CSE373: Data Structures & Algorithms
Lecture Supplement: Amortized Analysis

Hunter Zahn

Summer 2016

Announcements

- HW1 grade out*
  - Some feedback still coming your way: by the end of the day

Amortized

- Recall our plain-old stack implemented as an array that doubles its size if it runs out of room
  - How can we claim **`push`** is $O(\mathbf{1})$ time if resizing is $O(\mathbf{n})$ time?
  - We *can't*, but we *can* claim it's an $O(\mathbf{1})$ amortized operation
- What does amortized mean?
- When are amortized bounds good enough?
- How can we prove an amortized bound?

Will just do two simple examples

- Text has more sophisticated examples and proof techniques
- *Idea* of how amortized describes average cost is essential

Amortized Complexity

If a sequence of **M** operations takes $O(\mathbf{M}\,\mathbf{f(n)})$ time, we say the amortized runtime is $O(\mathbf{f(n)})$

Amortized bound: worst-case guarantee over sequences of operations

- Example: If any **n** operations take $O(\mathbf{n})$, then amortized $O(\mathbf{1})$
- Example: If any **n** operations take $O(\mathbf{n}^3)$, then amortized $O(\mathbf{n}^2)$

- The worst case time per operation can be larger than **f(n)**
  - As long as the worst case is *always* "rare enough" in *any* sequence of operations

Amortized guarantee ensures the average time per operation for any sequence is $O(\mathbf{f(n)})$

"Building Up Credit"

- Can think of preceding "cheap" operations as building up "credit" that can be used to "pay for" later "expensive" operations

- Because any sequence of operations must be under the bound, enough "cheap" operations must come *first*
  - Else a prefix of the sequence, which is also a sequence, would violate the bound

Example #1: Resizing stack

A stack implemented with an array where we double the size of the array if it becomes full

Claim: Any sequence of **`push`**/**`pop`**/**`isEmpty`** is amortized $O(\mathbf{1})$

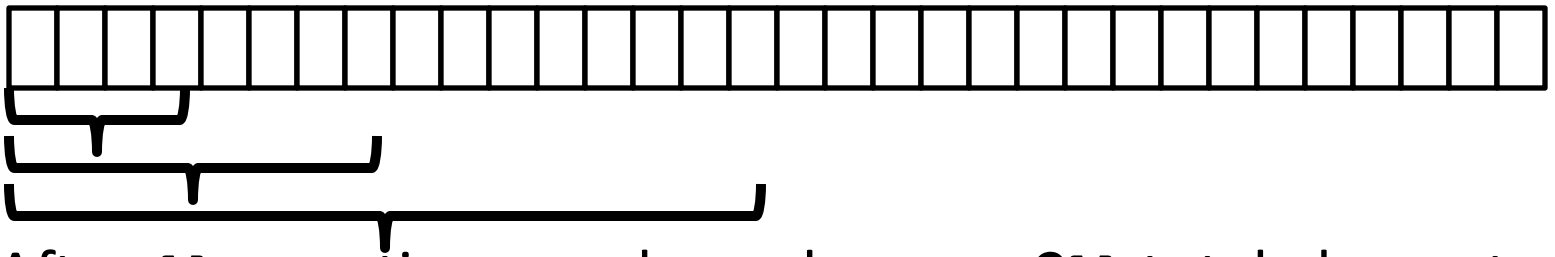
Need to show any sequence of **`M`** operations takes time $O(\mathbf{M})$

- Recall the non-resizing work is $O(\mathbf{M})$ (i.e., $\mathbf{M}*O(\mathbf{1})$)
- The resizing work is proportional to the total number of element copies we do for the resizing
- So it suffices to show that:

  After **`M`** operations, we have done **`<  2M`** total element copies

  (So average number of copies per operation is bounded by a constant)

