CSE 373: Data Structures and Algorithms

Lecture 15: Hashing II

Set implementation: insert

- Similar structure to `contains`
  - Calculate hash of new element
  - Check if the element is already in the set

- Add the element to the front of the list that is at `table[hash(value)]`

Set implementation: insert

```
public boolean add(String value) {
    int valuePosition = hash(value);

    // check to see if the value is already in the set
    StringHashEntry temp = table[valuePosition];
    while (temp != null) {
        if (temp.data.equals(value)) {
            return false;
        }
        temp = temp.next;
    }

    // add the value to the set
    StringHashEntry newEntry = new StringHashEntry(value, table[valuePosition]);
    table[valuePosition] = newEntry;
    size++;
    return true;
}
```

Set implementation: remove

```
public boolean remove(String value) {
    int valuePosition = hash(value);
    if (table[valuePosition] == null) {              // empty bucket
        return false;
    }
    if (table[valuePosition].data.equals(value)) {   // removing front
        table[valuePosition] = table[valuePosition].next;
        size--;  return true;
    }
    StringHashEntry temp = table[valuePosition];
    while (temp.next != null) {                      // find value
        if (temp.next.data.equals(value)) {
            temp.next = temp.next.next;
            size--;  return true;
        }
        temp = temp.next;
    }
    return false;
}
```

Hash versus tree

- Which is better, a hash set or a tree set?

Hash	Tree

Implementing Set ADT (Revisited)

	Insert	Remove	Search
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(\log(n)+n)$	$\Theta(\log(n) + n)$	$\Theta(\log(n))$
Linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
BST (if balanced)	Θ(log n)	Θ(log n)	Θ(log n)
Hash table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Probing hash tables

- Alternative strategy for collision resolution: try alternative cells until empty cell found
  - cells $h_0(x)$, $h_1(x)$, $h_2(x)$, ... tried in succession, where $h_i(x) = (hash(x) + f(i)) \% TableSize$
  - f is collision resolution strategy
  - bigger table needed

Linear probing

- **linear probing**: resolving collisions in slot *i* by putting the colliding element into the next available slot (*i*+1, *i*+2, ...)
  - add 41, 34, 7, 18, then 21, then 57
    - 21 collides (41 is already there), so we search ahead until we find empty slot 2
    - 57 collides (7 is already there), so we search ahead twice until we find empty slot 9
  - lookup algorithm becomes slightly modified; we have to loop now until we find the element or an empty slot
    - what happens when the table gets mostly full?

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Linear probing

- $f(i) = i$
- Probe sequence:

  0^{th} probe = $h(x)$ mod *TableSize*

  1^{th} probe = $(h(x) + 1)$ mod *TableSize*

  2^{th} probe = $(h(x) + 2)$ mod *TableSize*

  . . .

