Hash Tables (2)
Open Addressing: Linear Probing

**Linear Probing** – place colliding item in the next available location

- if $A[h(k) \mod N]$ is already occupied, then try next $A[(h(k)+1) \mod N]$, $A[(h(k)+2) \mod N]$, etc.
- wrap around from the last to the first bucket array location if necessary

$h(4567) = 4567 \mod 101 = 22$

\[\begin{array}{c}
22 & 7497 \\
23 & 24 \\
\vdots & \vdots \\
\end{array}\]

$h(4567+1) = 4568 \mod 101 = 23$

\[\begin{array}{c}
22 & 7497 \\
23 & 4567 \\
24 & \vdots \\
\end{array}\]

**findElement(k) after Linear Probing** – straightforward – follow the same probe sequence as in conflict resolution until the same key or an empty bucket is found.
### Open Addressing: Linear Probing (cont.)

**removeElement(k) after Linear Probing**

- **find** \((k,e)\) and delete it, making the bucket empty
  
  - **problem:** new empty location could cause another **findElement(k)** to stop prematurely
  
  - **solution:** replace deleted item with a special "deactivated item" object; furthermore

  - (a) modify **findElement(k)** so that it skips over deactivated items and continues probing until reaching the key or an empty bucket

  - (b) modify **insertItem(k)** so that it stops at first empty/deactivated item and replaces it with the new \((k,e)\) pair

---

**Search for** \(h(4567) = 22\) after removal of 7597

<table>
<thead>
<tr>
<th>22</th>
<th>7497</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>4567</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

search for \(h(4567) = 22\) after removal of 7597

<table>
<thead>
<tr>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>22</th>
<th>deact.</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>4567</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Open Addressing: Linear Probing (cont.)

Disadvantages of Linear Probing

1. more complex insert, find, remove methods
2. “primary clustering” phenomenon
   - items tend to cluster together in the bucket array, i.e. one part of the table gets quite dense, even though another part may have relatively few items
   - as clustering gets worse, insertItem(k,e) takes longer because it must step all the way through a cluster to find a vacant bucket
   - as clustering gets worse, findElement(k) takes longer since elements get placed further and further from their correct hashed index
Open Addressing: Linear Probing (cont.)

**Average RT** — in a non-full hash table, assuming no previous removals, the average running time of insert, find, remove is:

\[
\text{RT}_{\text{average}}(n) = \begin{cases} 
  \frac{1}{2} \left[ 1 + \frac{1}{1-\alpha} \right], & \text{for successful search} \\
  \frac{1}{2} \left[ 1 + \frac{1}{(1-\alpha)^2} \right], & \text{for unsuccessful search} 
\end{cases}
\]

**NOTE:**
1) \(\text{RT}_{\text{average}}\) for unsuccessful search is longer than for successful search. Why?!
2) Both formulas “collapse” for \(\alpha = 1\). Why?!
3) What would be the worst case RT of all three methods?!
Open Addressing:  Linear Probing  (cont.)

**Example 1**  [ linear probing ]

\[ h(k) = k \mod 13 \]

Insert keys: 18, 41, 22, 44, 59, 32, 31, 73, in this order.

- \( h(18) = 5 \)
- \( h(41) = 2 \)
- \( h(22) = 9 \)
- \( h(44) = 5+1 \)
- \( h(59) = 7 \)
- \( h(32) = 6+1+1 \)
- \( h(31) = 5+1+1+1+1+1 \)
- \( h(73) = 8+1+1+1 \)
**Open Addressing: Quadratic Probing**

**Quadratic Probing** – involves iteratively trying the following buckets

\[ A[h(k) + j^2 \mod N] \]

where \( j = 0, 1, 2, \ldots \), until finding an empty bucket.

- eliminates primary clusters by probing far-away buckets
- however, creates **secondary clusters** by probing the same pattern-sequence for each item
  - the clusters are still there, but “bounced” around
- if \( N \) is not prime, quadratic probing may not find an empty bucket even if one exists
- even if \( N \) is prime, may not find an empty slot if the bucket array is half-full
Open Addressing: Double Hashing

**Double Hashing** – uses a secondary hash function $h'(k)$ and places the colliding item in the first available cell of the series

$$A[h(k) + j \cdot h'(k)] \mod N$$

where $j = 0, 1, 2, \ldots$, until finding an empty bucket.

- linear & quadratic probing are key independent;
  - double hashing defines key dependant probing sequence – $h'(k)$ determines the size of steps

- $h'$ cannot have 0 values; common choice:
  $$h'(k) = q - (k \mod q)$$

$q < N$, and $q$ is a prime number

possible values for $h'(k)$ are 1, 2, .., $q$

$N = 13, h_2(k) = 26, h'(k) = 4$

probing sequence:
0, 4, 8, 12, 3, 7, 11, …
Open Addressing: Double Hashing (cont.)