Amount of copying



After **M** operations, we have done **< 2M** total element copies

Let **n** be the size of the array after **M** operations

- Then we have done a total of:

  n/2 + n/4 + n/8 + … INITIAL_SIZE < n

  element copies
- Because we must have done at least enough **push** operations to cause resizing up to size **n**:

  M ≥ n/2
- So

  2M ≥ n > *number of element copies*

Other approaches

- If array grows by a constant amount (say 1000), operations are not amortized $O(\mathbf{1})$
  - After $O(\mathbf{M})$ operations, you may have done $\Theta(\mathbf{M}^2)$ copies

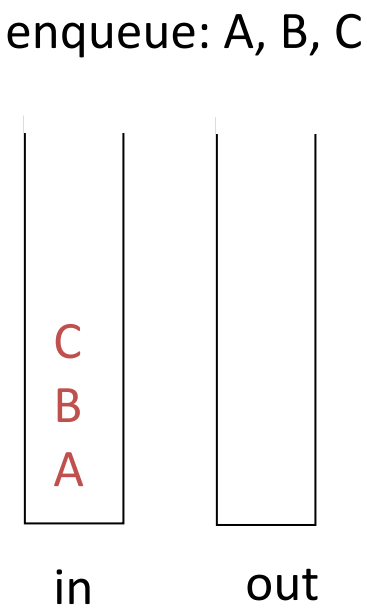
- If array shrinks when 1/2 empty, operations are not amortized $O(\mathbf{1})$
  - Terrible case: **`pop`** once and shrink, **`push`** once and grow, **`pop`** once and shrink, …

- If array shrinks when 3/4 empty, it is amortized $O(\mathbf{1})$
  - Proof is more complicated, but basic idea remains: by the time an expensive operation occurs, many cheap ones occurred

Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

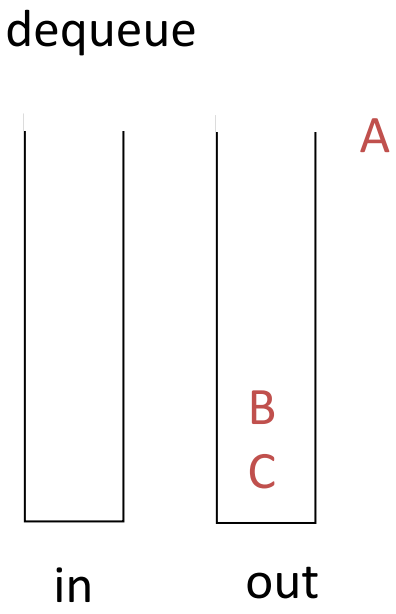
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

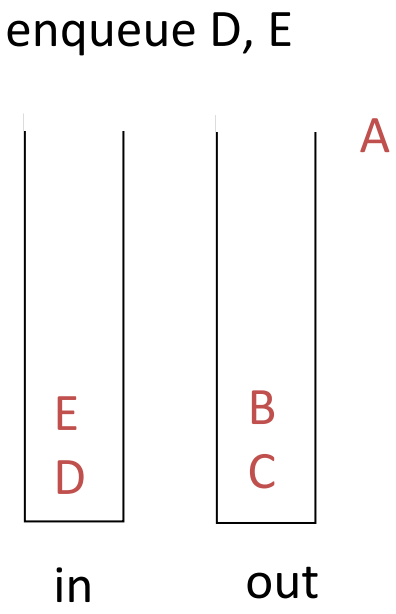
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

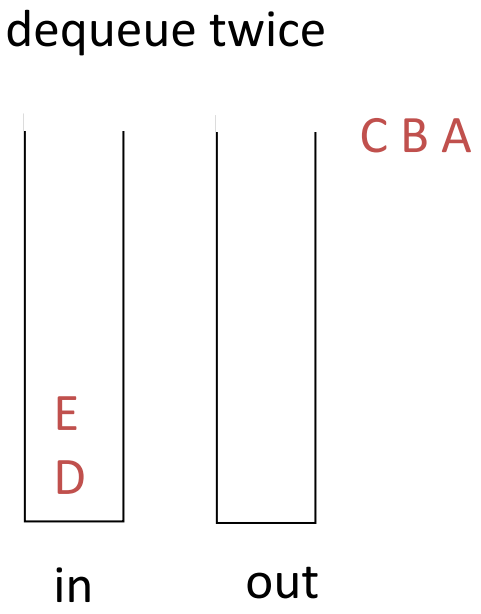
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

