How To View This Presentation

This presentation is arranged in outline format. To view the slides in proper order:

- For each slide:
  - Read the entire slide’s text.
  - Then click the [links](#) (if any) in the text, in the order they appear in the text.
  - Then click the [up](#) or [play](#) button at the bottom-right side of the slide.
  - Do not use the space bar to advance.
Overview

• Hashed data structures’ operations are very fast, and their density can approach 1
  – Node references are stored in a primary storage area
  – A hashing function “computes” the location a node’s reference rather than using a search algorithm

• There are two categories of hashed structures
  – Perfect hashed structures (fastest)
  – Non-perfect hashed structures (a bit slower)

• These structures can be generic and dynamic
  • Java’s Hashtable class is a generic, dynamic, non-perfect hashed structure
Primary Storage Area

• Usually implemented as an array
• Four popular schemes
  – Array of references to nodes
  – Array of linked list headers (the linked lists contain the nodes)
  – Array of nodes
  – Array of references to arrays of nodes
• Only the first two schemes can be implemented in Java
Java and Non-Java Primary Storage Area Schemes

Array of Node References

<table>
<thead>
<tr>
<th>Primary Storage Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill’s node</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Tom’s node</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Ann’s node</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Array of Linked List Headers

<table>
<thead>
<tr>
<th>Primary Storage Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill’s node</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Tom’s node</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>

Array of Nodes

<table>
<thead>
<tr>
<th>Primary Storage Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill’s node</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Tom’s node</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Ann’s node</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Array of References to Arrays of Nodes

<table>
<thead>
<tr>
<th>Primary Storage Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bills node</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Ann’s node</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Tom’s node</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

© 2009 Jones and Bartlett Publishers, LLC (www.jbpub.com)
A Hashing Function

- A **function**, $h$, used to map a key, $k$, (or node number) into a location, $i_p$ in the primary storage area, $i_p = h(k)$

- **Often non-numeric keys** and **negative keys** are first **preprocessed** into a positive numeric pseudokey
  - The pseudokey, $pk$, is then used in the hashing function to determine $i_p$

```
key, k  →  Preprocessing algorithm  →  pseudo key, pk  →  Hashing Function
          |                     |                     |  $i_p = h(pk)$
          |                     |                     |  the primary storage area location, $i_p$
```

```
key, k  →  Hashing Function
          |                     |                     |  $i_p = h(k)$
          |                     |                     |  the primary storage area location, $i_p$
```
Two Popular Hashing Functions

• Direct hashing Function
  \[ ip = k \quad \text{or} \quad ip = pk \]

• Division hashing function
  \[ ip = k \mod N \quad \text{or} \quad ip = pk \mod N \]
  Where \( N \) is the size of the primary storage area

• Each has its strengths \( \smiley \) and weaknesses \( \frownie \) as revealed by an example of their use
Stadium Ticket Data Base Example

• Stores the ticket number and ticket purchaser’s name for a 10,054 seat stadium

• Ticket number is
  – the key field
  – a six digit encoding of the seat number, event number, and event date
  – Ticket numbers range from 000000 to 999999

• The division and direct hashing algorithms are used to hash several ticket numbers into primary storage locations
## Results Of Hashing Several Ticket Numbers (N = 10,054 for the Division Algorithm)

e.g., for ticket number 342556:
- **Division**: \( ip = key \mod N = 342,554 \mod 10,054 = 720 \)
- **Direct**: \( ip = key = 342,556 \)

<table>
<thead>
<tr>
<th>Ticked Number (key value)</th>
<th>Calculated Location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Division Algorithm (N = 10,054)</td>
</tr>
<tr>
<td>342556</td>
<td>720</td>
</tr>
<tr>
<td>0000000</td>
<td>0</td>
</tr>
<tr>
<td>9999999</td>
<td>4,653</td>
</tr>
<tr>
<td>211854</td>
<td>720</td>
</tr>
</tbody>
</table>
Strengths And Weaknesses

• Division algorithm \((N = 10,054)\)
  – Generates a number in the range \(0 \rightarrow 10,053\)
    • No wasted space in the primary storage area (one location per ticket)
  – Maps different ticket numbers into the same location
    • Called a \textit{collision}, slows down the structure

• Direct Algorithm
  – Generates a location in the range \(0 \rightarrow 999,999\)
    • \textit{Wasted space} (100K locations, only 10,054 are used)
  – Never maps different keys into the same location
    • Fast, no collisions
Collisions Associated With The Division Algorithm

• A collision occurs when two keys map into the same primary storage area location
  – E.g. $342,556 \% 10,054 = 211,854 \% 10,054 = 720$

