CSE 326 Data Structures

Dave Bacon

Hashing / Disjoint Sets

Logistics

- Homework 5 on web due Friday
- Project 3 out soon
- Reading: Weiss Chapter 8

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

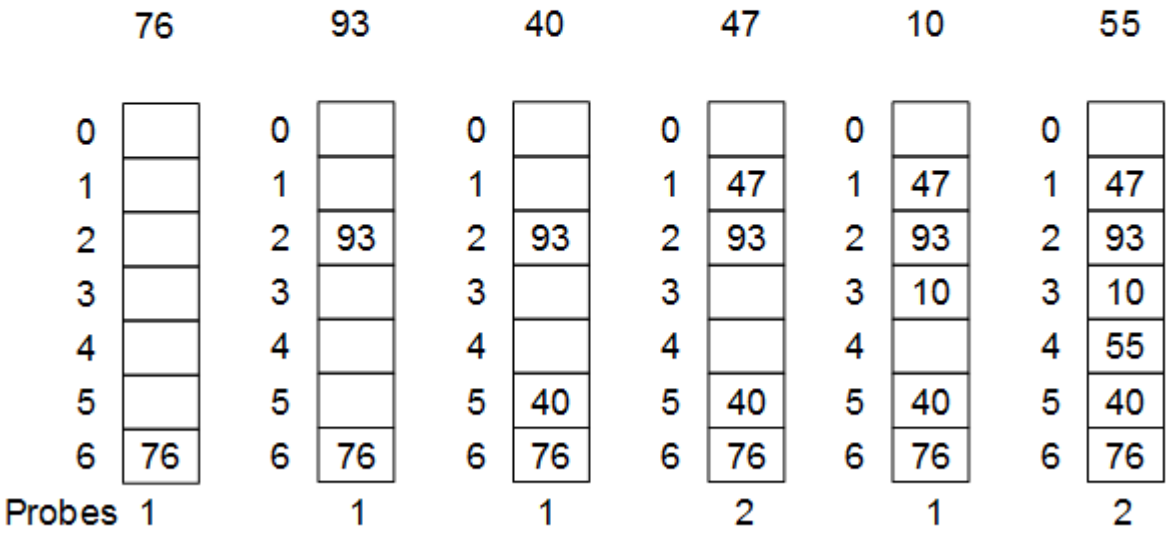
  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

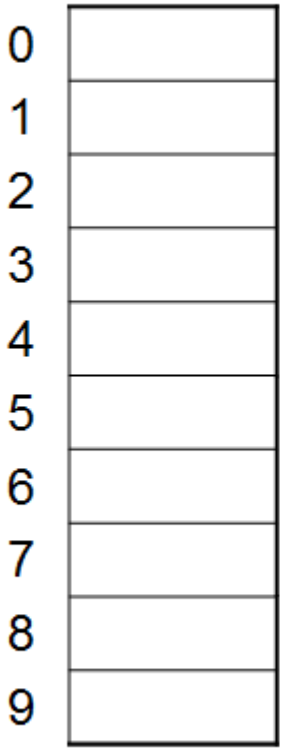
  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

$h(k) = k \bmod 7$ and $g(k) = 5 - (k \bmod 5)$



Resolving Collisions with Double Hashing



Hash Functions:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

M =

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity. (cost of doubling table and rehashing is amortized over many inserts)

Disjoint Sets

Chapter 8

Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union

- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)

