Goal

- Develop a structure that will allow user to insert/delete/find records in
  constant average time
  - structure will be a table (relatively small)
  - table completely contained in memory
  - implemented by an array
  - capitalizes on ability to access any element of the array in *constant* time

Hash Function

- Determines position of key in the array.
- Assume table (array) size is N
- Function $f(x)$ maps any key x to an int between 0 and $N-1$
- For example, assume that $N=15$, that key x is a non-negative integer between 0 and MAX_INT, and hash function $f(x) = x \% 15$.
- (Hash functions for strings aggregate the character values --- see Weiss §5.2.)

Hash Function

Let *f(x)* = *x* % 15. Then,

if *x* =	25	129	35	2501	47	36
f(x) =	10	9	5	11	2	6

Storing the keys in the array is straightforward:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
_	_	47	_	_	35	36	_	_	129	25	2501	_	_	_

Thus, *delete* and *find* can be done in O(1), and also *insert*, except…

Hash Function

What happens when you try to insert: $x = 65$?

$$x = 65$$

$$f(x) = 5$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
_	_	47	_	_	35 65(?)	36	_	_	129	25	2501	_	_	_

This is called a *collision*.

Handling Collisions

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Handling Collisions

Separate Chaining

Separate Chaining

Let each array element be the head of a chain.

```
0   1   2   3   4   5   6   7   8   9   10   11  12  13  14
        ↓               ↓   ↓               ↓   ↓    ↓
        47              65  36            129  25  2501
                        ↓
                        35
```

Where would you store: 29, 16, 14, 99, 127 ?

Separate Chaining

Let each array element be the head of a chain:

Where would you store: 29, 16, 14, 99, 127 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	↓	↓			↓	↓	↓		↓	↓	↓			↓
	16	47			65	36	127		99	25	2501			14
					↓				↓					↓
					35				129					29

New keys go at the front of the relevant chain.

Separate Chaining: Disadvantages

- Parts of the array might never be used.
- As chains get longer, search time increases to O(n) in the worst case.
- Constructing new chain nodes is relatively expensive (still constant time, but the constant is high).
- Is there a way to use the “unused” space in the array instead of using chains to make more space?

Handling Collisions

Linear Probing

Linear Probing

Let key x be stored in element $f(x)=t$ of the array

```
0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
         47                 35   36             129   25 2501
                            65(?)
```

What do you do in case of a collision?

If the hash table is *not full*, attempt to store key in the next array element (in this case $(t+1)\%N$, $(t+2)\%N$, $(t+3)\%N$ ...)

until you find an empty slot.

Linear Probing

Where do you store *65* ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			
					↑	↑	↑							

attempts

Where would you store: 29?

Linear Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1)\%N$, $(t+2)\%N$, ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			29
														↑ attempts

Where would you store: 16?

Linear Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1)\%N$, $(t+2)\%N$, ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65		129	25	2501			29
	↑													

Where would you store: 14?

Linear Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1)\%N$, $(t+2)\%N$, ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501			29
↑														↑
														attempts

Where would you store: 99?

Linear Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1)\%N$, $(t+2)\%N$, ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29
									↑	↑	↑	↑		

attempts

Where would you store: 127 ?

Linear Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1)\%N$, $(t+2)\%N$, ...

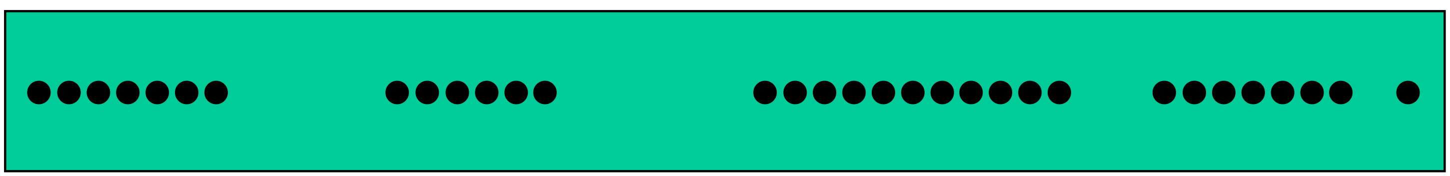
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65	127	129	25	2501	29	99	14
							↑	↑						

attempts

Linear Probing

- Eliminates need for separate data structures (chains), and the cost of constructing nodes.

- Leads to problem of clustering. Elements tend to *cluster* in dense intervals in the array.



- Search efficiency problem remains.
- Deletion becomes trickier….

