# Hash versus tree

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

<table>
<thead>
<tr>
<th>Hash</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Implementing Set ADT (Revisited)

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Remove</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted array</strong></td>
<td><strong>O(1)</strong></td>
<td><strong>O(n)</strong></td>
<td><strong>O(n)</strong></td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td><strong>O(log n + n)</strong></td>
<td><strong>O(log n + n)</strong></td>
<td><strong>O(log n)</strong></td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td><strong>O(1)</strong></td>
<td><strong>O(n)</strong></td>
<td><strong>O(n)</strong></td>
</tr>
<tr>
<td><strong>BST (if balanced)</strong></td>
<td><strong>O(log n)</strong></td>
<td><strong>O(log n)</strong></td>
<td><strong>O(log n)</strong></td>
</tr>
<tr>
<td><strong>Hash table</strong></td>
<td><strong>O(1)</strong></td>
<td><strong>O(1)</strong></td>
<td><strong>O(1)</strong></td>
</tr>
</tbody>
</table>
Probing hash tables

- Alternative strategy for collision resolution: try alternative cells until empty cell found
  - Cells $h_0(x), h_1(x), h_2(x), \ldots$ tried in succession, where:
    - $h_i(x) = (\text{hash}(x) + f(i)) \mod \text{TableSize}$
  - $f$ is collision resolution strategy
  - Because all data goes in table, bigger table needed
Linear probing

- **Linear probing**: resolve collisions in slot \( i \) by putting colliding element into next available slot \((i+1, i+2, \ldots)\)

Pseudocode for insert:

```
first probe = h(value)
while (table[probe] occupied)
    probe = (probe + 1) % TableSize
table[probe] = value
```

- add 41, 34, 7, 18, then 21, then 57

- lookup/search algorithm modified - have to loop until we find the element or an empty slot
  - What happens when the table gets mostly full?
Linear probing

- $f(i) = i$
- Probe sequence:
  - $0^{th}$ probe = $h(x) \mod TableSize$
  - $1^{st}$ probe = $(h(x) + 1) \mod TableSize$
  - $2^{nd}$ probe = $(h(x) + 2) \mod TableSize$
    
    
    ... 
  - $i^{th}$ probe = $(h(x) + i) \mod TableSize$
Deletion in Linear Probing

- To delete 18, first search for 18
- 18 found in bucket 8
- What happens if we set bucket 8 to null?
  - What will happen when we search for 57?
Deletion in Linear Probing (2)

- Instead of setting bucket 8 to null, place a special marker there

- When lookup encounters marker, it ignores it and continues search
  - What should insert do if it encounters marker?

- Too many markers degrades performance – *rehash* if there are too many

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>41</td>
<td>21</td>
<td></td>
<td>34</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td>57</td>
</tr>
</tbody>
</table>
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...
### Linear probing – clustering

<table>
<thead>
<tr>
<th></th>
<th>Collision in small cluster</th>
<th>Collision in large cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>no collision</td>
<td>no collision</td>
<td>no collision</td>
</tr>
</tbody>
</table>

---

---

---
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?
Quadratic probing in action

\[\text{hash ( 89, 10 )} = 9\]
\[\text{hash ( 18, 10 )} = 8\]
\[\text{hash ( 49, 10 )} = 9\]
\[\text{hash ( 58, 10 )} = 8\]
\[\text{hash ( 9, 10 )} = 9\]

<table>
<thead>
<tr>
<th>After insert 89</th>
<th>After insert 18</th>
<th>After insert 49</th>
<th>After insert 58</th>
<th>After insert 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>49</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>58</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>
Quadratic probing

- $f(i) = i^2$

- Probe sequence:
  
  0th probe = $h(x) \mod TableSize$
  
  1st probe = $(h(x) + 1) \mod TableSize$
  
  2nd probe = $(h(x) + 4) \mod TableSize$
  
  3rd 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
  
  \[ (h(x) + i^2) \% TableSize \]
  
  will cycle through 0, 1, ..., \( TableSize - 1 \)
Quadratic probing

Solution:
- require that TableSize be prime
- \((h(x) + i^2) \% \text{TableSize}\) for \(i = 0, \ldots, (\text{TableSize} - 1)/2\) will cycle through \((\text{TableSize} + 1)/2\) values before repeating

Example with TableSize = 11:
- 0, 1, 4, 9, 16 \equiv 5, 25 \equiv 3, 36 \equiv 3
- With TableSize = 13:
  - 0, 1, 4, 9, 16 \equiv 3, 25 \equiv 12, 36 \equiv 10, 49 \equiv 10
- With TableSize = 17:
  - 0, 1, 4, 9, 16, 25 \equiv 8, 36 \equiv 2, 49 \equiv 15, 64 \equiv 13, 81 \equiv 13

Note: the symbol \(\equiv\) means "\% TableSize"
Hashing practice problem

- Draw a diagram of the state of a hash table of size 10, initially empty, after adding the following elements.
  - \( h(x) = x \mod 10 \) as the hash function.
  - Assume that the hash table uses linear probing.

7, 84, 31, 57, 44, 19, 27, 14, and 64

- Repeat the problem above using quadratic probing.
Double hashing

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

- $f(i) = i \times g(x)$
  - 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 $TableSize$

- **Psuedocode for double hashing:**
  ```python
  if (table is full) error
  probe = h(value)
  offset = g(value)
  while (table[probe] occupied)
      probe = (probe + offset) % TableSize
  table[probe] = value
  ```
Double Hashing Example

\[ h(x) = x \% 7 \text{ and } g(x) = 5 - (x \% 5) \]

<table>
<thead>
<tr>
<th></th>
<th>41</th>
<th>16</th>
<th>40</th>
<th>47</th>
<th>10</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probes</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Probes 1
Double hashing

- $f(i) = i \times g(x)$

- **Probe sequence:**
  - $0^{th}$ probe = $h(x) \% TableSize$
  - $1^{th}$ probe = $(h(x) + g(x)) \% TableSize$
  - $2^{th}$ probe = $(h(x) + 2\times g(x)) \% TableSize$
  - $3^{th}$ probe = $(h(x) + 3\times g(x)) \% TableSize$
  - $\ldots$
  - $i^{th}$ probe = $(h(x) + i\times g(x)) \% TableSize$