CMPT 225
Hashing – Part 2
Collision Resolution Strategies
Last Lecture

- Hashing Functions
Learning Outcomes

- At the end of this unit, a student will be able to:
  - define hashing as well as chained and open addressed hash table
  - discuss tradeoffs in designing hash functions and between collision resolution strategies
  - demonstrate and trace operations on hash table
Today’s menu

• Our goal in this set of lecture notes is to
  ◦ describe collision in hashing
  ◦ present collision resolution strategies
  ◦ discuss tradeoffs of these collision resolution strategies
Problem with Hashing?

-> Collision

- **Definition**: Collision occurs when *multiple distinct* indexing keys are hashed to the same location in the hash table (i.e. the same hash table index is produced for each of these distinct indexing keys)

- These multiple distinct indexing keys are called *synonyms*
Two factors that may minimize the number of collisions are:

◦ Goodness of hash function
◦ Size of the table

but they cannot completely eliminate them
Collision Resolution Strategies

- **Definition**: Algorithms specifying what to do when collisions occur

- Some collision resolution strategies:
  - Open Addressing:
    - Linear Probing Hashing
    - Quadratic Probing Hashing
    - Random Probing Hashing
    - Rehashing (Double Hashing)
  - Chain Hashing
Inserting/Searching/Deleting in a Hash Table – Scenario 1

1. Compute hash index $h(k)$ using indexing key $k$ and “% array size”

2. Probe the resulting location in hash table $\rightarrow$ no collision

- If we are performing an **insertion**, then we go ahead and $\text{insert(newElement)}$
- If we are performing a **search**, then $\text{targetElement}$ is not found!
- If we are performing a **deletion**, then there is no element to delete
Inserting/Searching/Deleting in a Hash Table – Scenario 2

1. Compute hash index \( h(k) \) using indexing key \( k \) and “\% array size”
2. Probe the resulting location in hash table -> no collision

- If we are performing an **insertion**, then we are done since “the” element has already been inserted!
- If we are performing a **search**, then targetElement is found!
- If we are performing a **deletion**, then element is labelled “ToBeDelete”
Inserting/Searching/Deleting in a Hash Table – Scenario 3

1. Compute hash index $h(k)$ using indexing key $k$ and “% array size”
2. Probe the resulting location in hash table -> collision
3. Then we follow one of the open addressing collision resolution strategies described on the following slides
Overview of Open Addressing

1. Compute hash index $h(k)$
2. Probe the resulting location in hash table – possible outcomes are:

   1) We find an empty cell – then we follow Scenario 1 on one of the previous slides
   2) We find the cell occupied by “the” element – then we follow Scenario 2 on one of the previous slides
   3) We find the cell occupied by another element -> collision occurs and it is resolved by probing another cell in hash table, i.e., repeating above Step 1 and 2 (see Scenario 3)
   4) We discover that the hash table is full, i.e., we have probed all locations
      * In the case of an insertion, we need to expand the hash table
      * In the case of a search (retrieval or deletion), the element we were looking has not been found
We compute the hash indices using the following probing sequence:

1\textsuperscript{st} probe: \( h(k) \)  \(\rightarrow\) collision occurs  \((\text{original - 1}\textsuperscript{st} - \text{hash index computed})\)

2\textsuperscript{nd} probe: \( h'(k) = ( h(k) + p(1) ) \mod \text{sizeOfHashTable} \)  \((2\textsuperscript{nd} \text{hash index computed})\)

3\textsuperscript{rd} probe: \( h'(k) = ( h(k) + p(2) ) \mod \text{sizeOfHashTable} \) \((3\textsuperscript{rd} \text{hash index computed})\)

\ldots

\( j\textsuperscript{th} \) probe: \( h'(k) = ( h(k) + p(i) ) \mod \text{sizeOfHashTable} \)  \((j\textsuperscript{th} \text{hash index computed})\)

where \( p(i) \)  \(\rightarrow\) probing function
1. Linear Probing Hashing

- \( p(i) = i \) for \( i = 1, 2, \ldots \)
- Hence, we compute the next hash indices using the following probing sequence:
  
  1\(^{st}\) probe: \( h(k) \rightarrow \textit{collision occurs} \) (original (1\(^{st}\)) hash index computed)
  2\(^{nd}\) probe: \( h'(k) = (h(k) + 1) \mod \text{sizeOfHashTable} \) (2\(^{nd}\) hash index computed)
  3\(^{rd}\) probe: \( h'(k) = (h(k) + 2) \mod \text{sizeOfHashTable} \) (3\(^{rd}\) hash index computed)
  
  ... 