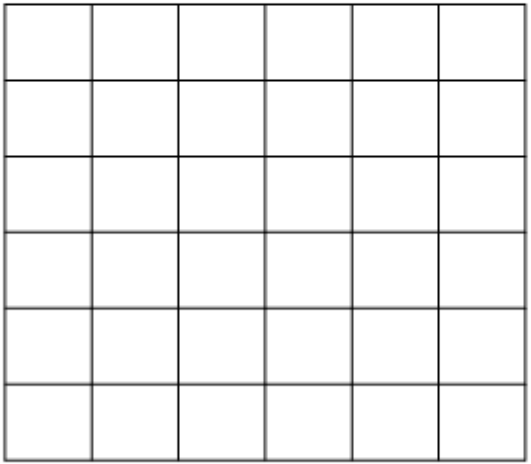
    {3,5,7,1,6}, {4,2,8}, {9},

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

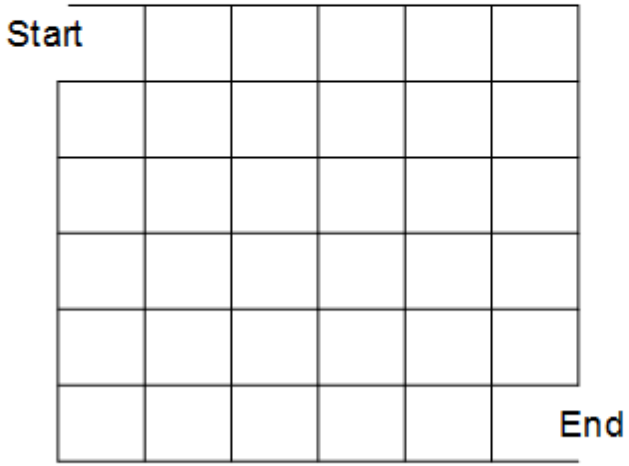
Building Mazes

- Build a random maze by erasing edges.



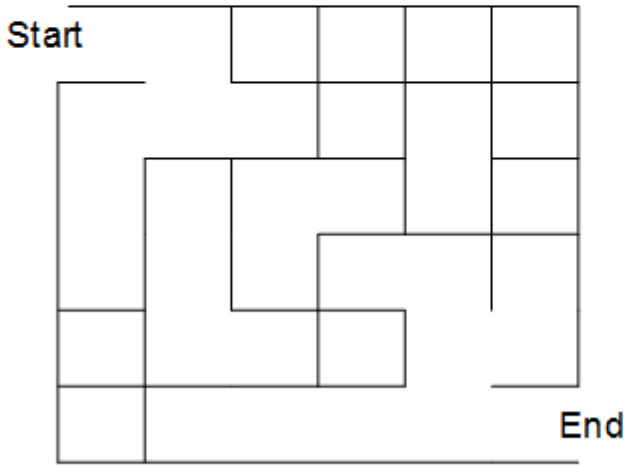
Building Mazes (2)

- Pick Start and End



Building Mazes (3)

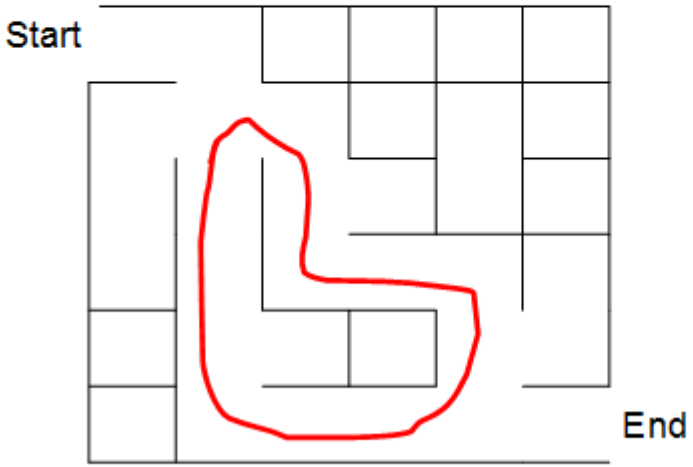
- Repeatedly pick random edges to delete.



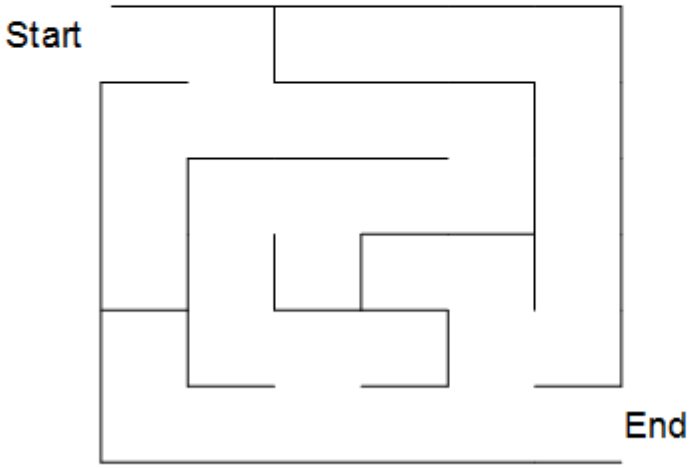
Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- Only one path from any one cell to another (There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.)

