CSE 373: Data Structures & Algorithms
Disjoint Sets & Union-Find

Riley Porter

Winter 2017

Course Logistics

- Hashing topic summary out now (thanks Matthew!)
- HW3 still out. Some changes to clarify based on common confusion popping up:
  - WordInfo objects now have a hashCode()
  - some clarification in the Mutability paragraph in implementation notes
  - rehashing pseudocode on the topic summary AND in section tomorrow
- Midterm a week from Friday, we'll do review in lecture next week

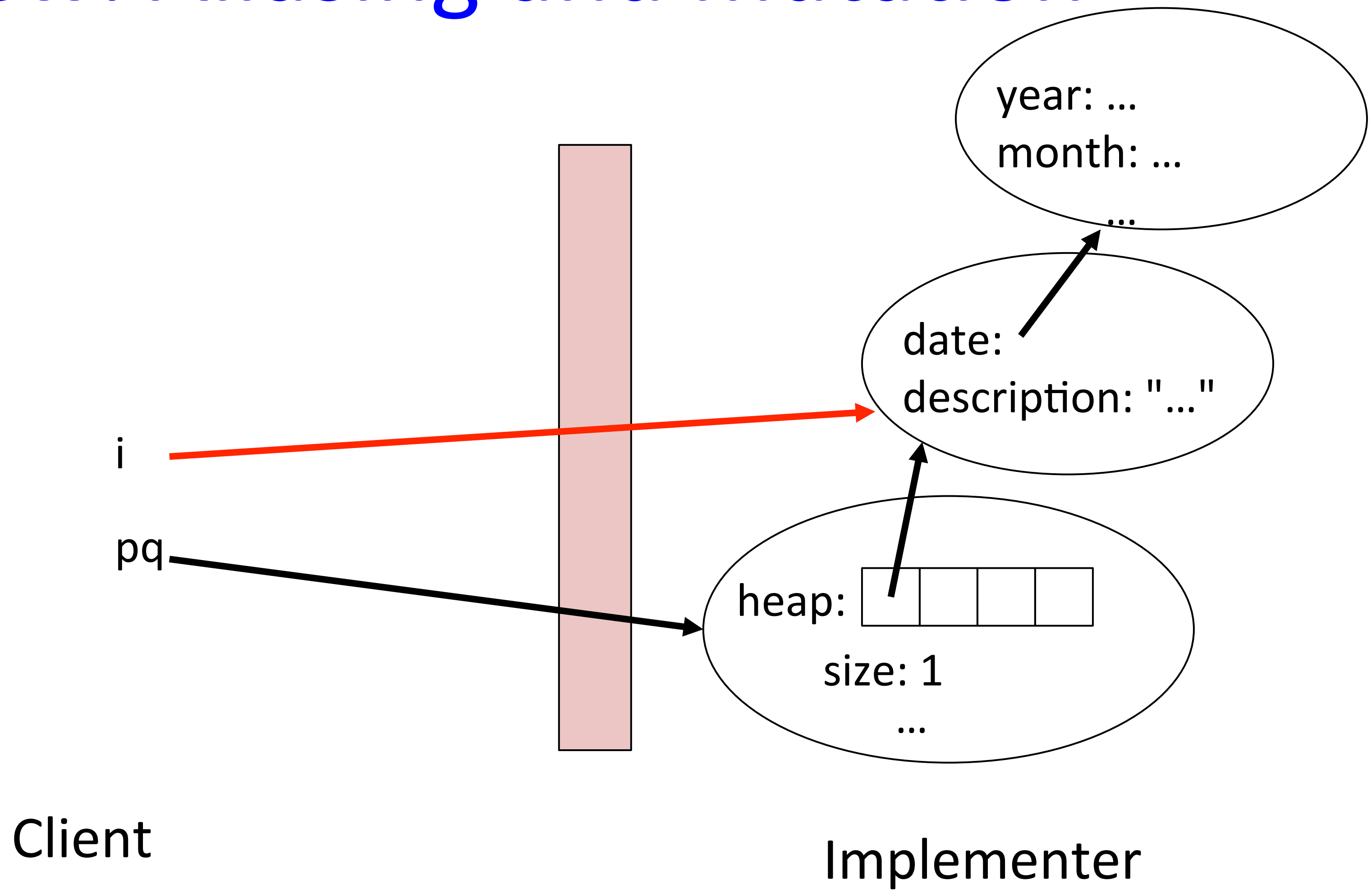
Review: Abstractions from Monday

```
public class ToDoPQ {
    private ToDoItem[] heap;
    private int size;
   void insert(ToDoItem t) {…}
   ToDoItem deleteMin() {…}
    // more methods
}
public class ToDoItem {
    private Date date;
    private String description;
    // methods
}
public class Date {
    private int day;
    private int month;
    private int year;
   // methods
}
```