  \( j^{th}\) probe: \( h'(k) = (h(k) + i) \mod \text{sizeOfHashTable} \) (\( j^{th}\) hash index computed)
**Inserting an Element using Linear Probing Hashing**

**Step 1.** If hash table is not full proceed to **Step 2**
else expand hash table *(unbeknownst to the user)*

**Step 2.** Compute hash index \((h(k) \text{ or } h'(k))\) of element

**Step 3.** Probe cell at \(\text{hashTable}[h(k) \text{ or } h'(k)]\)

- Is cell occupied?
  - No -> insert element -> done! -> \(O(1)\)
  - Yes -> is element to be inserted already in cell?
    - Yes -> done! *(assumption: no duplication)* -> \(O(1)\)
    - No -> **Collision**
      - Got to **Step 2.** i.e., compute next hash index \(h'(k)\) of
        element following the Linear Probing Hashing alg.
      - In other words: start **linear search** for an empty cell,
        wrapping around to the beginning of hash table if we reach
        the end *(using modulo operator or other means)*
      - Worst case: \(O(n)\)
Example

Insert the following elements with indexing key value:

32, 47, 26, 34, 87, 39, 78, 61, 48, 66

Hash index \( h(k) \):

\# of probes:

Hash table:

\( n = 10 \)
Example – Result

Key Space
00 to 99

\[ \text{hash(key)} \Rightarrow \text{key \% 10} \]

Index Space
0 to 9

\( n = 10 \)

Insert the following elements with key value:
32, 47, 26, 34, 87, 39, 78, 61, 48, 66

Hash code:
2 7 6 4 7 9 8 1 8 6

# of probes:
1 1 1 1 2 1 3 1 6 10

Hash Table:
0 1 2 3 4 5 6 7 8 9
78 61 32 48 34 66 26 47 87 39
Searching an Element using Linear Probing Hashing

**Step 1.** If hash table is not empty

**Step 2.** Compute hash index \( h(k) \) or \( h'(k) \) of element

**Step 3.** Probe cell at \( \text{hashTable}[h(k) \text{ or } h'(k)] \)
   
   - is element found?
     Yes -> done! -> \( O(1) \)
     No -> is cell empty?
       Yes -> element not in hash table -> \( O(1) \)
       No -> **Collision**
         
         Got to **Step 2.** i.e., compute next hash index \( h'(k) \) of element following the Linear Probing Hashing alg.
         
         • In other words: start **linear search** for the element, wrapping around to the beginning of hash table if we reach the end (using modulo operator or other means)
         • Worst case: \( O(n) \)
Observations about Linear Probing Hashing

- As we fill our hash table, what is happening?

- Major drawback of linear probing hashing strategy:
Definition of a Cluster

- Consecutive group of occupied cells
Example Of Clustering

About to insert '78'

using Linear Probing Hashing.

Primary cluster
Hence...

- Cluster formation undermines the performance of hash table operations:
  - Insertion
  - Search (retrieval and deletion)

- Question: How to avoid primary cluster buildup?
Solution to Primary Clustering

Answer: Choosing the probing function $p(i)$ more carefully
2. Quadratic Probing Hashing #1

- \( p(i) = i^2 \) for \( i = 1, 2, ... \)

- Hence, we compute the hash indices using the following probing sequence:
  
  1\textsuperscript{st} probe: \( h(k) \) (original (1\textsuperscript{st}) hash index computed)
  
  2\textsuperscript{nd} probe: \( h'(k) = ( h(k) + 1 ) \mod \text{sizeOfHashTable} \) (2\textsuperscript{nd} hash index computed)
  
  3\textsuperscript{rd} probe: \( h'(k) = ( h(k) + 4 ) \mod \text{sizeOfHashTable} \) (3\textsuperscript{rd} hash index computed)
  
  4\textsuperscript{th} probe: \( h'(k) = ( h(k) + 9 ) \mod \text{sizeOfHashTable} \) (4\textsuperscript{th} hash index computed)
  
  ...  