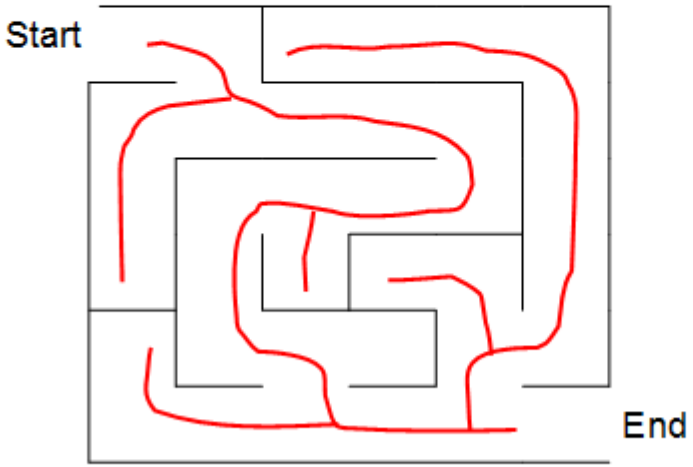
A Cycle



A Good Solution



A Hidden Tree



Number the Cells

We have disjoint sets S ={ {1}, {2}, {3}, {4},… {36} } each cell is unto itself.
We have all possible edges E ={ (1,2), (1,7), (2,8), (2,3), … } 60 edges total.

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

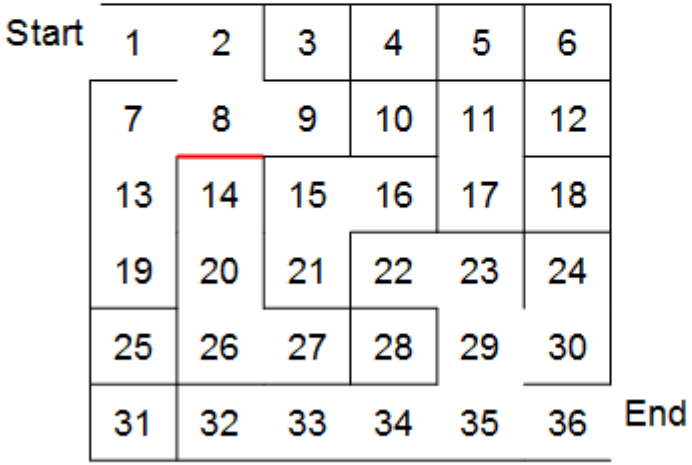
Basic Algorithm

- S = set of sets of connected cells
- E = set of edges
- Maze = set of maze edges (initially empty)

```
While there is more than one set in S {
   pick a random edge (x,y) and remove from E
   u := Find(x);
   v := Find(y);
   if u ≠ v then   // removing edge (x,y) connects previously non-
                   // connected cells x and y - leave this edge removed!
      Union(u,v)
   else            // cells x and y were already connected, add this
                   // edge to set of edges that will make up final maze.
       add (x,y) to Maze
}
All remaining members of E together with Maze form the maze
```

Example Step

Pick (8,14)