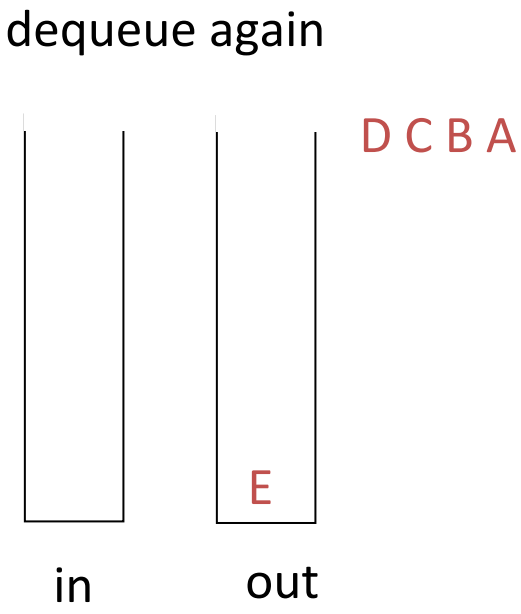
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Correctness and usefulness

- If **`x`** is enqueued before **`y`**, then **`x`** will be popped from **in** later than **`y`** and therefore popped from **out** sooner than **`y`**
  - So it is a queue

- **Example**:
  - Wouldn't it be nice to have a queue of t-shirts to wear instead of a stack (like in your dresser)?
  - So have two stacks
    - *in*: stack of t-shirts go after you wash them
    - *out*: stack of t-shirts to wear
    - if *out* is empty, reverse *in* into *out*

Analysis

- **`dequeue`** is not $O(1)$ worst-case because **out** might be empty and **in** may have lots of items

- But if the stack operations are (amortized) $O(1)$, then any sequence of queue operations is amortized $O(1)$

  - The total amount of work done per element is 1 **`push`** onto **in**, 1 **`pop`** off of **in**, 1 **`push`** onto **out**, 1 **`pop`** off of **out**

  - When you reverse **`n`** elements, there were **`n`** earlier $O(1)$ **`enqueue`** operations to average with

Amortized useful?

- When the average per operation is all we care about (i.e., sum over all operations), amortized is perfectly fine
  - This is the usual situation

- If we need every operation to finish quickly (e.g., in a web server), amortized bounds may be too weak

- While amortized analysis is about averages, we are averaging cost-per-operation on worst-case input
  - **Contrast**: Average-case analysis is about averages across possible inputs. Example: if all initial permutations of an array are equally likely, then quicksort is $O(\mathbf{n \log n})$ on average even though on some inputs it is $O(\mathbf{n}^2)$)

Not always so simple

- Proofs for amortized bounds can be much more complicated
- Example: Splay trees are dictionaries with amortized $O(\texttt{log n})$ operations
  - No extra height field like AVL trees
  - See Chapter 4.5 if curious
- For more complicated examples, the proofs need much more sophisticated invariants and "potential functions" to describe how earlier cheap operations build up "energy" or "money" to "pay for" later expensive operations
  - See Chapter 11 if curious
- But complicated *proofs* have nothing to do with the code!