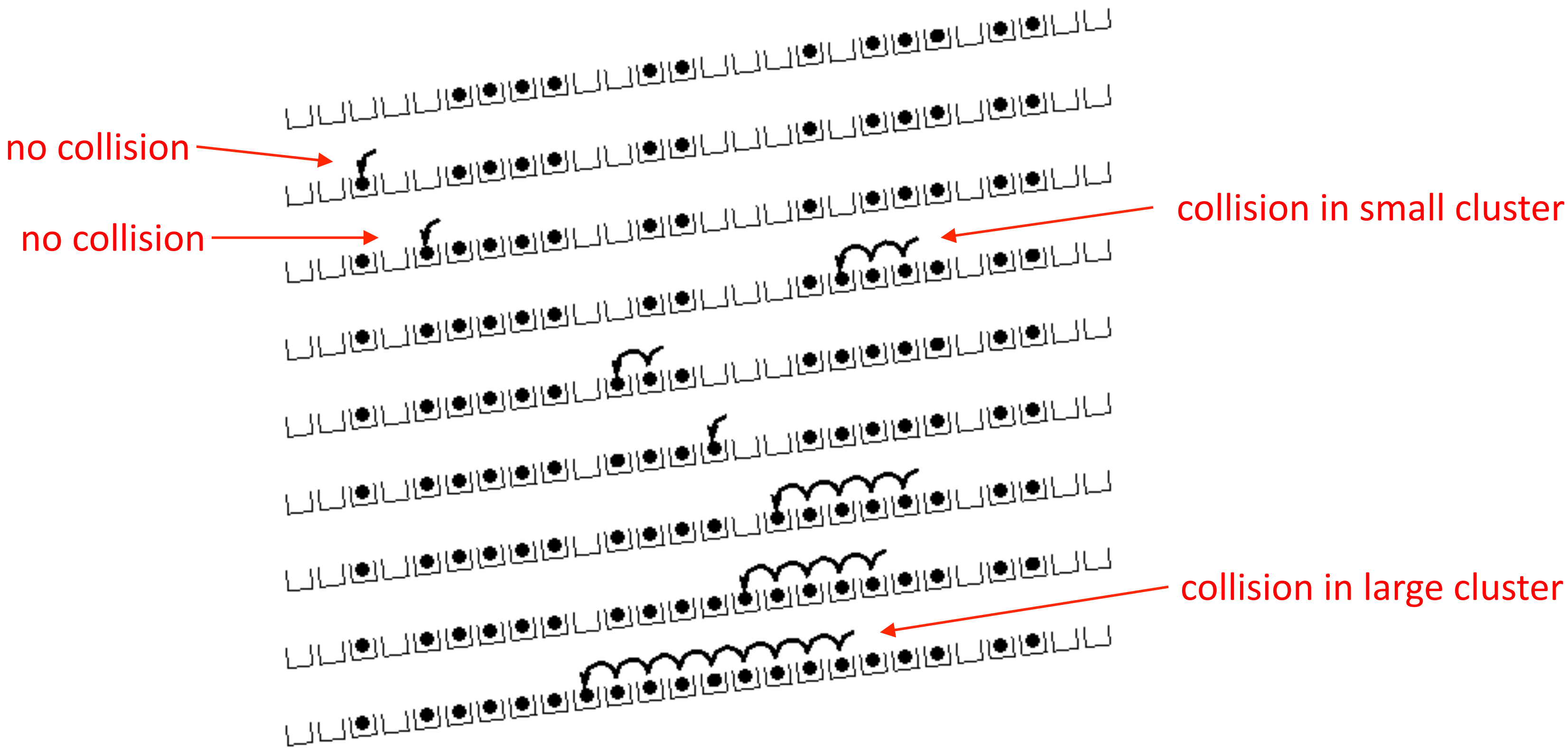
  i^{th} probe = $(h(x) + i)$ mod *TableSize*

Primary clustering problem

- **clustering**: nodes being placed close together by probing, which degrades hash table's performance
  - add 89, 18, 49, 58, 9
  - now searching for the value 28 will have to check half the hash table! no longer constant time...

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Linear probing – clustering



Alternative probing strategy

- Primary clustering occurs with linear probing because the same linear pattern:
  - if a slot is inside a cluster, then the next slot must either:
    - also be in that cluster, or
    - expand the cluster

- Instead of searching forward in a linear fashion, consider searching forward using a quadratic function

Quadratic probing

- **quadratic probing**: resolving collisions on slot *i* by putting the colliding element into slot *i*+1, *i*+4, *i*+9, *i*+16, ...
  - add 89, 18, 49, 58, 9
    - 49 collides (89 is already there), so we search ahead by +1 to empty slot 0
    - 58 collides (18 is already there), so we search ahead by +1 to occupied slot 9, then +4 to empty slot 2
    - 9 collides (89 is already there), so we search ahead by +1 to occupied slot 0, then +4 to empty slot 3
  - what is the lookup algorithm?

