Set Data Structures

Let's "set" our eyes on a solution

Importance of Sets

Organizing Information

Importance of Sets

- A *set* is an unordered collection of "objects"
- Sets are used in computer science is a wide variety of ways

Importance of Sets

- Depending on the attributes of the set, and how we use it, there are different approaches
- Programmers must choose the best model given how it will be used

Set Review: Membership

- Set notation uses a special symbols to denote if an object is a member of a set
- Below, the set V contains vegetables

```
potato ∈ V
bacon ∉ V
```


Subsets

- Set A is considered a subset of set B if all the members of A are also members of B
- The subset operator is similar looking to the member operator

```
{ 1, 4 } ⊆ { 1, 3, 4, 5 }
```

Set Review: Union

- A union of two sets combines all members of each set into a new one
- So, the result is two merged sets

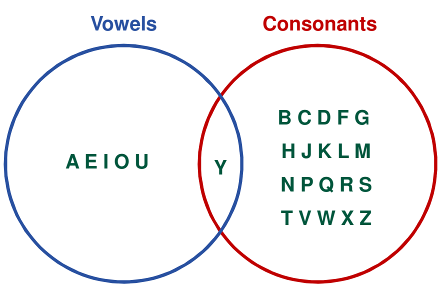
$$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$$

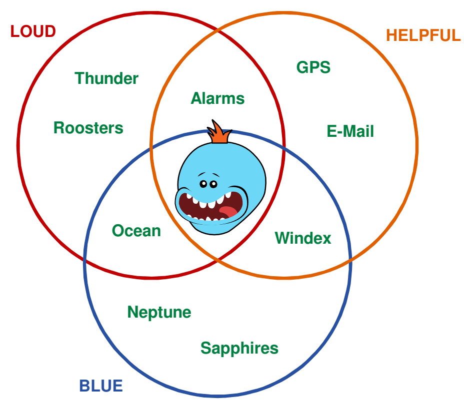
Set Review: Intersection

- The intersection of two sets contains only those elements that are found in <u>both</u> sets
- So, the result is where the two sets overlap

$$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$$

Example: Letters in English





Set Review: Difference

- *Difference* (aka exclusion) removes all items found in set from another
- Typically, it is written either A – B or A \ B

$$A - B = \{ x \mid x \in A \text{ and } x \notin B \}$$

Set Review: Complement

- The *complement* of a set A, is all elements in the Universe, <u>not</u> in A
- Typically written as A' or A^c
- Alternatively written using an overbar

$$A' = \{ x \mid x \notin A \}$$

Multiset

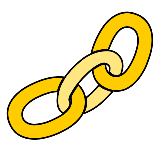
- A *multiset* is closely related to a set, but permits duplicate elements (as does a tuple)
- The Bag ADT, from the beginning of the semester, supports a basic multiset

Sets using Linked Lists

Probably the most obvious way

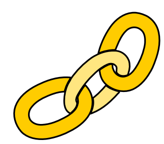
Sets using Linked Lists

- One obvious approach, to store a set structure, is to use a linked list
- We can either attempt to maintain a sorted list or unsorted list



Sets using Linked Lists

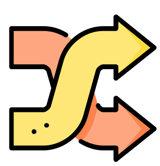
- Remember: there is <u>no</u> ordering in a set, *so we are free to move elements around as we wish*
- We can take advantage of this to reduce time complexity



Unsorted List

- If we are maintaining an <u>unsorted</u> <u>multiset</u>, then individual elements can be appended with O(1)
- In fact entire lists can be appended (a union of two sets) at a cost of O(n)
- … or just linked together at a cost of O(1)

Unsorted List without Duplicates

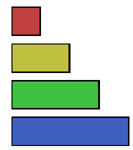


- However, <u>if</u> duplicates must be avoided, then issues will arise
- To append a single element, the entire list must be searched: O(n)