CSE373: Data Structures & Algorithms
Disjoint Sets & Union-Find

Hunter Zahn

Summer 2016

The plan

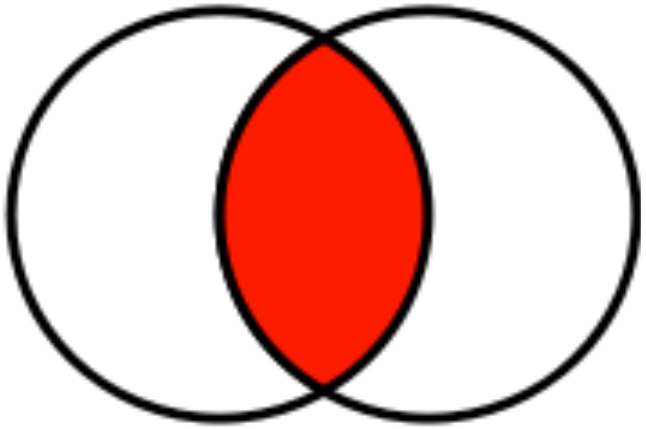
- What are *disjoint sets*
- The union-find ADT for disjoint sets
- Applications of union-find

Next lecture:

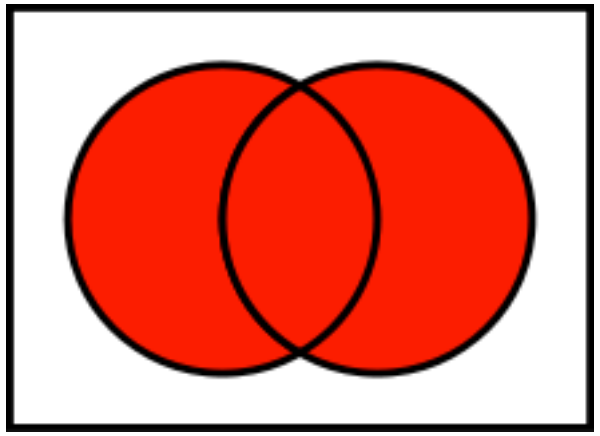
- Basic implementation of the ADT with "up trees"
- Optimizations that make the implementation much faster

Terminology

Intersection ∩



Union ∪



Empty set: ∅

Disjoint sets

- A set is a collection of elements (no-repeats)
- Two sets are disjoint if they have no elements in common
  - $\mathbf{S_1} \cap \mathbf{S_2} = \varnothing$
- Example: {a, e, c} and {d, b} are disjoint
- Example: {x, y, z} and {t, u, x} are 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}
- One partition: {a}, {d,e}, {b,c}
- Another partition: {a,b,c}, ∅, {d}, {e}
- A third: {a,b,c,d,e}
- Not a partition: {a,b,d}, {c,d,e}
- Not a partition of *S*: {a,b}, {e,c}

The plan

- What are *disjoint sets*
- The union-find ADT for disjoint sets
- Applications of union-find

Next lecture:

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

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

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)

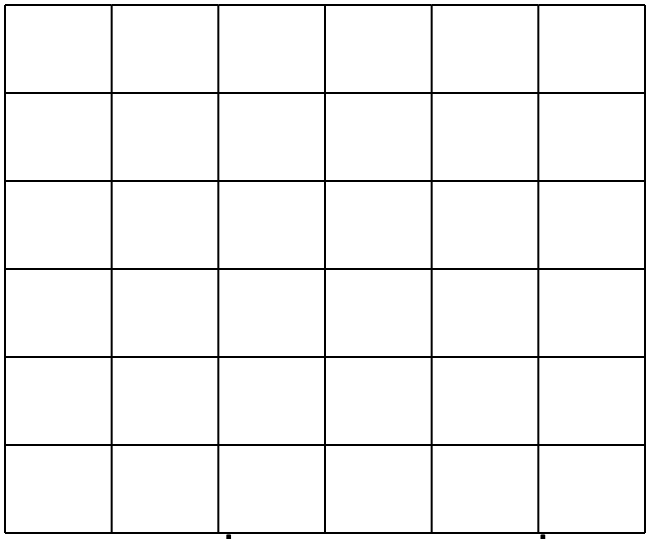
  {1}, {2, 4, 5, 6, 7}, {3}, {8}, {9}

No other operations

- All that can "happen" is sets get unioned
  - No "un-union" or "create new set" or …
- As always: trade-offs – implementations will exploit this small ADT
  - ideas?
- Surprisingly useful ADT: list of applications after one example
  - But not as common as dictionaries or priority queues

