Hash Tables (continued) & Disjoint Sets

Chapter 5 & 8 in Weiss

1

Quadratic Probing

$f(i) = i^2$

Less likely to encounter Primary Clustering

- Probe sequence:
  - 0^{th} probe = h(k) mod TableSize
  - 1^{th} probe = (h(k) + 1) mod TableSize
  - 2^{th} probe = (h(k) + 4) mod TableSize
  - 3^{th} probe = (h(k) + 9) mod TableSize
  - . . .
  - i^{th} probe = (h(k) + i^2) mod TableSize

2

Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

3

Quadratic Probing Example

	insert(76) 76%7 = 6	insert(40) 40%7 = 5	insert(48) 48%7 = 6	insert(5) 5%7 = 5	insert(55) 55%7 = 6	But… insert(47) 47%7 = 5
0			48	48	48	48
1						
2				5	5	5
3					55	55
4						
5		40	40	40	40	40
6	76	76	76	76	76	76

4

Quadratic Probing: Success guarantee for $\lambda < ½$

- If size is prime and $\lambda < ½$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all `0 ≤ i,j ≤ size/2` and `i ≠ j`
    `(h(x) + i²) mod size ≠ (h(x) + j²) mod size`
  - by contradiction: suppose that for some i ≠ j:
    `(h(x) + i²) mod size = (h(x) + j²) mod size`
    `⇒ i² mod size = j² mod size`
    `⇒ (i² - j²) mod size = 0`
    `⇒ [(i + j)(i - j)] mod size = 0`

  BUT size does not divide `(i-j)` or `(i+j)`

5

Quadratic Probing: Properties

- For *any* $\lambda < ½$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad
- But what about keys that hash to the same *spot*?
  - ***Secondary Clustering!***

6

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 mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Resolving Collisions with Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions:
H(K) = K mod 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}

13

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},

14

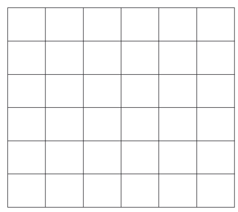
Find

- 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

15

Building Mazes

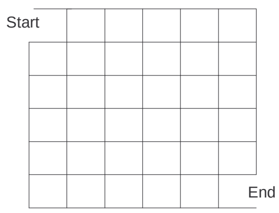
- Build a random maze by erasing edges.



16

Building Mazes (2)

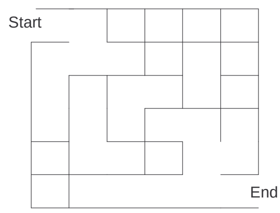
- Pick Start and End



17

Building Mazes (3)

- Repeatedly pick random edges to delete.



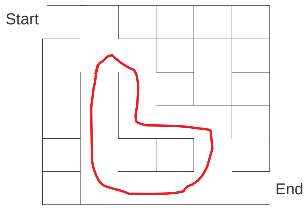
18

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.)

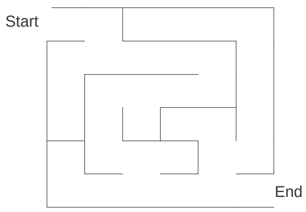
19

A Cycle



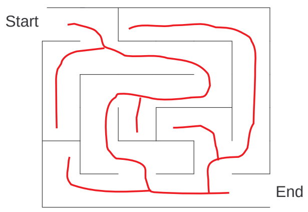
20

A Good Solution



21

A Hidden Tree



22

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

23

Basic Algorithm

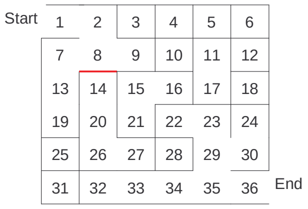
- S = set of sets of connected cells
- E = set of edges
- Maze = set of maze edges (initially empty)

```
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
```

24

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}

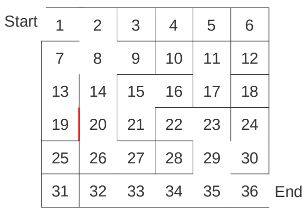
Find(8) = 7
Find(14) = 20

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

S
{1,2,3,4,5,6,7,… 36}