Unsorted List without Duplicates

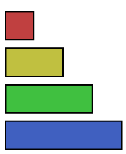
- Likewise, a union will have a time complexity of $O(n^2)$
- Conceptually, this would be the same as merging two unsorted arrays
- So, this might not be the way to go…

Sorted List



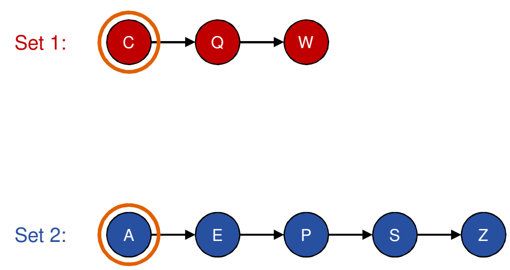
- Just like in the Merge Sort, if two lists are sorted, the merging algorithm takes O(n)
- Well, in this case, the "merge" is a set union

Sorted List

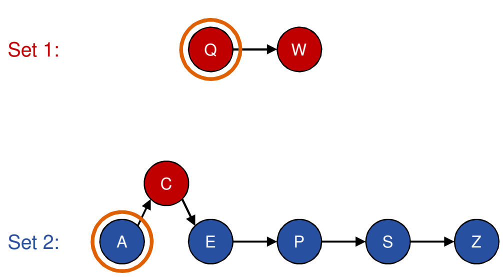


- The algorithm is simple:
  - advance through both lists and insert items where needed
  - only links need to be updated – just like adding an item into a linked list