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

Example

S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

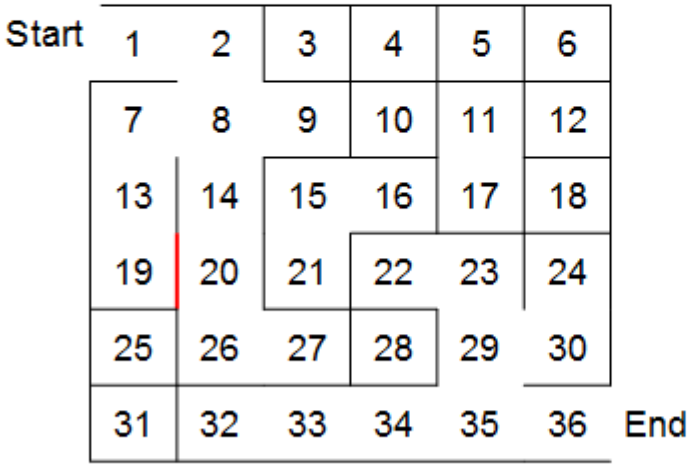
Find(8) = 7
Find(14) = 20

Union(7,20)

S
{1,2,7,8,9,13,19,14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

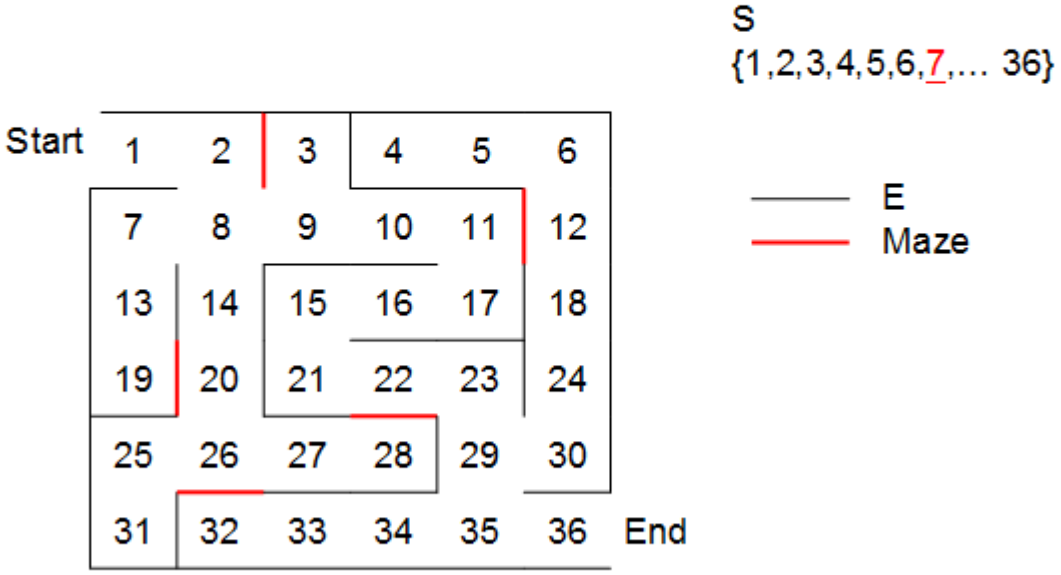
Example

Pick (19,20)



S
{1,2,7,8,9,13,19
14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example at the End



Implementing the DS ADT

can there be more unions?

- n elements,
  Total Cost of: m finds, $\leq n$-1 unions

- Target complexity: $O(m+n)$
  i.e. $O(1)$ amortized

- $O(1)$ worst-case for find as well as union would be great, but…

  Known result: both find and union *cannot* be done in worst-case $O(1)$ time

Attempt #1

- Hash elements to a hashtable
- Store set identifier for each element as data

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

Attempt #2

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

Attempt #3

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together
- Always update identifiers of *smaller* set

runtime for find:

runtime for union:

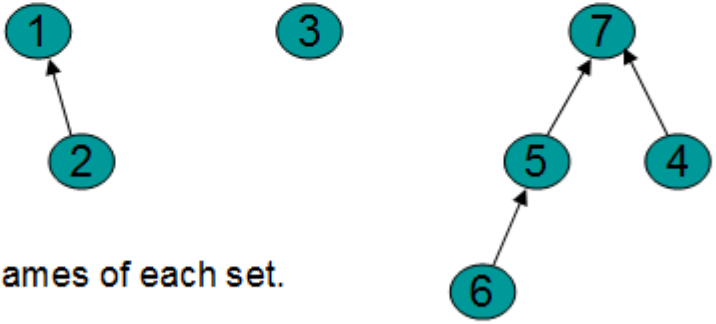
runtime for m finds, n-1 unions:

[Read section 8.2]

Up-Tree for Disjoint Union/Find

Initial state:

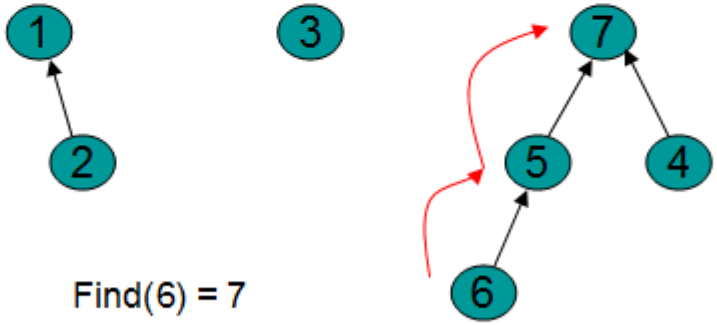
After several Unions:



Roots are the names of each set.

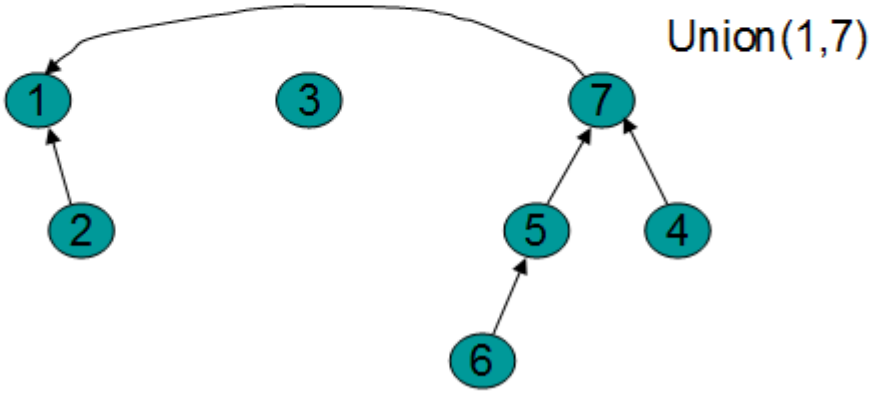
Find Operation

Find(x) - follow x to the root and return the root



Union Operation

Union(x,y) - assuming x and y are roots, point y to x.

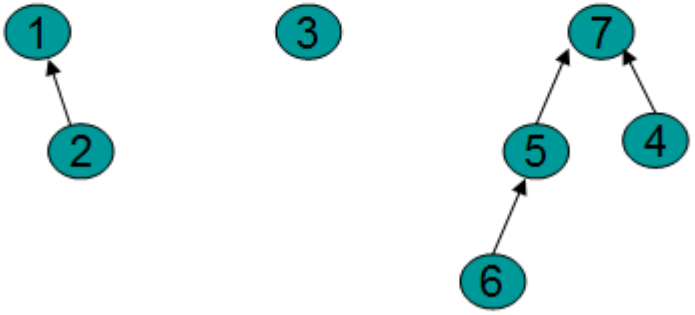


Simple Implementation

- Array of indices

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

Up[x] = 0 means
x is a root.



Implementation

```
int Find(int x) {

  while(up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

```
void Union(int x, int y) {
  up[y] = x;
}
```

runtime for Union():

runtime for Find():

runtime for m Finds and n-1 Unions:

Find Solutions

Recursive

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
if up[x] = 0 then return x
else return Find(up,up[x]);
}
```

Iterative

```
Find(up[] : integer array, x : integer) : integer {
//precondition: x is in the range 1 to size//
while up[x] ≠ 0 do
  x := up[x];
return x;
}
```

Now this doesn't look good ☹

Can we do better? *Yes!*

1. Improve union so that *find* only takes $\Theta(\log n)$
   - Union-by-size
   - Reduces complexity to $\Theta(m \log n + n)$

2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to <u>almost</u> $\Theta(m + n)$