What could go wrong with these classes depending on their implementation?

Think of some client code that might break abstraction through aliases.

Review: Aliasing and mutation



Review: The Fix

- How do we protect against aliases getting passed back to the client?
  - Copy-in and Copy-out: whenever the client gives us a new object to store or whenever we're giving the client a reference to an object, we better copy it.
  - Deep copying: copy the objects all the way down
  - Immutability: protect by only storing things that can't change. Deep copy down to the level of immutability

New Topic!

Consider this problem. How would you implement a solution?

Given: `Set<Set<String>> friendGroups` representing groups of friends. You can assume unique names for each person and each person is only in one group.

Example input:

```
[
 ["Riley", "Pascale", "Matthew", "Hunter"],
 [ "Chloe", "Paul", "Zelina"],
 [ "Rebecca", "Raquel", "Trung", "Kyle", "Josh"]
]
```

Problem: Given two Strings "Pascale" and "Raquel" determine if they are in the same group of friends.

Solution Ideas

1. Traverse each Set until you find the Set containing the first name, then see if it also contains the second name.
2. Store a map of people to the set of people they are friends with. Then find the Set of friends for the first name and see if it contains the second name. Note, this works for friends in multiple groups as well.

```
[
"Riley” → ["Pascale", "Matthew", "Hunter"],
"Pascale” → [”Riley", "Matthew", "Hunter"],
...]
```

3. Store friendship in a Graph. A lot like solution 2 actually. We’re not there yet, but we’ll get there soon.
4. Disjoint Sets and Union-Find (new today and Friday!)
5. Others?

Disjoint Sets and Union Find: the plan

- What are sets and *disjoint sets*
- The union-find ADT for disjoint sets

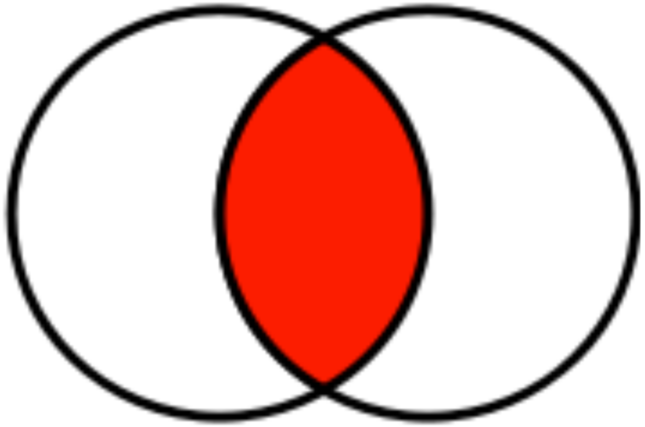
Friday:

- Basic implementation with "up trees"
- Optimizations that make the implementation much faster

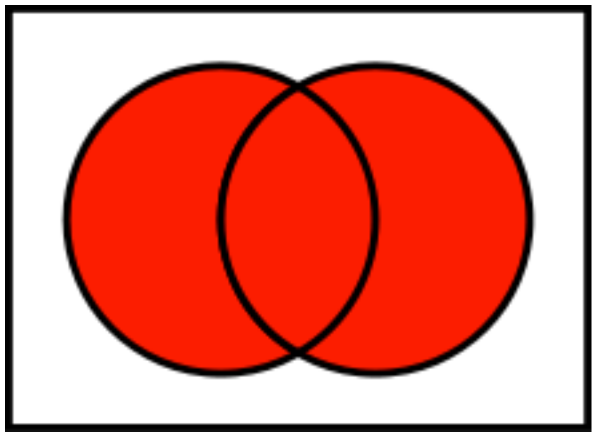
Terminology

Empty set: ∅

Intersection ∩



Union ∪



Notation for elements in a set:

Set S containing e1, e2 and e3: **{e1, e2, el3}**

e1 is an element of S: **e1 ∈ S**

Disjoint sets

- A set is a collection of elements (no-repeats)
- Every set contains the empty set by default
- Two sets are disjoint if they have no elements in common
  - $\mathbf{S_1 \cap S_2 = \varnothing}$
- Examples:
  - {a, e, c} and {d, b} Disjoint
  - {x, y, z} and {t, u, x} Not disjoint

Partitions

A partition P of a set S is a set of sets $\{S_1,S_2,\ldots,S_n\}$ such that every element of S is in **exactly one** S_i

Put another way:

- $S_1 \cup S_2 \cup \ldots \cup S_k = S$
- For all i and j, $i \neq j$ implies $S_i \cap S_j = \varnothing$ (sets are disjoint with each other)

Example: Let S be {a,b,c,d,e}

- {a}, {d,e}, {b,c} Partition
- {a,b,c}, ∅, {d}, {e} Partition
- {a,b,c,d,e} Partition
- {a,b,d}, {c,d,e} Not a partition, not disjoint, both sets have d
- {a,b}, {e,c} Not a partition of S (doesn't have d)

Union Find ADT: Operations

- Given an unchanging set *S*, **`create`** an initial partition of a set
  - Typically each item in its own subset: {a}, {b}, {c}, …
  - Give each subset a "name" by choosing a *representative element*

- Operation **`find`** takes an element of *S* and returns the representative element of the subset it is in

- Operation **`union`** takes two subsets and (permanently) makes one larger subset
  - A different partition with one fewer set
  - Affects result of subsequent **`find`** operations
  - Choice of representative element up to implementation

Subset Find for our problem

- Given an unchanging set *S*, **`create`** an initial partition of a set

```
    “Riley” -> ["Riley", "Pascale", "Matthew", "Hunter"],
    “Chloe” -> [ "Chloe", "Paul", "Zelina"],
    “Rebecca” -> [ "Rebecca", "Raquel", "Trung", "Kyle",
"Josh"]
```

- Operation **`find`** takes an element of *S* and returns the representative element of the subset it is in

```
find(“Pascale”) returns “Riley”
find(“Chloe”) returns “Chloe”
```

Not the same subset since not the same representative

Union of two subsets for our problem

- Operation **`union`** takes two subsets and (permanently) makes one larger subset

Chloe and Riley become friends, merging their two groups. Now those to subsets become one subset. We can represent that in two ways:

Merge the sets:

```
"Chloe" -> [ "Chloe", "Paul", "Zelina", "Riley",
             "Pascale", "Matthew", "Hunter"]
```

Or tell Riley that her representative is now Chloe, and on find anyone in Riley's old subset like `find("Pascale")` see what group Riley is in:

```
"Riley" -> ["Pascale", "Matthew", "Hunter"],
"Chloe" -> [ "Chloe", "Paul", "Zelina",
             "Riley"]
```

Either way, `find("Pascale") returns "Chloe"`

Another Example

- Let S = {1,2,3,4,5,6,7,8,9}
- Let initial partition be (will highlight representative elements red)

  {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}
- **`union`**(2,5):

  {1}, {2, 5}, {3}, {4}, {6}, {7}, {8}, {9}
- **`find`**(4) = 4, **`find`**(2) = 2, **`find`**(5) = 2
- **`union`**(4,6), **`union`**(2,7)

  {1}, {2, 5, 7}, {3}, {4, 6}, {8}, {9}
- **`find`**(4) = 6, **`find`**(2) = 2, **`find`**(5) = 2
- **`union`**(2,6)