Example application: maze-building

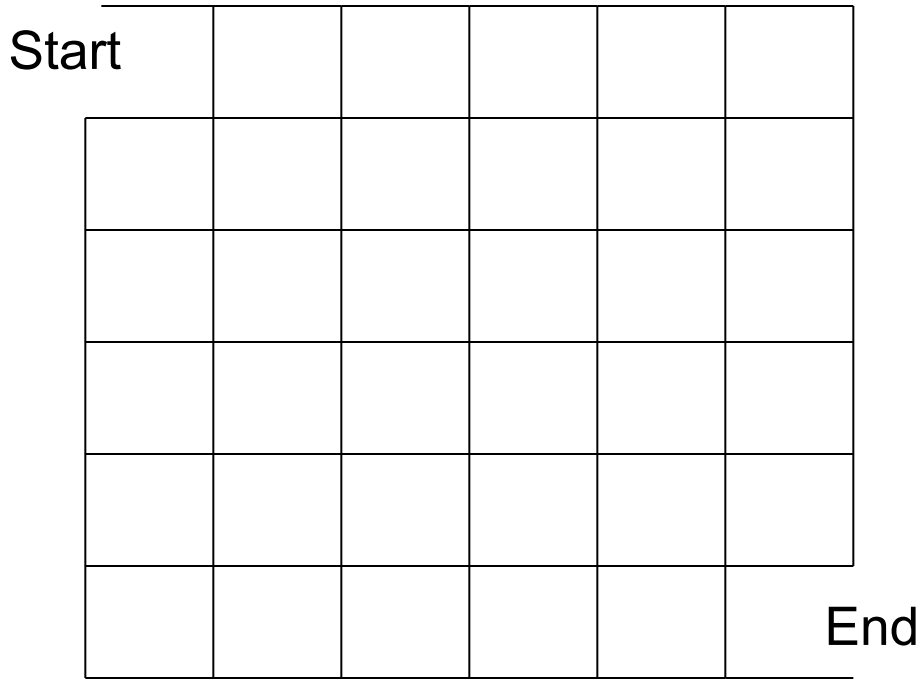
- Build a random maze by erasing edges



  - Possible to get from anywhere to anywhere
    - Including "start" to "finish"
  - No loops possible without backtracking
    - After a "bad turn" have to "undo"