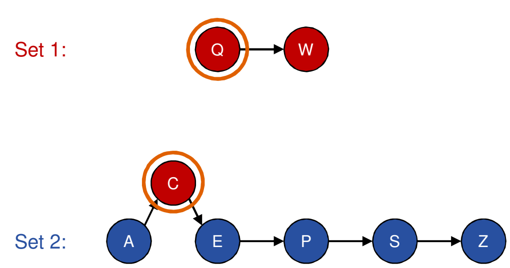
Union Set 1 into Set 2



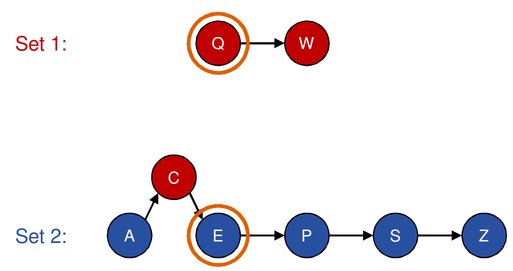
Union Set 1 into Set 2



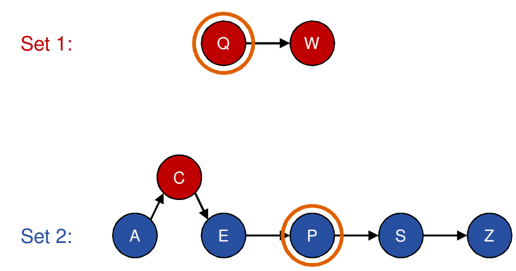
Union Set 1 into Set 2



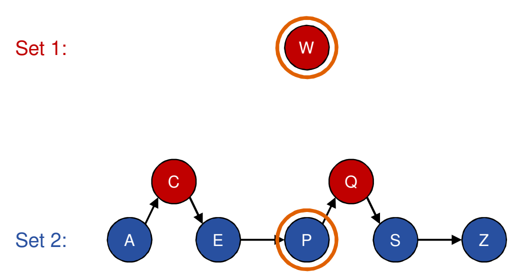
Union Set 1 into Set 2



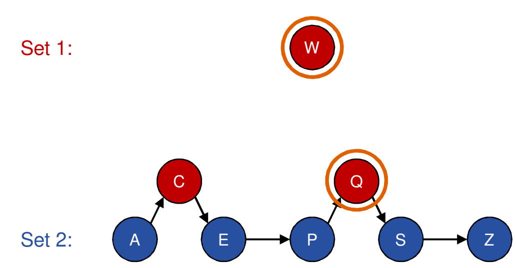
Union Set 1 into Set 2



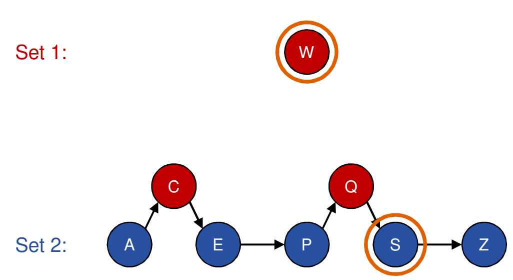
Union Set 1 into Set 2



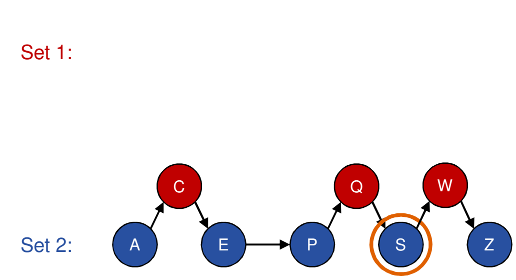
Union Set 1 into Set 2



Union Set 1 into Set 2



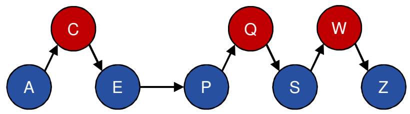
Union Set 1 into Set 2



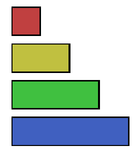
Union Set 1 into Set 2

Set 1:

Set 2:



Difference and Intersection



- Both Intersection and Difference can be performed using the same type of algorithm
- So, the sorted list approach does have some benefits

Sets using Arrays

Hidden math = easy code

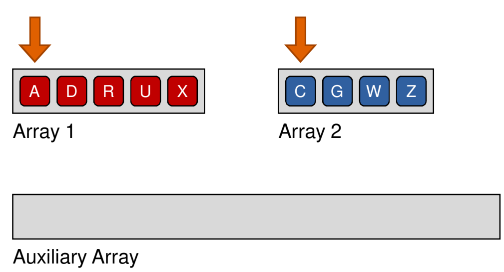
Sets using Arrays

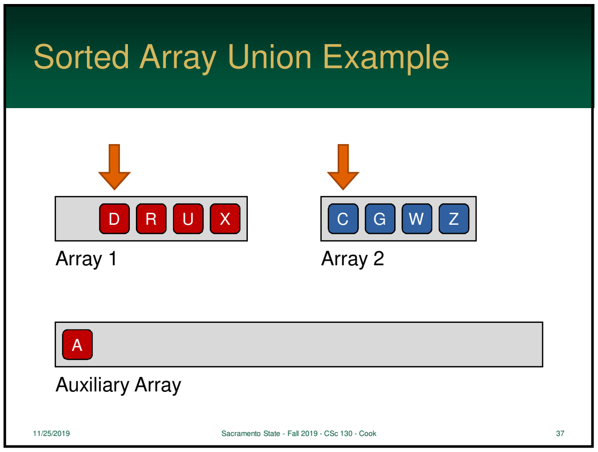
- Arrays can also be used to store sets
- Information can be looked up much faster if the array is sorted – i.e. the binary search
- So, for a lookup, arrays are far better