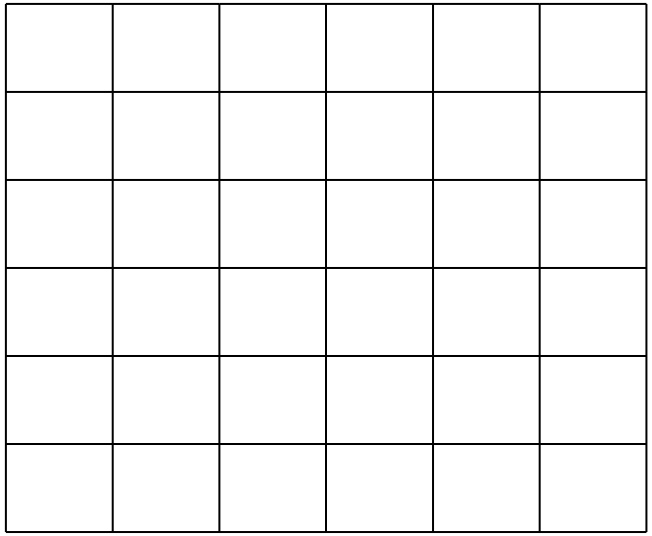
  {1}, {2, 4, 5, 6, 7}, {3}, {8}, {9}

No other operations

- All that can "happen" is sets get unioned
  - No "un-union" or "create new set" or …
- As always: trade-offs – implementations are different
  - ideas? How do we maintain "representative" of a subset?
- Surprisingly useful ADT, but not as common as dictionaries, priority queues / heaps, AVL trees or hashing

Example application: maze-building

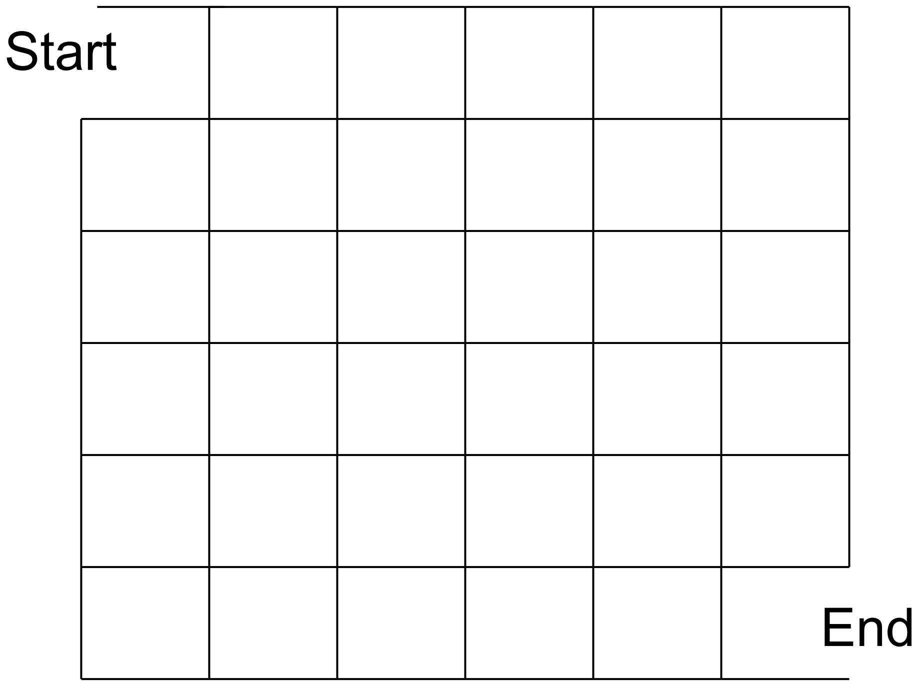
- Build a random maze by erasing edges



Criteria:
- Possible to get from anywhere to anywhere
- No loops possible without backtracking
  - After a "bad turn" have to "undo"