Advantages of Double Hashing — drastically reduces clustering and requires fewer comparisons than linear probing
  • as a result, smaller hash tables can be used

Disadvantages of Double Hashing — similar as in the case of linear/quadratic probing, the performance degrades as the table fills up

Average RT — in a non-full hash table, assuming no previous removals, the average running time of insert, find, remove is

\[
RT_{\text{average}}(n) = \begin{cases} 
-\frac{\ln(1-\alpha)}{\alpha}, & \text{for successful search} \\
\frac{1}{1-\alpha}, & \text{for unsuccessful search}
\end{cases}
\]

Using more than one hash function is called rehashing. While having more than 2 hash functions can be desirable, such schemes are difficult to implement.
Open Addressing: Double Hashing (cont.)

Example 2 [double hashing]

\[ h(k) = k \mod 13, \quad h'(k) = 7 - k \mod 7 \]

Insert keys: 18, 41, 22, 44, 59, 32, 31, 73, in this order.

Probes: \[ [h(k) + j \cdot h'(k)] \mod 13 \]

\[
\begin{align*}
    h(18) &= 5 & h'(18) &= 3 & 5 \\
    h(41) &= 2 & h'(41) &= 1 & 2 \\
    h(22) &= 9 & h'(22) &= 6 & 9 \\
    h(44) &= 5 & h'(44) &= 5 & 5, 10 \\
    h(59) &= 7 & h'(59) &= 4 & 7 \\
    h(32) &= 6 & h'(32) &= 3 & 6 \\
    h(31) &= 5 & h'(31) &= 4 & 5, 9, 0 \quad \text{wrap-around} \\
    h(73) &= 8 & h'(73) &= 4 & 8
\end{align*}
\]

<table>
<thead>
<tr>
<th>31</th>
<th>41</th>
<th>18</th>
<th>32</th>
<th>59</th>
<th>73</th>
<th>22</th>
<th>44</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Collision Handling Schemes: Comparison

NOTE: Quadratic Probing and Double Hashing have identical performance.
Collision Handling Schemes: Comparison (cont.)

(1) In all 4 cases, hashing efficiency decreases as the load factor increases.

(2) When $\alpha = 0.5$, i.e. hash table is half-full, all 4 methods have nearly equal efficiency.

(3) For Linear Probing, Quadratic Probing and Double Hashing $\alpha$ should be kept below 2/3 to maintain reasonably good performance.
   - if $\alpha$ goes above 2/3 we could improve performance by resizing bucket array and using new hash function

(4) Quadratic and Double Hashing perform better than Linear Probing on average, but they also suffer when table fills up.
   - all open addressing schemes have linear $O(n)$ worst case RT when $\alpha \to 1$

Would you ever use open addressing instead of Chaining?!

Would you ever use Linear instead of Quadratic Probing and Double Hashing?!
Hash Tables: Conclusions

Hash Tables should be Used When ...

- when we can afford a large table size (i.e. small load factor $\alpha$, below 2/3) and occasionally slow search.

Hash Tables should NOT be Used if ...

1. if traversal in sorted order is required
2. if range queries are required
3. if a guaranteed (better than $O(n)$) search time needs to be provided

In all three cases use binary search tree instead!
Linear Sorting Algorithms: Bucket Sort

**Bucket Sort** — let S be an unordered sequence of n unique integers distributed over interval [0, N-1]; use an auxiliary array (B) of size N ≥ n

**Phase 1:** empty sequence S by moving each item k into bucket B[k]

**Phase 2:** for i=0,..,N-1 move the item of bucket B[i] to the end of sequence S

**Complexity** — O(n) + O(N) = O(N) ≥ O(n)

NOTE: No external comparators are needed !!!

**Complexity** — O(n) + O(N) = O(N) ≥ O(n)

S  
175 210 130 325 270

B

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>130</td>
<td>...</td>
<td>175</td>
<td>...</td>
<td>210</td>
</tr>
<tr>
<td>0</td>
<td>130</td>
<td>175</td>
<td>210</td>
<td>270</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: No external comparators are needed !!!
Linear Sorting Algorithms: Bucket Sort (cont.)

Example 3 [Bucket Sort - special case]

Assume $n$ integers are unique and uniformly distributed over interval $[0, n-1]$, or $[a, a+n-1]$, where $a \geq 0$.

**Phase 1:** Empty sequence $S$ by moving each item $k$ into bucket $B[k-a]$.

**Phase 2:** As before.

Complexity: $O(n) + O(n) = O(n)$ !!!
Bucket Sort in Case of “Duplicates”

– let S be a sequence of any n integers distributed over an interval [0,N-1]; use an auxiliary array (B) of size N; each array position contains pointer to a linked list

Phase 1: empty sequence S by moving each item k at the end of list B[k]

Phase 2: for i=0,..,N-1 move the items of list B[i] to the end of sequence S

Linear Sorting Algorithms: Bucket Sort (cont.)
Linear Sorting Algorithms: Bucket Sort (cont.)

Complexity — (1) + (2) + (3) = O(n) + O(N)

(1) put numbers from S to B: O(n)
(2) scan B for non-empty lists: O(N)
(3) wherever a non-empty list is found, extract all list elements back to S: O(m_1+m_2+..+m_x) = O(n)

If N in O(n) ⇒ Bucket Sort is O(n) !!!

NOTE: Bucket Sort in case of duplicates is a STABLE sorting algorithm.

Stable Sort — We say that a sorting is stable on a sequence (S) if for any two items k_i and k_j of S, such that k_i precedes k_j before sorting, item k_i also precedes k_j after sorting.