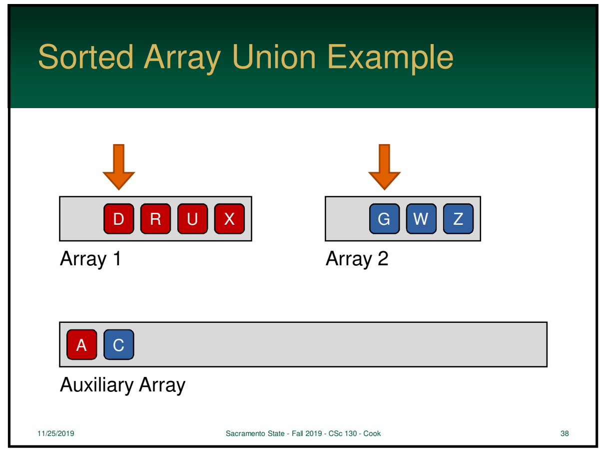
Sets using Arrays

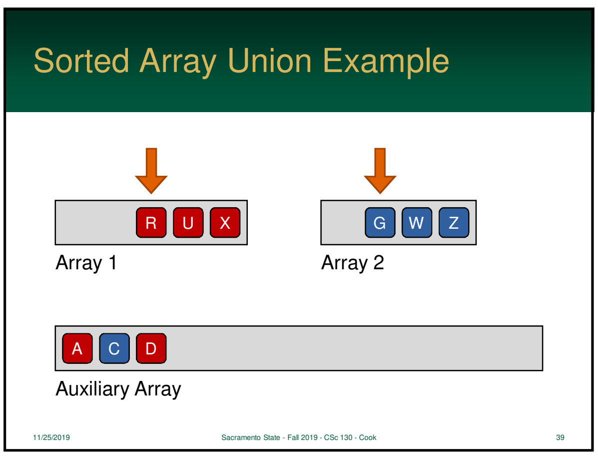
- Like Linked Lists, it is far better to maintain a sorted list
- This is also the only logical solution for arrays, since an append takes O(n)
- Might as well insert it into the sorted position!

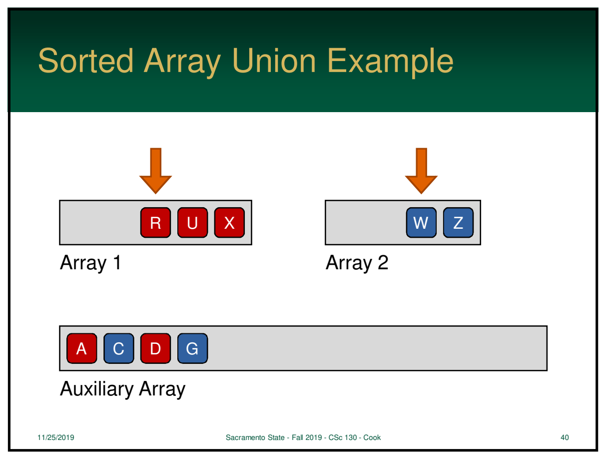
Sorted Array Union Example

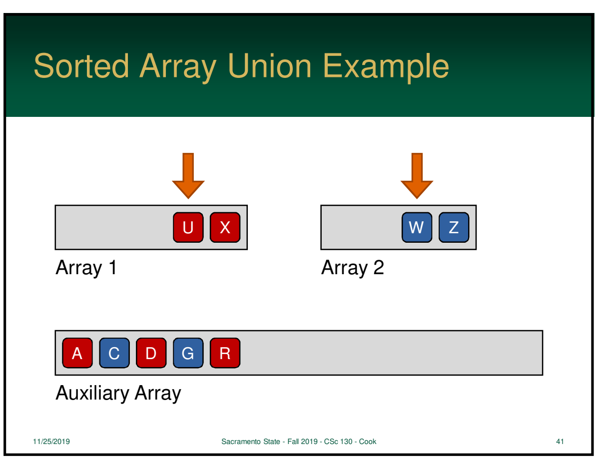


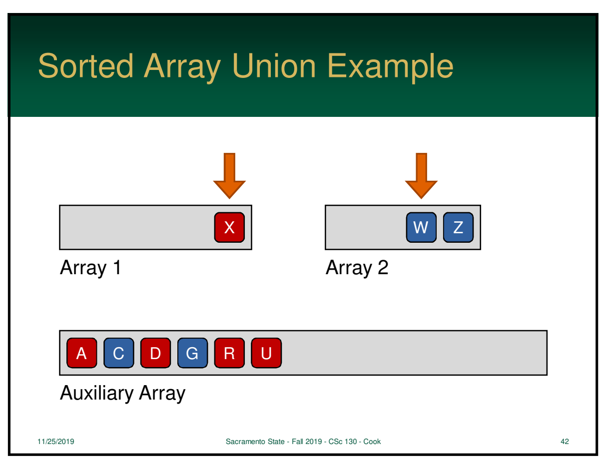

Sorted Array Union Example
D R U X
Array 1
C G W Z
Array 2
A
Auxiliary Array
11/25/2019
Sacramento State - Fall 2019 - CSc 130 - Cook
37