Deletion problem

- H=KEY MOD 10
- Insert 47, 57, 68, 18, 67
- Find 68
- Find 10
- Delete 47
- Find 57

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Deletion Problem -- SOLUTION

- "Lazy" deletion

- Each cell is in one of 3 possible states:
  - active
  - empty
  - deleted

- For Find or Delete
  - only stop search when EMPTY state detected (not DELETED)

Deletion-Aware Algorithms

- Insert
  - Cell empty or deleted — insert at H, *cell = active*
  - Cell active — H = (H + 1) mod TS

- Find
  - cell empty — NOT found
  - cell deleted — H = (H + 1) mod TS
  - cell active — if key == key in cell -> FOUND
    else H = (H + 1) mod TS

- Delete
  - cell active; key != key in cell — H = (H + 1) mod TS
  - cell active; key == key in cell — DELETE; *cell=deleted*
  - cell deleted — H = (H + 1) mod TS
  - cell empty — NOT found

Handling Collisions

Quadratic Probing

Quadratic Probing

Let key x be stored in element $f(x)=t$ of the array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			
					65(?)									

What do you do in case of a collision?

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$, $(t+3^2)\%N$ ...

until you find an empty slot.

Quadratic Probing

Where do you store *65* ? *f(65)=t=5*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			65
					t	t+1			t+4					t+9

attempts

Where would you store: 29?

Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

```
 0    1    2    3    4    5    6    7    8    9   10    11   12   13   14
29        47                 35   36                129   25  2501            65
 ↑                                                                              ↑
t+1                                                                             t
                                                                         attempts
```

Where would you store: 16?

Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47			35	36			129	25	2501			65

```
      ↑
      t
 attempts
```

Where would you store: 14?

Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47	14		35	36			129	25	2501			65
↑			↑											↑
t+1			t+4											t attempts

Where would you store: 99?

Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47	14		35	36			129	25	2501		99	65
									↑	↑			↑	
									t	t+1			t+4	

attempts

Where would you store: 127 ?

Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

Where would you store: 127 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	47	14		35	36	127		129	25	2501		99	65

↑
t
attempts

Quadratic Probing

- Tends to distribute keys better than linear probing
- Alleviates problem of clustering
- Runs the risk of an infinite loop on insertion, unless precautions are taken.
- E.g., consider inserting the key 16 into a table of size 16, with positions 0, 1, 4 and 9 already occupied.
- Therefore, table size should be prime.

Handling Collisions

Double Hashing

Double Hashing

- Use a hash function for the decrement value
  - $Hash(key, i) = H_1(key) – (H_2(key) * i)$
- Now the decrement is a function of the key
  - The slots visited by the hash function will vary even if the initial slot was the same
  - Avoids clustering
- Theoretically interesting, but in practice slower than quadratic probing, because of the need to evaluate a second hash function.

Double Hashing

Let key x be stored in element $f(x)=t$ of the array

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			
					65(?)									

What do you do in case of a collision?

Define a second hash function $f_2(x)=d$. Attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$, $(t+3d)\%N$ ...

until you find an empty slot.

Double Hashing

- Typical second hash function

$$f_2(x) = R - (x \% R)$$

where R is a prime number, $R < N$

Double Hashing

Where do you store *65* ? $f(65)=t=5$

Let $f_2(x) = 11 - (x \% 11)$ $f_2(65)=d=1$

Note: $R=11$, $N=15$

Attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$, $(t+3d)\%N$ ...

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			
					↑	↑	↑							
					t	t+1	t+2							

attempts

Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+d)\%N$ ...

Let $f_2(x) = 11 - (x \% 11)$ $f_2(29)=d=4$

Where would you store: 29?

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			29
														↑
														t
														attempt

Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+d)\%N$ ...

Let $f_2(x) = 11 - (x \% 11)$ $\quad$ $f_2(16)=d=6$

Where would you store: 16?

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65		129	25	2501			29
	↑													
	t													
	attempt													

Where would you store: 14?

Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+d)\%N$ ...

Let $f_2(x) = 11 - (x \% 11)$ $f_2(14)=d=8$

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501			29
↑							↑							↑
t+16 attempts							t+8							t

Where would you store: 99?

Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+d)\%N$ ...

Let $f_2(x) = 11 - (x \% 11)$ $\quad f_2(99)=d=11$

Array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29
	↑				↑				↑			↑		
	t+22				t+11				t			t+33		

attempts

Where would you store: 127 ?

Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+d)\%N$ ...