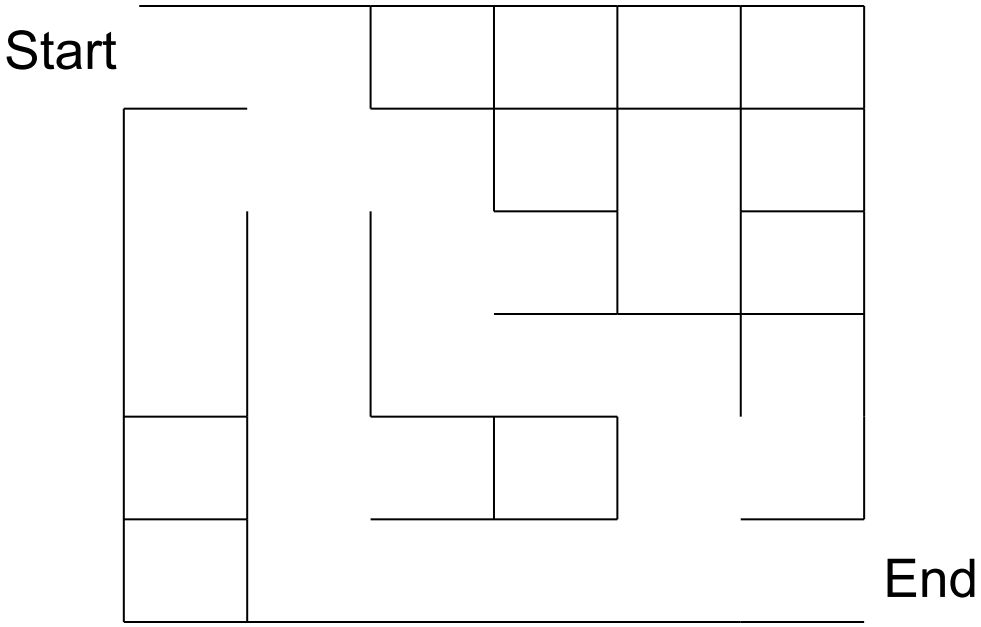
Maze building

Pick start edge and end edge



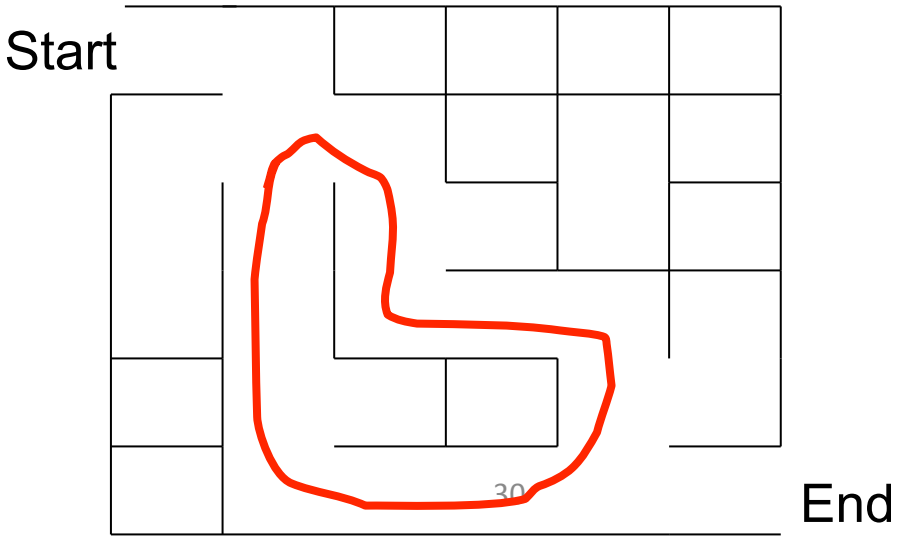
Repeatedly pick random edges to delete

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



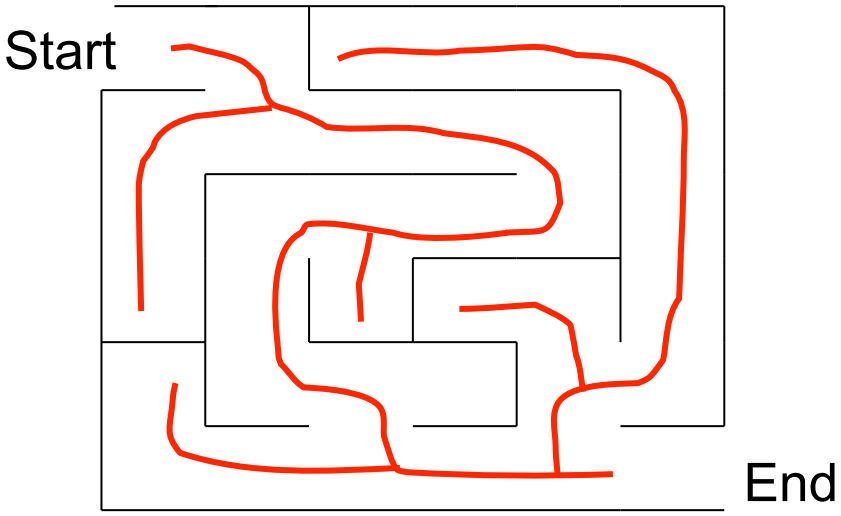
Problems with this approach

1. How can you tell when there is a path from start to finish?
   - We do not really have an algorithm yet
2. We have *cycles*, which a "good" maze avoids
   - Want one solution and no cycles