Sorted Array Union Example
D R U X
Array 1
G W Z
Array 2
A C
Auxiliary Array
11/25/2019
Sacramento State - Fall 2019 - CSc 130 - Cook
38


Sorted Array Union Example
R U X
Array 1
G W Z
Array 2
A C D
Auxiliary Array
11/25/2019
Sacramento State - Fall 2019 - CSc 130 - Cook
39


Sorted Array Union Example
R U X
Array 1
W Z
Array 2
A C D G
Auxiliary Array
11/25/2019
Sacramento State - Fall 2019 - CSc 130 - Cook
40


Sorted Array Union Example
U X
Array 1
W Z
Array 2
A C D G R
Auxiliary Array
11/25/2019
Sacramento State - Fall 2019 - CSc 130 - Cook
41


Sorted Array Union Example
X
Array 1
W Z
Array 2
A C D G R U
Auxiliary Array
11/25/2019
Sacramento State - Fall 2019 - CSc 130 - Cook
42

Sorted Array Union Example

X

Array 1

Z

Array 2

A C D G R U W

Auxiliary Array

Sorted Array Union Example

Array 1

Z

Array 2

A C D G R U W X

Auxiliary Array

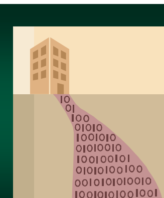
Sorted Array Union Example

Array 1

Array 2

A C D G R U W X Z

Auxiliary Array

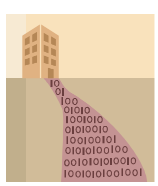


Bit Vectors

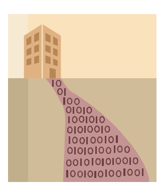
Sets and Bits

Bit Vectors

- A *bit vector* can store finite, countable sets using bits
- Also known as a *bit array*, *bit set*, and *bit map*
- Compact format that can perform a set operations with a single operation *(fast!)*



Bit Vectors



- Each object in the universe is represented as a single bit in the string of bits
- If the $x \in A$, then the bit is 1, otherwise 0

Example 1

- U = { fry, bender, farnsworth, leela, zoidberg}
- A = { fry, bender, leela }

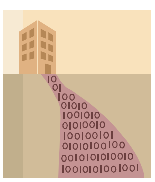
```
U = 11111
A = 11010
```


Example 2

- U = { 2, 3, 5, 7, 11, 13, 17, 19}
- A = { 3, 5, 11, 19 }

```
U = 11111111
A = 01101001
```


Why this is useful



- Computers can easily perform and & or operations on bytes (or multiple bytes)
- This means set operations can be performed amazingly fast

Let's look at the definitions again…

- The definitions of union and intersection are nearly identical
- The relationship between the elements is defined using an *and* or *or*

$$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$$
$$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$$

Let's look at the definitions again…

- We can apply a *bit-wise-and* & a *bit-wise-or* to our bit array
- It will apply the operation to each of the bits in matching columns

$$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$$
$$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$$

Let's look at the definitions again…

- So, each bit in A will be compared to its matching bit in B
- Bit match can do sets!

$$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$$
$$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$$

Example: Union (using or)

```
U = {a,b,c,d,e,f,g}
A = {b,c,d} = 0111000
B = {d,e,f} = 0001110

    0111000
 or 0001110
    0111110 = {b,c,d,e,f}
```


Example: Intersection (using and)

```
U = {a,b,c,d,e,f,g}
A = {b,c,d} = 0111000
B = {d,e,f} = 0001110

    0111000
and 0001110
    0001000 = {d}
```