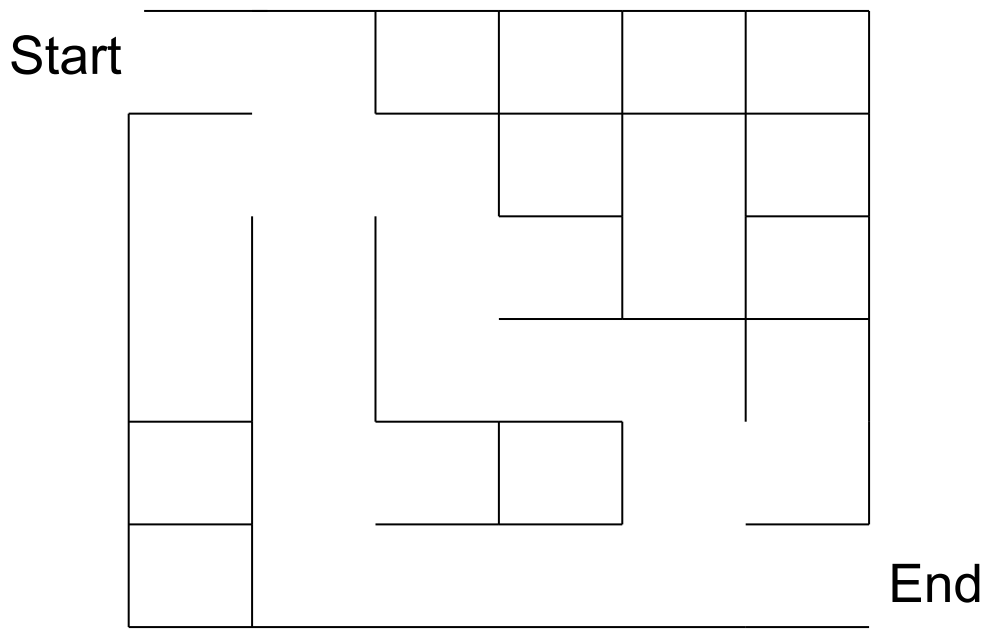
Maze building

Pick start edge and end edge



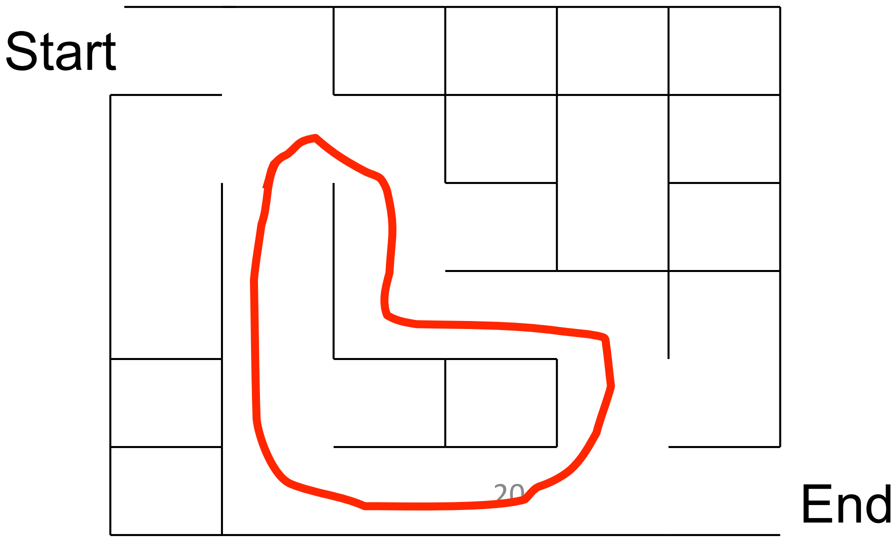
Repeatedly pick random edges to delete

One approach: just keep deleting random edges until you can get from start to finish