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

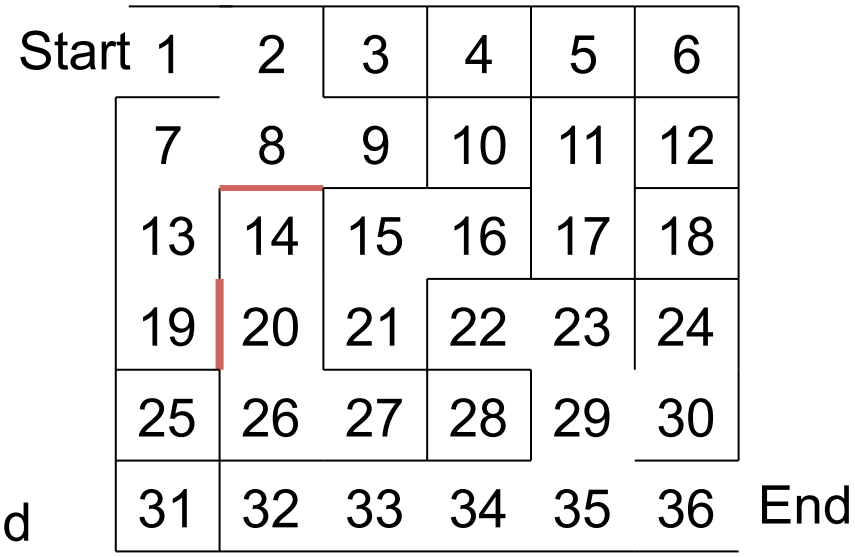
The trick

- 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

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 algorithm

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

while P has more than one set { // stop when possible to get anywhere

- Pick a random edge (x,y) to remove from E
- u = **`find`**(x)
- v = **`find`**(y)
- if u==v

  then add (x,y) to M // same subset, do not create cycle

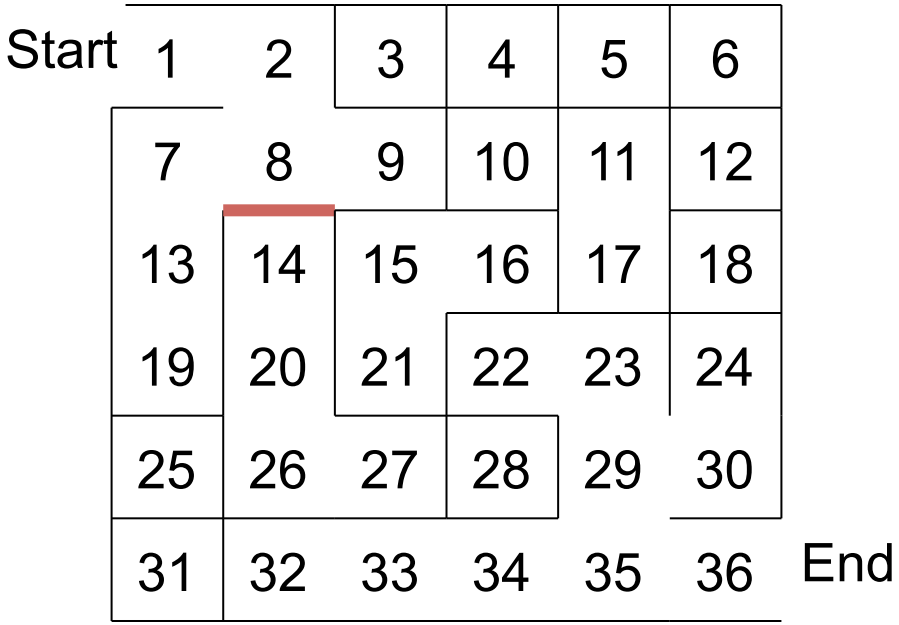
  else **`union`**(u,v) // do not put edge in M, connect subsets