Complement

- Then, how do we do a complement of a set A?
- We must flip all the bits from 1 to 0, and 0 to 1
- We can use a *binary-not* or the *XOR* operation

$$A' = \{ x \mid x \notin A \}$$

Example: Complement (using not)

```
U = {a,b,c,d,e,f,g}
A = {b,c,d} = 0111000

not 0111000
    1000111 = {a,e,f,g}
```


Exclusion

- Finally, how do we do set difference?
- The "subtract" operator will not work
- Let's look at the definition a bit more closely

$$A - B = \{ x \mid x \in A \text{ and } x \notin B \}$$

Exclusion

- It's essentially the definition of *intersection*
- Except, the second operand is the definition of *complement*.

$$A - B = \{ x \mid x \in A \text{ and } x \notin B \}$$

Example: Intersection (using and)

```
U = {a,b,c,d,e,f,g}
A = {b,c,d,f} = 0111010
B = {d,e,f} = 0001110

B' = not 0001110 = 1110001

    0111010
and 1110001
    0110000 = {b,c}
```


Java/C Code

```
Intersection  :  a & b
Union         :  a | b
Complement    :  ~a
Exclusion     :  a & ~b
```

&& and || are Boolean. These are bit-wise.

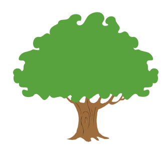
The tilde ~ is a bitwise not.

Sets using Binary Trees

Hey, we just covered that!

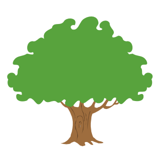
Sets using Binary Trees

- Obviously, we can also use trees to store a set
- There are a number of advantages:
  - O(log n) lookups
  - O(log n) appends



Sets using Binary Trees

- Binary Trees are a tad more complex (and advanced) than linked-lists and arrays
- How do we merge two binary trees? Let's assume they are beautifully balanced using AVL or Red-Black
- There are <u>two</u> approaches



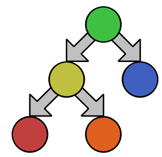
#1. Add Contents from A to B

- We can simply read all the content of tree A and then add it to tree B
- Each value will be added in O(log n)
- So, to do a union using this approach takes O(n log n) – which is a tad slower than linked lists

#2. Convert to Linked Lists

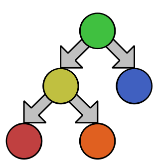
- If the set operator for Union is a "true function" – *it only returns an object and doesn't modify either operand* – then we can use linked lists
- The conversion from a tree to a linked list is O(n) – we do an infix traversal

#2. Convert to Linked Lists



- So, we can do an infix traversal of both tree A and tree B and output sorted lists
- This will take O(n)
- The merging of two sorted linked lists also is O(n)

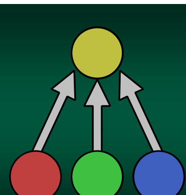
#2. Convert to Linked Lists



- This linked list, *because it is sorted*, can be converted back into a B-Tree in O(n)
- But, please note, because of all the conversions, it is a rather ugly O(n)

Set Implementations

	Bit Vector	Sorted Array	Unsorted Linked List	Sorted Linked List	Tree
Limitations	Countable, Finite	None	Multiset	None	None
Find	O(1)	O(log n)	O(n)	O(n)	O(log n)
Append	O(1)	O(n)	O(1)	O(n)	O(log n)
Union	O(1)	O(n)	$O(n^2)$	O(n)	O(n)
Intersection	O(1)	O(n)	$O(n^2)$	O(n)	O(n)
Difference	O(1)	O(n)	$O(n^2)$	O(n)	O(n)

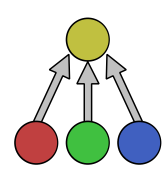


Union-Find

For when your union has to be there overnight… er… O(1)