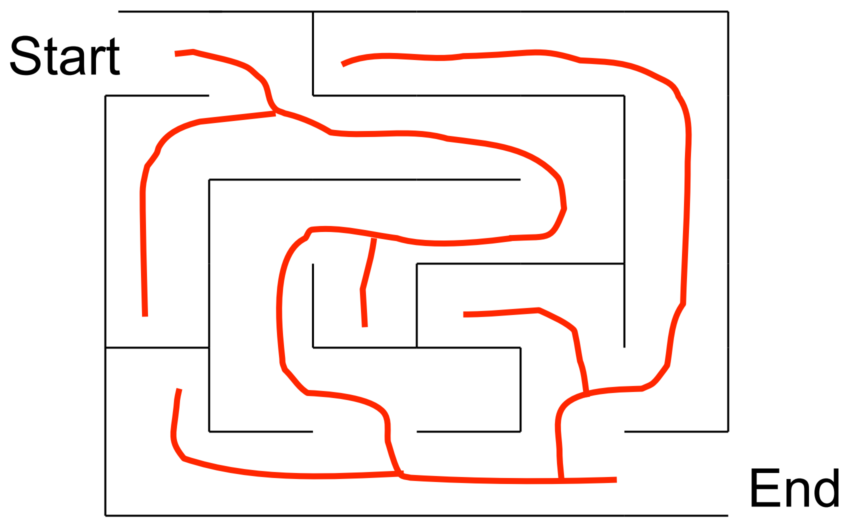
Problems with this approach

1. How can you tell when there is a path from start to finish?
   – We do not really have an algorithm yet (Graphs)
2. We have *cycles*, which a "good" maze avoids
3. We can't get from anywhere to anywhere else



Revised approach

- Consider edges in random order
- But only delete them if they introduce no cycles (how? TBD)
- When done, will have one way to get from any place to any other place (assuming no backtracking)



- Notice the funny-looking *tree* in red

Cells and edges

- Let's number each cell
  - 36 total for 6 x 6
- An (internal) edge (x,y) is the line between cells x and y
  - 60 total for 6x6: (1,2), (2,3), …, (1,7), (2,8), …

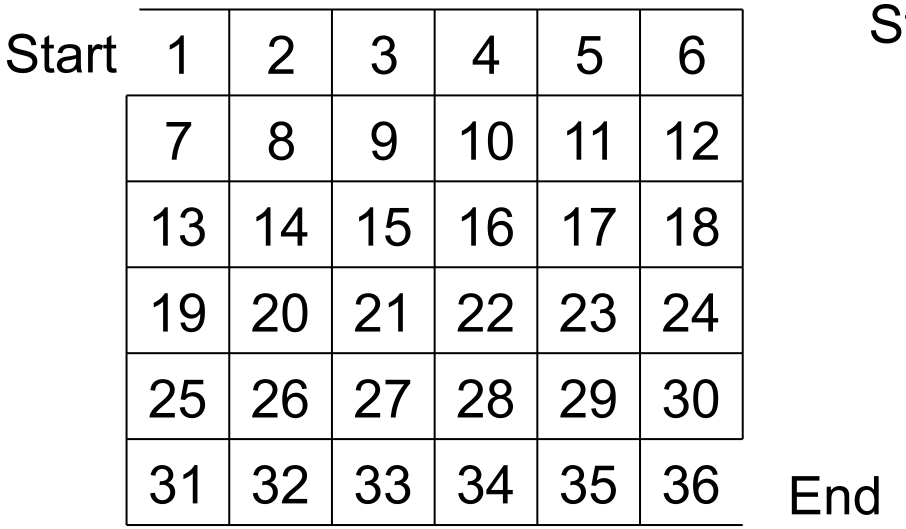
Start

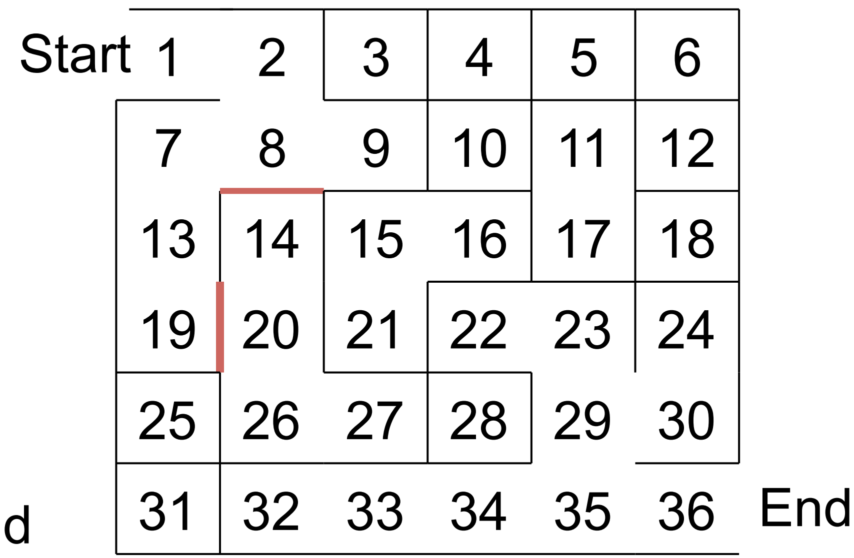
1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