}

Add remaining members of E to M, then output M as the maze

Example 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 step

P
{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}

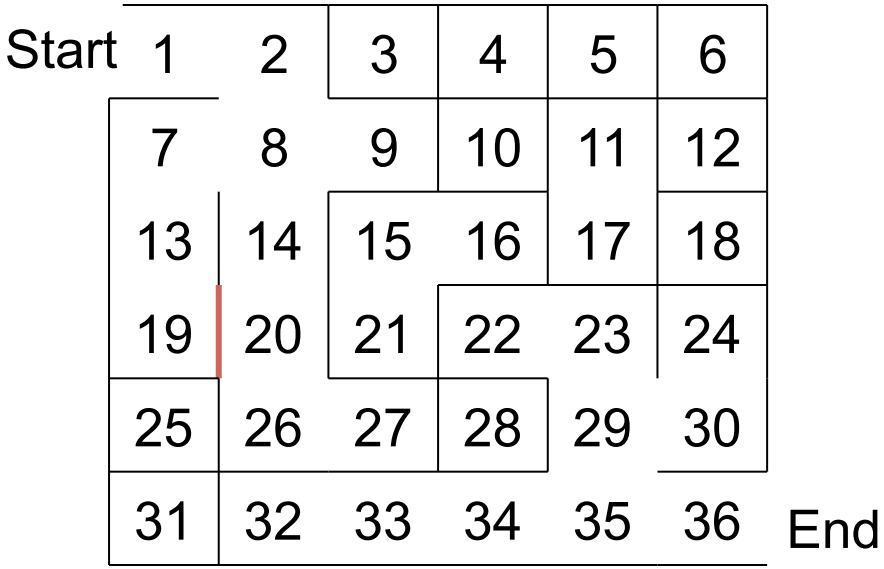
Find(8) = 7
Find(14) = 20

Union(7,20)

P
{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 M step

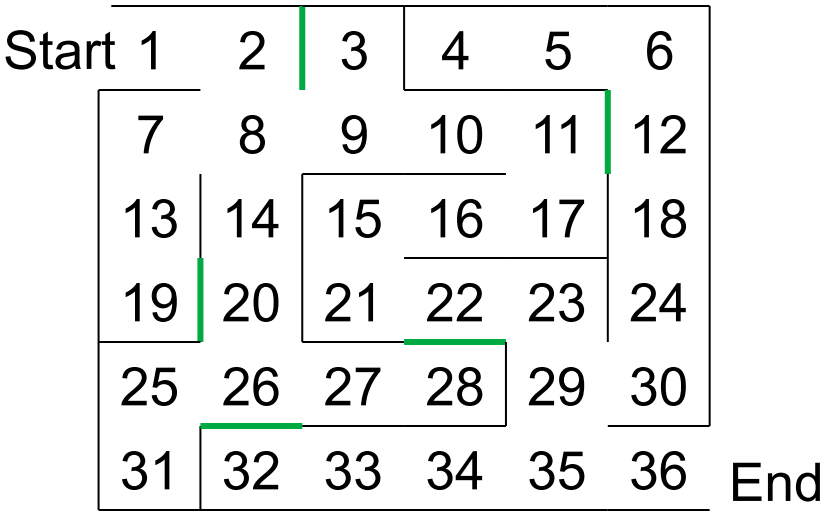
Pick (19,20)



P
{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 P has one set
- Suppose green edges are already in M and black edges were not yet picked
  - Add all black edges to M



P
{1,2,3,4,5,6,7,… 36}

Other applications

- Maze-building is:
  - Cute
  - A surprising use of the union-find ADT
- Many other uses (which is why an ADT taught in CSE373):
  - 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
- Not as common as dictionaries, queues, and stacks, but valuable because implementations are very fast, so when applicable can provide big improvements