  \( j\textsuperscript{th} \) probe: \( h'(k) = ( h(k) + i^2 ) \mod \text{sizeOfHashTable} \) (\( j\textsuperscript{th} \) hash index computed)
2. Quadratic Probing Hashing #2

- \( p(i) = +/- i^2 \) for \( i = 1, 2, ... \)

- Hence, we compute the hash indices using the following probing sequence:

  1\st\ probe: \( h(k) \) (original \( 1\st \) hash index computed)

  2\nd\ probe: \( h'(k) = ( h(k) + 1 ) \ % \ \text{sizeOfHashTable} \) (2\nd\ hash index computed)

  3\rd\ probe: \( h'(k) = ( h(k) - 1 ) \ % \ \text{sizeOfHashTable} \) (3\rd\ hash index computed)

  4\th\ probe: \( h'(k) = ( h(k) + 4 ) \ % \ \text{sizeOfHashTable} \) (4\th\ hash index computed)

  5\th\ probe: \( h'(k) = ( h(k) - 4 ) \ % \ \text{sizeOfHashTable} \) (5\th\ hash index computed)

  ...

  when j is odd:

  \( j\th \) probe: \( h'(k) = ( h(k) + i^2 ) \ % \ \text{sizeOfHashTable} \) (\( j\th \) hash index computed)

  when j is even:

  \( j\th \) probe: \( h'(k) = ( h(k) - i^2 ) \ % \ \text{sizeOfHashTable} \) (\( j\th \) hash index computed)
LINEAR PROBING HASHING

Hash Table

\[ \text{hash fcn(key)} \]

QUADRATIC PROBING, HASHING

1. \[ #1 \]

2. \[ #2 \]

Hash Table

\[ \text{hash fcn(key)} \]
Quadratic Probing Hashing

For this strategy to work well, one may apply the following constraint:

- Size of hash table should not be an even number
  - Increase probability that each position in hash table is included in probing sequence (i.e., hashed)
- Ideally, size of hash table should be a prime $4j+3$ (whenever this equation produces a prime for a particular value of $j$)
  - Guarantees the inclusion of all positions in the probing sequence (Radke 1970)
Examples of Quadratic Probing Hashing

• Example #1:
  ◦ If j=2, then size of hash table is 11
  ◦ Assume that \( h(k) = 9 \), for some indexing key \( k \), what is the resulting sequence of probes using
    • Quadratic Probing Hashing #1?
    • Quadratic Probing Hashing #2?
Example #1

\[ j = 2 \quad \therefore \text{size} = 11 \]

\[ h(K) = 9 \]

\[ \text{Q.P.H. #1: 9, 10, 13, 18, 3, 1, 1, 3, 7, 2, 10} \]
Example #1

\[ j = 2 \quad \therefore \text{size} = 11 \]
\[ h(k) = 9 \]

Q.P.H. #2: \[ 9, 10, 8, 2, 5, 7, \emptyset, 3, 4, 1, 6 \]
Examples of Quadratic Probing Hashing

Example #2:

○ If size of hash table is 10
○ Assume that $h(k) = 9$, for some indexing key $k$, what is the resulting sequence of probes using
  • Quadratic Probing Hashing #1?
  • Quadratic Probing Hashing #2?
Example #2

\[ h(k) = 9 \]

Q.P.H #1: 9, \( \emptyset \), 3, 8, 5, 4, 5, 8, 3, \( \emptyset \)

Q.P.H #2: 9, \( \emptyset \), 8, 3, 5, 8, \( \emptyset \), 5, 3, 4
Summary – Quadratic Probing Hashing

- Advantage: reduce the kind of clustering that occurs with linear probing hashing (called primary clustering)

- Disadvantage: produces a different kind of clustering (called secondary clustering)
Observation

So far…

- All synonyms produce the same hash index sequences (no matter what the indexing key value is)

  - Goal: For each indexing key, generate a different hash index sequence
For Example ...

assuming a hash fen ...

\[ \begin{align*}
\text{Key 1} \\
\text{Key 59} \\
\text{Key 127}
\end{align*} \]

synonyms \rightarrow \text{index } h(K) = 3

\[ \begin{align*}
\therefore \text{sequence of hash indices for these synonyms}
\end{align*} \]