End

The trick

- Partition the cells into **disjoint sets**: "are they connected"
  - Initially every cell is in its own subset
- If an edge would connect two different subsets:
  - then remove the edge and **union** the subsets
  - else leave the edge because removing it makes a cycle





Pseudocode of the algorithm

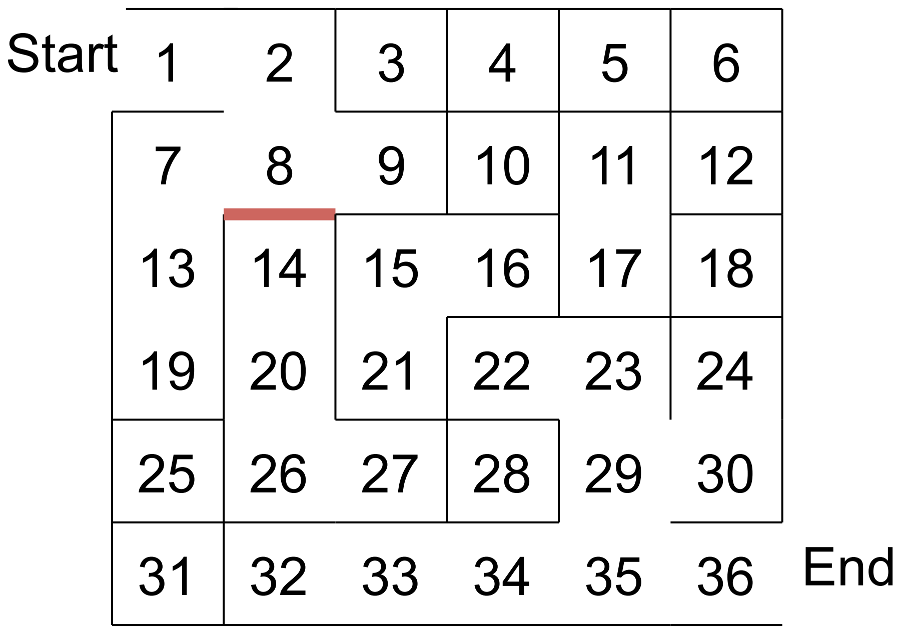
- Partition = **disjoint sets** of connected cells, initially each cell in its own 1-element set
- Edges = **set** of edges not yet processed, initially all (internal) edges
- Maze = **set** of edges kept in maze (initially empty)

```
// stop when possible to get anywhere
while Partition has more than one set {
    pick a random edge (cell_1,cell_2) to remove from
    Edges
    set_1 = find(cell_1)
    set_2 = find(cell_2)
    if set_1 == set_2:
        // same subset, do not create cycle
         add (cell_1, cell_2) to Maze
    else:
        // do not put edge in Maze, connect subsets
        union(set_1, set_2)
}
```

Add remaining members of Edges to Maze, then output Maze

pick random Edge step

Pick (8,14)



{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

Example pick random Edge step

Partition:
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

Chosen
Edge: (8, 14)

