Set Data Structures

Part 12

1

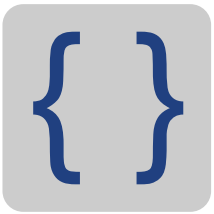
Importance of Sets

Organizing Information

2

Importance of Sets

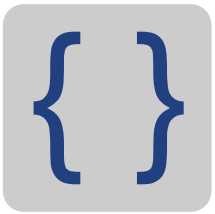
- A *set* is an unordered collection of "objects"
- Sets are used in computer science is a wide variety of ways



3

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



4

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
```


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

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

6

Set Review: Intersection

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

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

7

Set Review: Difference

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

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

8

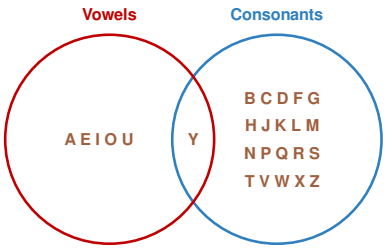
Set Review: Complement

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

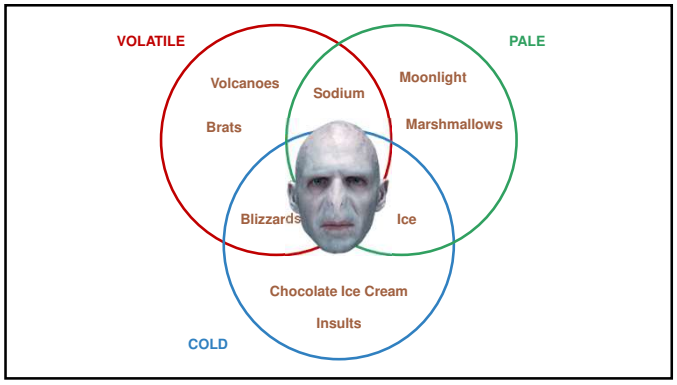
$$A' = \{ x \mid x \notin A \}$$

9

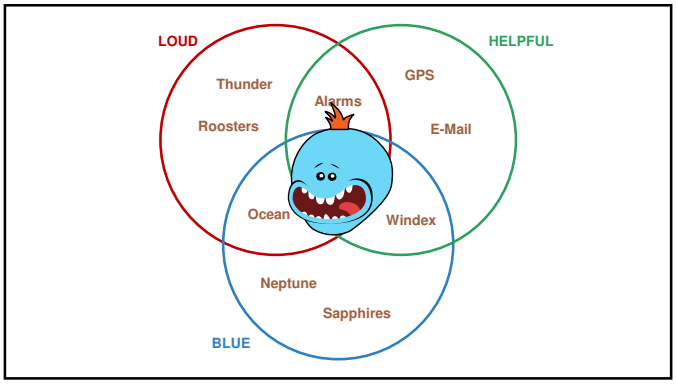
Example: Letters in English



10



11

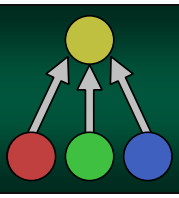


12

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

13



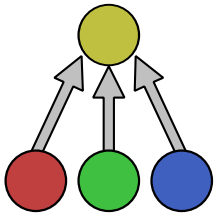
Union-Find

When it absolutely, positively has to be unioned overnight … er… O(1)

14

Union-Find

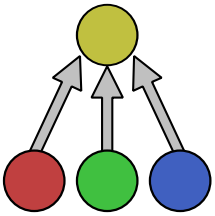
- Often there are situations where we have a number of known objects
- … i.e. a finite countable set
- … and we want to quickly union them together



15

Union-Find

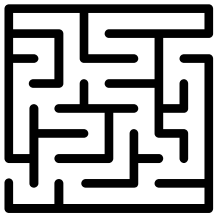
- *Union-Find* data structure maintains a list of nodes *partitioned* into *disjoint* subsets
- e.g. { {a} {b} {c} {d} {e} } is a full partition of {a,b,c,d,e}



16

Example Applications

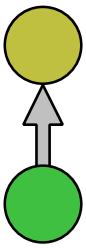
- Kruskal's Tree-spanning Algorithm – uses a set of known edges
- Maze creation algorithms use it to track sets of connected paths – so the maze is always solvable



17

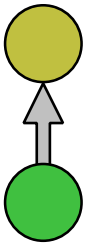
The Approach

- 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 – i.e. upwards



18

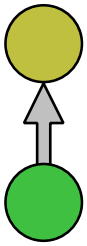