• A collision algorithm is added to the mapping process
  – Produces a different location 😊
  – Slows down the mapping process and therefore the structure 😞
Wasted Space Associated With The Direct Algorithm

- The primary storage area must be sized to the maximum value of the hashing function +1 = N (to prevent indexing out of bounds)
  - For the ticket example, 999,999 + 1 = 100,000
- Often the maximum number of nodes, n, is less than N + 1
  - For the ticket example, n = 10,054 tickets (nodes)
- N - n locations in primary storage is unused
  - For the ticket example, 100,000 - 10,054 = 89,946 wasted locations (89.9% unused)
Perfect Hashed Structures

- Hashed structures that use perfect hashing functions
- Their operation speed is faster than any other data structure
- Their densities can be very high
- The Direct Hashed data structure is a perfect hashed structure
Perfect Hashing Functions

• Perfect hashing functions
  – Map each key into a unique primary storage area location
    • E.g., the Direct Hashing function
  – Can waste space in the primary storage area
    • E.g., our stadium ticket example

• Some perfect hashing functions minimize the wasted space in primary storage
  – Called minimum perfect hashing functions
  – Often difficult to design, highly application dependent
  – Under some conditions, the Direct Hashing function is a perfect hashing function
Condition That Makes The Direct Hashing Algorithm A Perfect Hashing Algorithm

• When most (or all) possible key values will be stored in the structure

• For example
  – Keys are three digits 0 → 999 and ~1,000 (= 10*10*10) nodes will be stored in the structure
  – Keys are four capital letters and ~456,976 (= 26*26*26*26) nodes will be stored in the structure
The Direct Hashed Data Structure

• Uses the direct hashing function \( ip = k \); no collisions
• Its **primary storage** area is an array of node references
• Alphanumeric keys must be preprocessed to generate numeric pseudokeys
• When dealing with numeric keys, the **subtraction preprocessing** algorithm is often used
• Supports all four operations in the key field mode
• Its **operation algorithms** are simple
• It’s **performance** (speed *and* density) can both be very good
N-1 is the maximum value of the numeric key or pseudokey, usually < the maximum number of nodes to be stored.
**Subtraction Preprocessing Algorithm**

\[ pk = k - k_{min} \]

- Used when the minimum value of a numeric key, \( k_{min} \), is non-zero
  - Keeps **negative keys** within array bounds
  - Reduces primary storage size for \( k_{min} > 0 \)

<table>
<thead>
<tr>
<th>Minimum key Negative (( = -3 ))</th>
<th>Minimum key &gt; 0 (( = 112 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>The given key, ( k ),</strong>&lt;br&gt;<strong>(the index without preprocessing)</strong></td>
<td><strong>The index with preprocessing</strong>&lt;br&gt;( ip = pk = k - k_{min} = k - (-3) )</td>
</tr>
<tr>
<td>(-3)</td>
<td>0</td>
</tr>
<tr>
<td>(-2)</td>
<td>1</td>
</tr>
<tr>
<td>(-1)</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
Direct Hashed Structure Operation Algorithms

- **Insert, Fetch, Delete** algorithms very simple
- All three algorithms use the Direct Hashing function
- Update is assumed to be a Delete operation followed by an Insert Operation
Action Of The Hashed Structure’s Insert Algorithm

The structure after Bill and Bob are inserted

Bob
preprocessing and hashing

Bill
preprocessing and hashing

Algorithm
Direct Hashed Structure’s Insert Algorithm

Inserts `newNode` whose key is `targetKey` into the structure

```
1. // access the primary storage area
2. pseudoKey = preProcessing(targetKey);
3. ip = pseudoKey; // direct hashing function
4. // insert the new node
5. data[ip] = newNode.deepCopy();
```
Action Of The Hashed Structure’s Fetch Algorithm

The structure after Bill’s node is fetched

Algorithm
Direct Hashed Structure’s Fetch Algorithm

Fetcheds the node whose key is \texttt{targetKey} from the structure

1. // access the primary area
2. \texttt{pseudoKey} = \texttt{preProcessing(targetKey)};
3. \texttt{ip} = \texttt{pseudoKey}; // direct hashing function
4. // return a clone of the node or a \texttt{null} reference
5. \texttt{if(data[ip] == null)}
6. { \texttt{return null}; }
7. \texttt{else}
8. { \texttt{return data[ip].deepCopy( );}
9. }
Action Of The Hashed Structure’s Delete Algorithm

