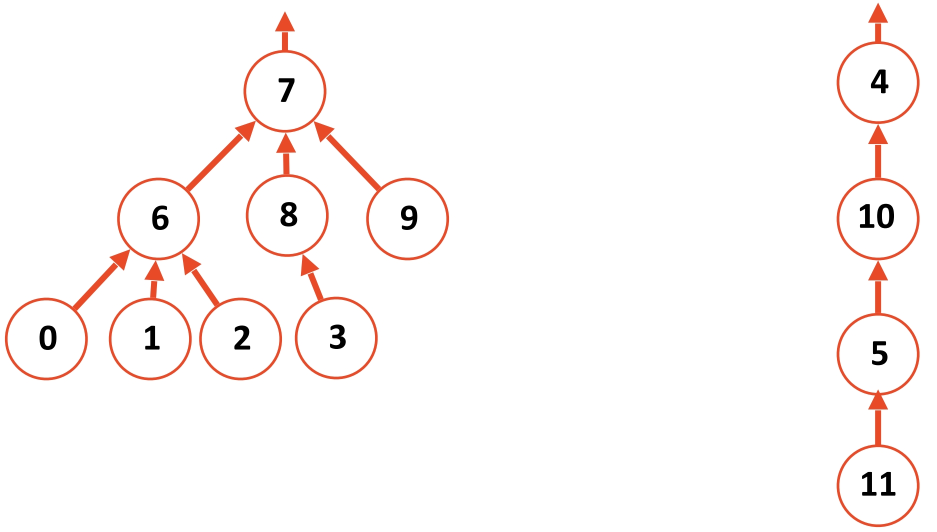
0	1	2	3	4	5	6	7	8	9
-1	8			-1				4	

Disjoint Sets – Union



0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5

Disjoint Sets – Smart Union

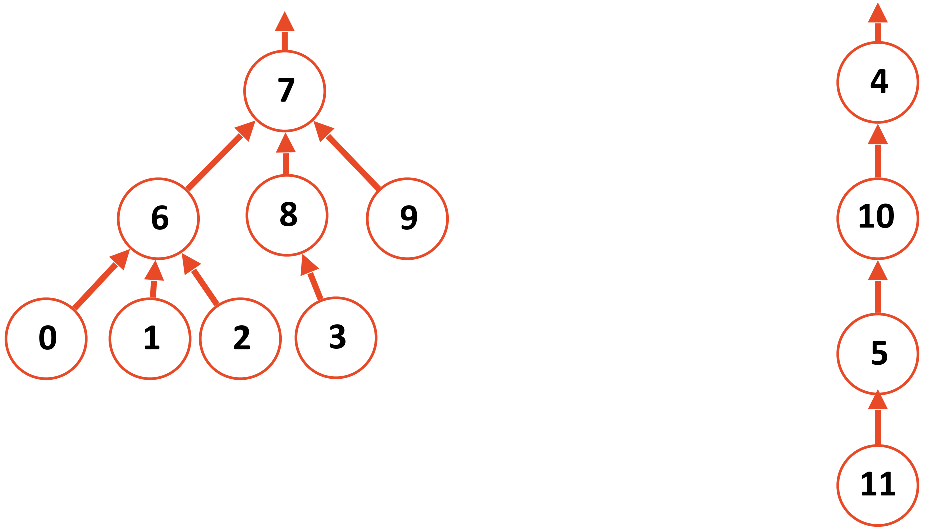


Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Disjoint Sets – Smart Union

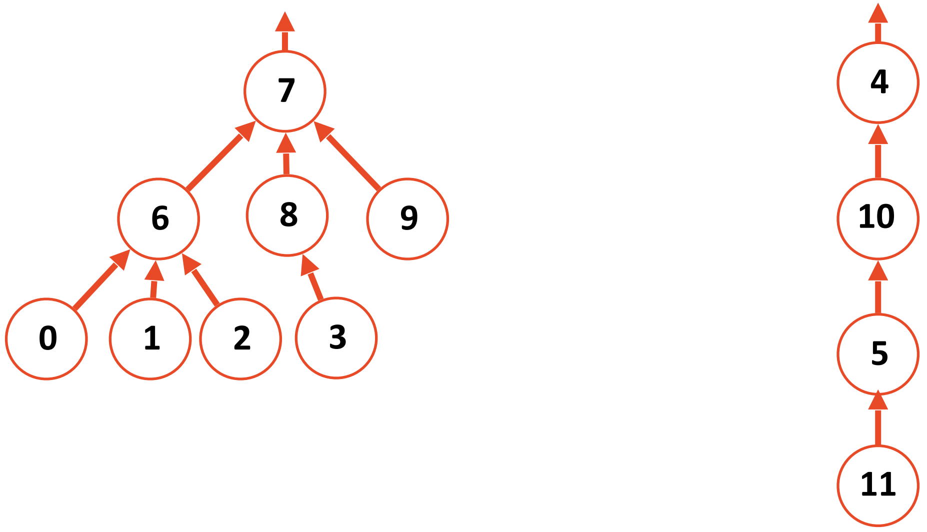


Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

Idea: *Minimize the number of nodes that increase in height*

Disjoint Sets – Smart Union



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

***Idea**: Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is: _______________.