Arrangement of Nodes



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

19

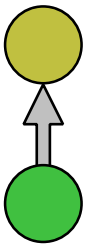
Arrangement of Nodes



- The root is called the *representative* of the set
- This node is **not** special nor is its value special
- … it just happens to be the node the represents the set

20

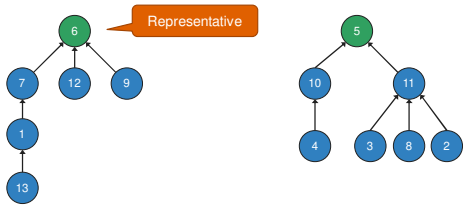
Arrangement of Nodes



- 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

21

Examples



{ 1, 6 ,7, 9, 12, 13 }

{ 2, 3, 4, 5, 8, 10, 11 }

22

Union-Find Operations

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

23

Union-Find Interface

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

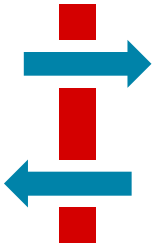

24

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" for the ADT

25

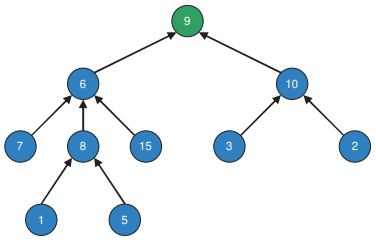
Find



- The *find operation* starts with a node and locates its *representative*
- Travels up the tree until it finds a node without a link (which is the representative)

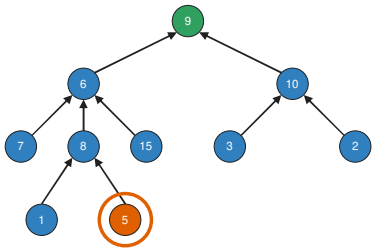
26

Find(5) → 9



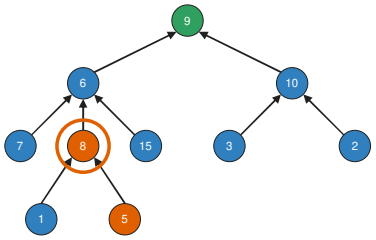
27

Find(5) → 9



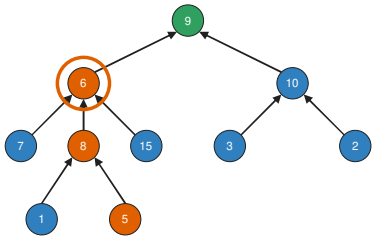
28

Find(5) → 9



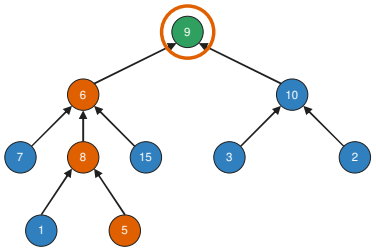
29

Find(5) → 9



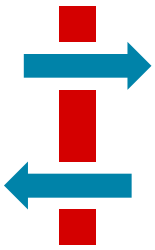
30

Find(5) → 9



31

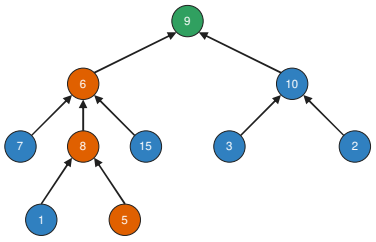
Find Automatically Optimizes the Tree



- 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

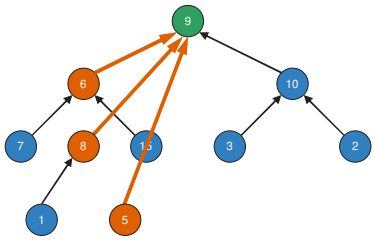
32

Connect Each Node to Representative



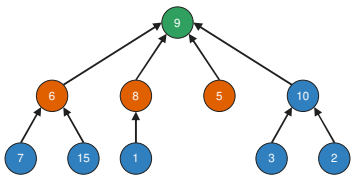
33

Connect Each Node to Representative



34

Updated, Flattened, Tree



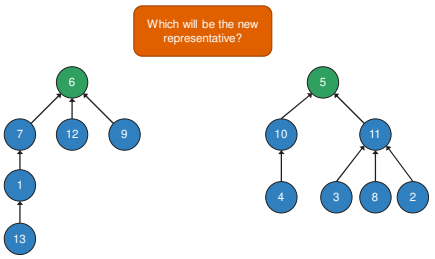
35

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

36

So, which one do we choose?



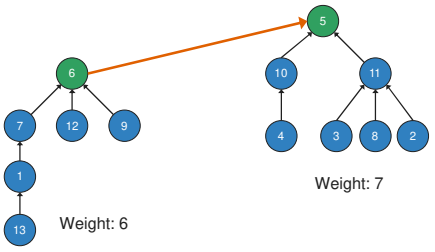
37

Approach #1: 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

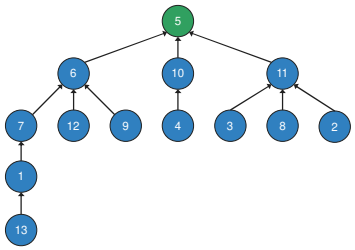
38

So, which one do we choose?



39

Approach #1: Result



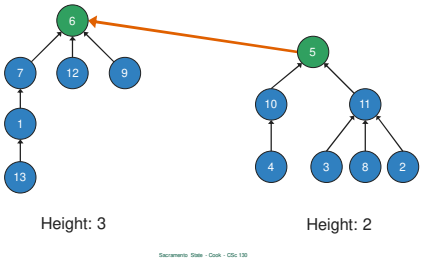
40

Approach #2: 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

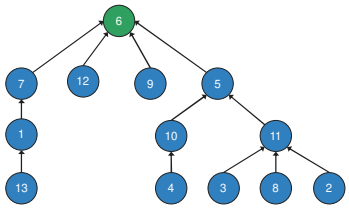