Let $f_2(x) = 11 - (x \% 11)$ $\quad f_2(127)=d=5$

Array:

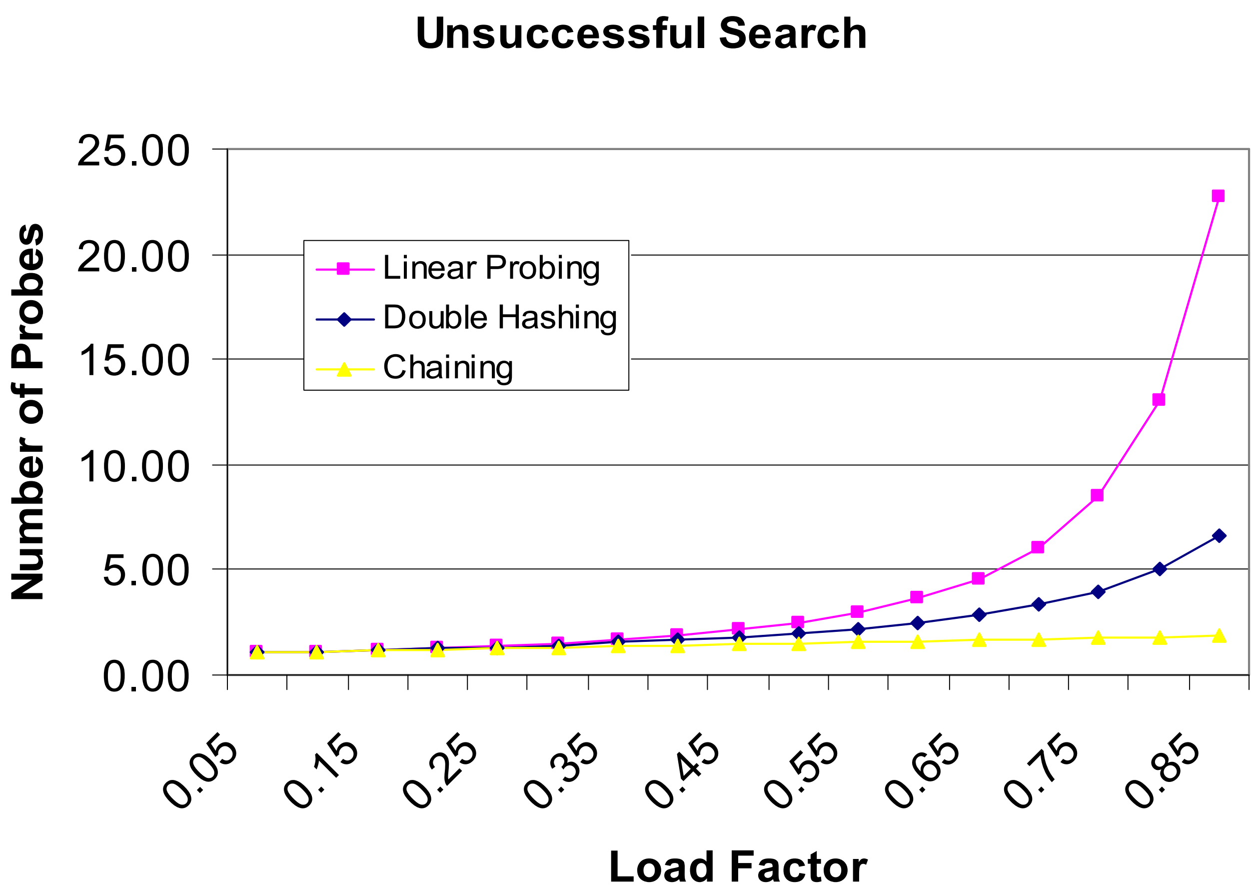
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29
		↑					↑					↑		
		t+10					t					t+5		

attempts

Infinite loop!

Performance

Load factor = % of table that's occupied.



REHASHING

- When the load factor exceeds a threshold, double the table size (smallest prime > 2 * old table size).
- Rehash each record in the old table into the new table.
- Expensive: O(N) work done in copying.
- However, if the threshold is large (e.g., ½), then we need to rehash only once per O(N) insertions, so the cost is "amortized" constant-time.

Factors affecting efficiency

- – Choice of hash function
- – Collision resolution strategy
- – Load Factor

- Hashing offers excellent performance for insertion and retrieval of data.

Comparison of Hash Table & BST

	BST	HashTable
Average Speed	$O(\log_2 N)$	$O(1)$
Find Min/Max	Yes	No
Items in a range	Yes	No
Sorted Input	Very Bad	No problems

Use HashTable if there is any suspicion of SORTED input & NO ordering information is required.