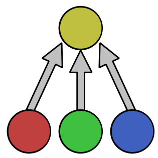
Union-Find

- Often there are situations where we have a number of known objects
- … and we want to quickly associate them together
- In other words, we want to create ad-hoc unions



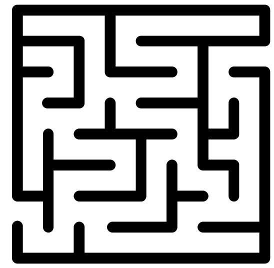
Union-Find

- *Union-Find* data structure maintains a list of nodes *partitioned* into a number of disjoint subsets
- The union of all disjoint sets is the Universe – i.e. all the nodes



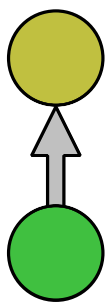
Example Applications

- Kruskal's Algorithm uses it to create sets of edges – to prevent cycles
- Maze creation algorithms use it track sets of connected paths – so the maze is always solvable

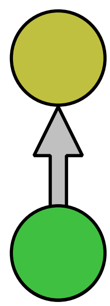


The Approach

- Sets are stored as a variation of the classic tree
- However, in this approach children link to their parents
- So, the branches point *"backwards"* from standard trees

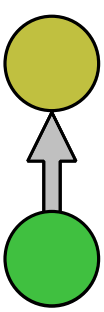


Arrangement of Nodes



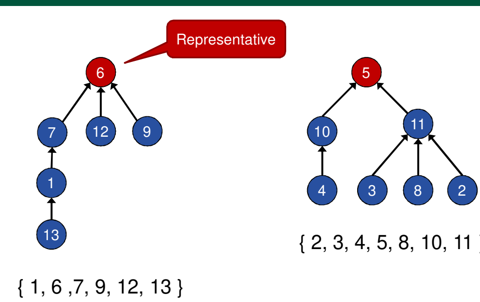
- A parent node can have multiple children - this is not a binary tree!
- Every node - in the tree - is part of the same set
- The root is called the *representative* of the set

Arrangement of Nodes



- Any node can "find" its representative – by following the upwards branches
- If any two nodes have the same representative, then they are in the same set

Examples



Union-Find Operations

- The Union-Find data structure contains 3 operations that handle sets
- All three can modify the structure of the tree – which automatically optimizes itself into **O(1)**

Union-Find API

public class UnionFind	
void	makeSet(object value)
void	union(object a, object b)
object	find(object a)

Makeset

- Creates a set (within the Union-Find instance) that contains only a single element
- Also known as a *singleton*
- Essentially, this is the same as an "add"

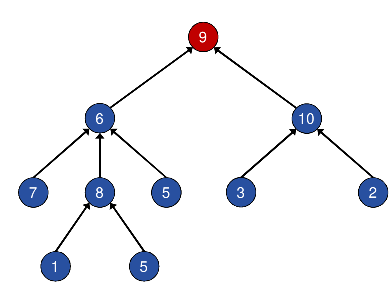
Find

- The *find operation* starts with an node and locates its representative
- It also optimizes the tree at the same time by updating links

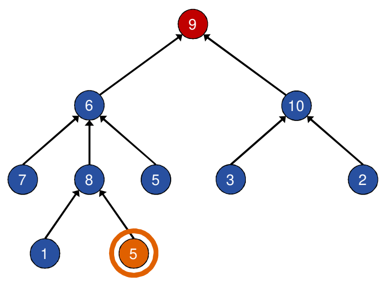
Find

- After the answer is found, all nodes, along the path, are pointed to the representative
- This means the tree optimizes by becoming "flattened" (and faster) with each find