41

Approach #2: Union By Height

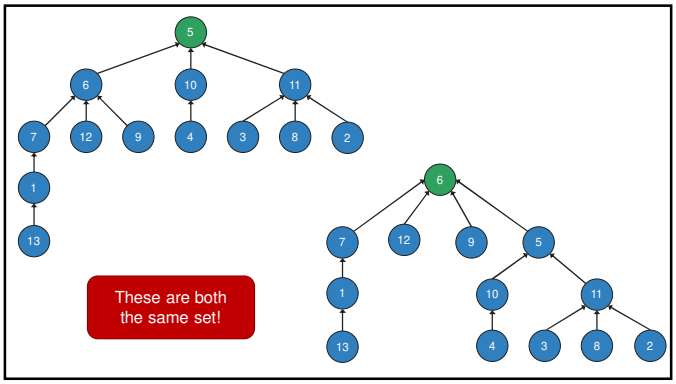


42

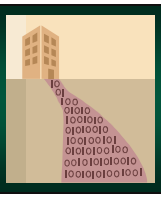
Approach #2: Result



43



44



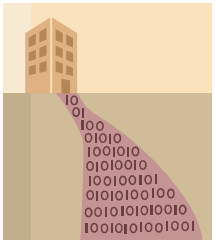
Bit Vectors

Sets and Bits

45

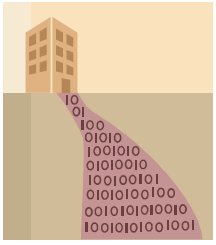
Bit Vectors

- 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!)*



46

Bit Vectors



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

47

Example 1

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

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


48

Example 1

- U = { rick, beth, jerry, morty, summer }
- A = { rick, morty }

```
U = 11111
A = 10010
```


49

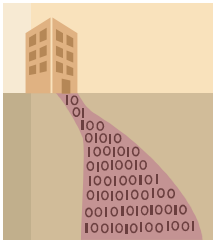
Example 2

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

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


50

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

51

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*

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

52

Let's look at the definitions again…

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

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

53

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!

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

54

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}
```


55

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}
```


56

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

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

57

Example: Complement (using not)

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

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


58

Exclusion

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

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

59

Exclusion

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

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

60

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}
```


61

Java/C Code

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

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

The tilde ~ is a bitwise not.

62