0	49
1	
2	58
3	9
4	
5	
6	
7	
8	18
9	89

Quadratic probing in action

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Quadratic probing

- $f(i) = i^2$

- Probe sequence:

  0^{th} probe = $h(x)$ mod *TableSize*

  1^{th} probe = $(h(x) + 1)$ mod *TableSize*

  2^{th} probe = $(h(x) + 4)$ mod *TableSize*

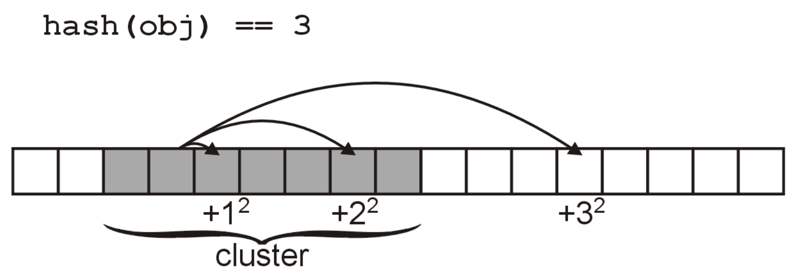
  3^{th} probe = $(h(x) + 9)$ mod *TableSize*

  . . .

  i^{th} probe = $(h(x) + i^2)$ mod *TableSize*

Quadratic probing benefit

- If one of $h + i^2$ falls into a cluster, this does not imply the next one will



- For example, suppose an element was to be inserted in bucket 23 in a hash table with 31 buckets
  - The sequence in which the buckets would be checked is:

23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic probing benefit

- Even if two buckets are initially close, the sequence in which subsequent buckets are checked varies greatly
  - Again, with *TableSize* = 31, compare the first 16 buckets which are checked starting with elements 22 and 23:

22 22, 23, 26, 0, 7, 16, 27, 9, 24, 10, 29, 19, 11, 5, 1, 30

23 23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

- Quadratic probing solves the problem of primary clustering

Quadratic probing drawbacks

- Suppose we have 8 buckets:

  $1^2 \% 8 = 1$, $2^2 \% 8 = 4$, $3^2 \% 8 = 1$

  – In this case, we are checking bucket $h(x) + 1$ twice having checked only one other bucket

- No guarantee that

  $(h(x) + i^2) \% \mathit{TableSize}$

  will cycle through 0, 1, ..., $\mathit{TableSize} - 1$

Quadratic probing

- Solution:
  - require that *TableSize* be prime
  - $(h(x) + i^2)$ % *TableSize* for $i = 0, ..., (TableSize - 1)/2$ will cycle through $(TableSize + 1)/2$ values before repeating

- Example with M = 11:

  0, 1, 4, 9, 16 ≡ 5, 25 ≡ 3, 36 ≡ 3
- With M = 13:

  0, 1, 4, 9, 16 ≡ 3, 25 ≡ 12, 36 ≡ 10, 49 ≡ 10
- With M = 17:

  0, 1, 4, 9, 16, 25 ≡ 8, 36 ≡ 2, 49 ≡ 15, 64 ≡ 13, 81 ≡ 13

Note: the symbol ≡ means "% M ="

Double hashing

- **double hashing**: resolve collisions on slot *i* by applying a second hash function

- $f(i) = i * g(x)$
  where *g* is a second hash function
  - limitations on what *g* can evaluate to?
  - recommended: $g(x) = R - (x \% R)$, where *R* prime smaller than *TableSize*

- Probe sequence:

  0^{th} probe = $h(x) \% TableSize$

  1^{th} probe = $(h(x) + g(x)) \% TableSize$

  2^{th} probe = $(h(x) + 2*g(x)) \% TableSize$

  3^{th} probe = $(h(x) + 3*g(x)) \% TableSize$

  . . .

  i^{th} probe = $(h(\underline{x}) + i*g(\underline{x})) \% TableSize$

Double Hashing Example

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

	41	16	40	47	10	55
0						
1				47	47	47
2		16	16	16	16	16
3					10	10
4						55
5			40	40	40	40
6	41	41	41	41	41	41
Probes	1	1	1	2	1	2