\text{LINEAR: } 3, 4, 5, 6 \ldots \\
\text{QUADRATIC: } 3, 4, 7, 12 \ldots
A Solution:

3. Random Probing Hashing

- \( p(i) \) is a random number generator

- Hence, we compute the hash indices using the following probing sequence:

  1\(^{st}\) probe: \( h(k) \) (original \( 1^{st} \) hash index computed)

  2\(^{nd}\) probe: \( h'(k) = (h(k) + r_1) \mod \text{sizeOfHashTable} \) (2\(^{nd}\) hash index computed)

  3\(^{rd}\) probe: \( h'(k) = (h(k) + r_2) \mod \text{sizeOfHashTable} \) (3\(^{rd}\) hash index computed)

  ...

  \( j^{th}\) probe: \( h'(k) = (h(k) + r_{s-1}) \mod \text{sizeOfHashTable} \) (\( j^{th}\) hash index computed)

  where \( r_1, r_2, ..., r_{s-1} \) are random numbers (\( \neq 0 \)) and \( s \) is \( \text{sizeOfHashTable} \)
Example

Insert \( k_1 \) & \( k_2 \) (\( k_0, k_1, k_2 \) are synonyms)

\[
\begin{align*}
\text{Step 1} & \quad h(k_1) = 2 \\
& \quad h(k_2) = 2
\end{align*}
\]

\[ x_i : \]
\[
\begin{align*}
& r_1 = 3 \\
& r_2 = 1 \\
& r_3 = 5
\end{align*}
\]

probe \#2 = 5

probe \#3 = 3

probe \#4 = 7

capacity = 11
Summary – Random Probing Hashing

• Advantage:
  ◦ No more constraint on hash table size
  ◦ Prevents formation of secondary clusters

• Disadvantage:
  ◦ Imposes the constraint that the probing sequence must be the same every time it is generated for a particular indexing key
    • Otherwise, an element with indexing key $k$ that has already been inserted into the hash table may not be found again
Random Probing Hashing

- **Solution #1**: If the chosen random number generator is such that it *may generates different probing sequences* for a particular indexing key (if, for example, the random number generator is initialized at first invocation only), we must save the generated random numbers $r_1, r_2, \ldots, r_{s-1}$ for that indexing key.

- **Solution #2**: If the chosen random number generator is such that it *always generates the same probing sequence* for a particular indexing key (if, for example, the random number generator can be initialized with the same seed every time the probing sequence for a particular indexing key is generated), we must save the seed for that indexing key or generate the seed from the indexing key.
4. Rehashing Probing Hashing

- $p(i) = h_p(k)$ is a hashing function itself

- Hence, we compute the hash indices using the following probing sequence:
  
  1\textsuperscript{st} probe: $h(k)$ (original (1\textsuperscript{st}) hash index computed)
  
  2\textsuperscript{nd} probe: $h'(k) = (h(k) + h_p(k)) \mod \text{sizeOfHashTable}$ (2\textsuperscript{nd} hash index computed)
  
  3\textsuperscript{rd} probe: $h'(k) = (h(k) + 2 \times h_p(k)) \mod \text{sizeOfHashTable}$ (3\textsuperscript{rd} hash index computed)
  
  ... 
  
  \(j\textsuperscript{th}\) probe: $h'(k) = (h(k) + (j-1) \times h_p(k)) \mod \text{sizeOfHashTable}$ (\(j\textsuperscript{th}\) hash index computed)
Rehashing Probing Hashing

• Constraints:
  ◦ Size of hash table should be a prime number so that each position in the table can be included in the sequence
  ◦ $h_p(k) \neq 0$
Example

• Textbook:
  ◦ See Double Hashing pages 552-553
Summary – Random and Rehashing Probing Hashing

- **Advantage:** reduce clustering, hence improve time efficiency (i.e., help keep time efficiency $O(1)$)

- **Disadvantage:**
  - overhead
    - -> some space
    - -> some computation
  - We still have to ensure that all locations are probed
What we saw in this lecture

- Described collision in hashing
- Presented collision resolution strategies
- Discussed trade-offs of these collision resolution strategies
Next Lecture

- Hashing – Part 3 – Chain Hashing