Find(8) = 7
Find(14) = 20

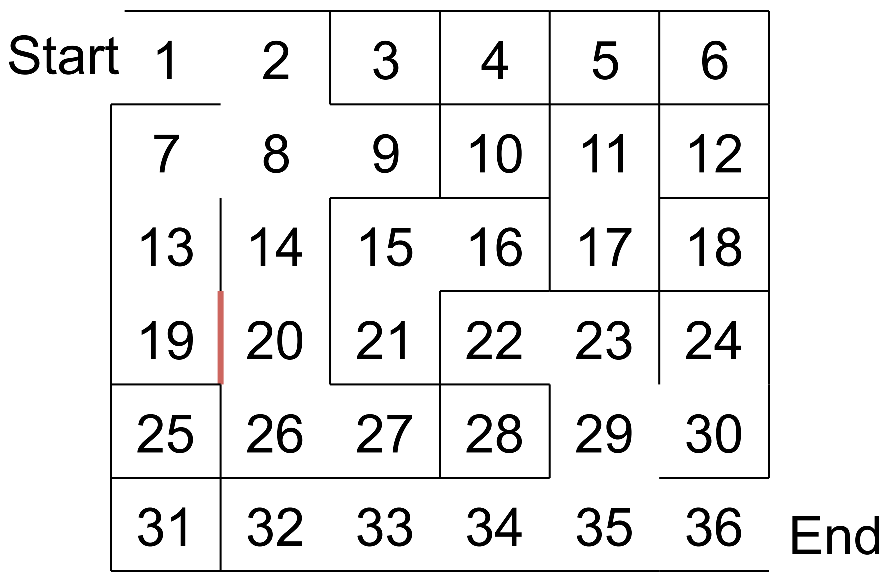
Union(7,20)

Since we unioned the two sets, we "deleted" the edge and don't add the edge to our Maze

Partition:
{1,2,7,8,9,13,19,14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

Add edge to Maze step

Pick (19,20)

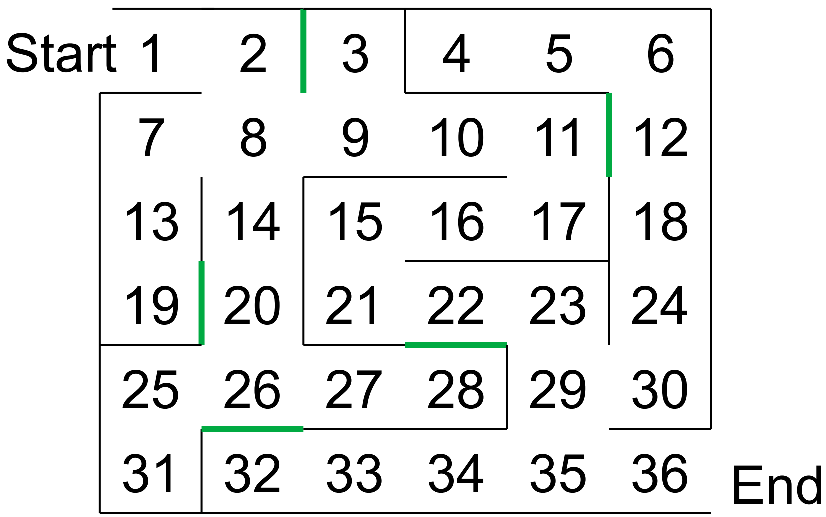


Since we didn't union the sets together, we don't want to delete this edge (it would introduce a cycle). We add the edge (19,20) to our Maze.

Partition:
{1,2,7,8,9,13,19,14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
{18}
{25}
{28}
{31}
{22,23,24,29,30,32
33,34,35,36}

At the end

- Stop when Partition has one set
- Suppose green edges are already in Maze and black edges were not yet picked
  - Add all black edges to Maze



Partition:
{1,2,3,4,5,6,7,… 36}

Applications / Thoughts on Union-Find

- Maze-building is cute ☺ and a surprising use of the union-find ADT
- Many other uses:
  - Road/network/graph connectivity (will see this again)
    - "connected components" e.g., in social network
  - Partition an image by connected-pixels-of-similar-color
  - Type inference in programming languages
- Our friend group example could be done with Graphs (we'll learn about them later) but we can use Union-Find for a much less storage intense implementation. Cool! ☺
- Union-Find is not as common as dictionaries, queues, and stacks, but valuable because implementations are very fast, so when applicable can provide big improvements

Today's Takeaways

- Understand:
    - disjoint sets, partitions, and set notation
    - find operation
    - union operation
    - Maze application
- Be thinking about how you might implement this. How do you store a subset? How do you know what the "representative" is? How do you merge? (We'll talk about it on Friday)