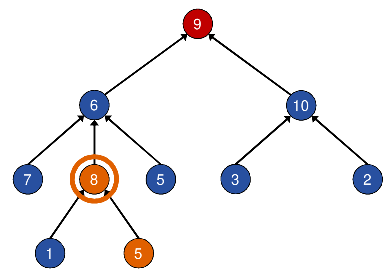
Find(5) → 9



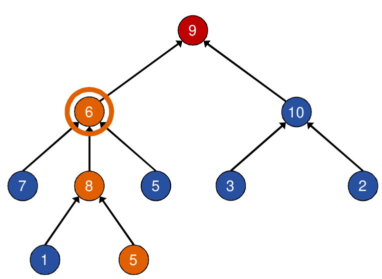
Find(5) → 9



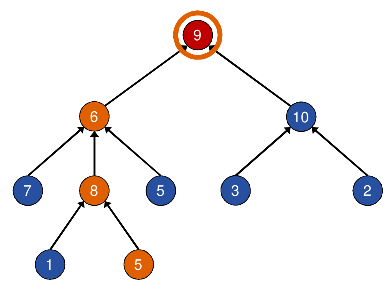
Find(5) → 9



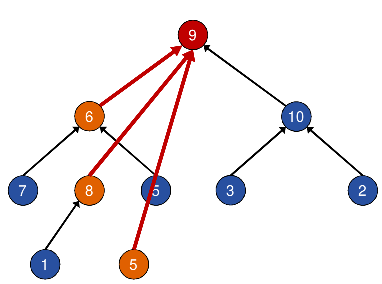
Find(5) → 9



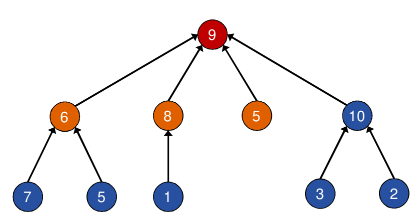
Find(5) → 9



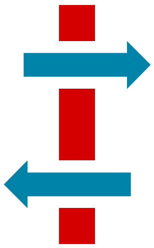
Update Path Branches



Updated, Flattened, Tree

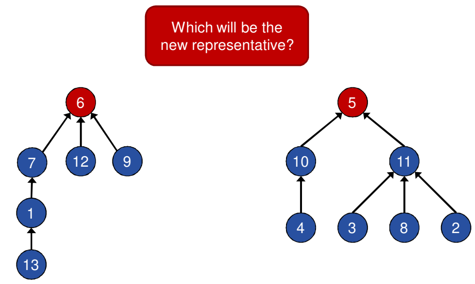


Union



- Combines two sets into a single set
- This is accomplished by updating a single parent link
- So, one representative will be pointed at another representative

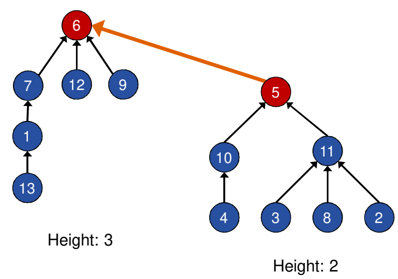
So, which one do we choose?



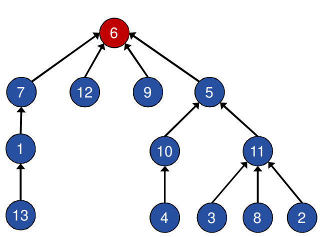
Approach #1: Union By Height

- Using *Union by Height*, the tree with the smaller height is made a subtree of the taller tree
- This helps create a more balanced union in terms of height
- But, Find, will fix this automatically

Approach #1: Union By Height



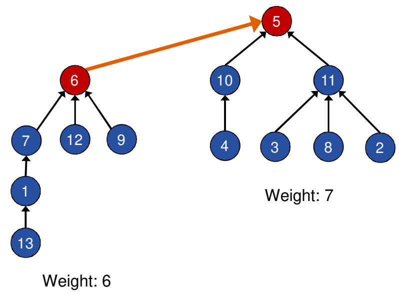
Approach #1: Result



Approach #2: Union by Weight

- Using *Union by Weight*, the tree with the fewer nodes (not edge weight) is made a subtree of the larger tree
- This helps create a more balanced union in terms of weight
- Again, the Find operation will optimize the tree

Approach #1: Union by Weight



Approach #1: Result