The structure after Bob’s node is deleted

Algorithm
Direct Hashed Structure Delete Algorithm

Deletes the node whose key is targetKey from the structure

1. // access the primary area
2.  pseudoKey = preProcessing(targetKey);
3.  ip = pseudoKey; //direct hashing function
4.  // delete the node
5.  if(data[ip] == null)
6.     return false;  // not in structure
7.  else
8.     {  data[ip] == null;  // delete the node
9.     return true;
10. }
Direct Hashed Structure's Performance

• **Speed**
  – Its speed is always very, very fast

• **Density**
  – For some applications its density can be high
  – Dependent on the
    • Node width (as usual)
    • Degree to which the primary storage area is used
  – In our stadium ticket example,
    \[ n = \text{number of nodes} = 10,054 \quad \text{and} \quad N = \text{array size} = 100,000 \]
    the density would be 0.8 if the node width was 160 bytes
Speed Of The Direct Hashed Structure

• None of the operation algorithms have search loops, therefore O(1)

• Memory Access analysis
  – The algorithms all use preprocessing
    • The subtraction algorithm requires 2 memory accesses
  – The algorithms perform 1 memory access to fetch data[i]
  – Delete performs 1 more access to overwrite data[i]

• Insert and Fetch 3 access, Delete 4 accesses
Density Of The Direct Hashed Structure

- Density = information bytes / total bytes
  - Information bytes = \( n \times w \)
    - \( n \) is the number of nodes, \( w \) is the bytes per node
  - Overhead = \( 4N \)
    - \( N \) is the array size, 4 bytes per array element
- Density = \( n \times w / (n \times w + 4N) \)
  = \( 1 / (1 + 4 \times N/(n \times w)) \)
  = \( 1 / (1 + 4 / (l \times w)) \)
  - \( l = n/N \) called the Loading Factor
- For \( l \approx 1 \), the density is equivalent to our other array-based structures
Variation in Density Of the Direct Hashed Structure

\( w \) is the node width in bytes

Loading factor, \( l = \frac{n}{N} \)

- \( w = 10 \)
- \( w = 20 \)
- \( w = 60 \)
- \( w = 100 \)
- \( w = 1000 \)
Conditions That Produce A Density of 0.8 For the Direct Hashed Structure

<table>
<thead>
<tr>
<th>Loading Factor, $l = \frac{n}{N}$</th>
<th>Node width, $w$, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>350</td>
</tr>
<tr>
<td>0.2</td>
<td>300</td>
</tr>
<tr>
<td>0.3</td>
<td>250</td>
</tr>
<tr>
<td>0.4</td>
<td>200</td>
</tr>
<tr>
<td>0.5</td>
<td>150</td>
</tr>
<tr>
<td>0.6</td>
<td>100</td>
</tr>
<tr>
<td>0.7</td>
<td>50</td>
</tr>
<tr>
<td>0.8</td>
<td>0</td>
</tr>
</tbody>
</table>

Stadium Ticket Example: $l = \frac{10,054}{100,000} = 0.1005$
### Performance Of The Direct Hashed Structure

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Operation Speed (in memory accesses)</th>
<th>Condition for Density &gt;0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insert</td>
<td>Delete</td>
</tr>
<tr>
<td>Unsorted-Optimized Array</td>
<td>3</td>
<td>≤ n</td>
</tr>
<tr>
<td>Stack and Queue</td>
<td>5</td>
<td>combined with fetch</td>
</tr>
<tr>
<td>Singly Linked List</td>
<td>6</td>
<td>1.5n</td>
</tr>
<tr>
<td>Direct Hashed (with pre-processing)</td>
<td>1 or (3)</td>
<td>2 or (4)</td>
</tr>
</tbody>
</table>

[1] assumes all operations equally probable; w is node width, l is the loading factor.
Non-Perfect Hashed Structures

- Referred to simply as Hashed Structures
- Their hashing algorithms result in collisions
- Their performance can be optimized if
  - Their array size is properly chosen
  - Preprocessing is properly chosen
  - The collision algorithm is properly chosen
  - A problem associated with Delete is dealt with
- An efficient non-perfect hashed structure is the LQHashed structure
Collisions

- A collision occurs when two keys map into the same primary storage area location.

- Since non-perfect hashed structures use non-perfect hashing functions, collisions are inevitable.

- Their **operation algorithms** include a collision algorithm to resolve collisions.
Hashed Structure Operation Algorithm

1. Preprocessing algorithm generates pseudo key, $pk$ from the given key, $k$

2. Hashing algorithm generates the array location, $i_p = h(pk)$

3. Check if correct location?
   - Yes: Perform operation on node at $i_p$
   - No: Collision algorithm generates new location

4. End
Memory Level View Of an Insert Operation

The First Unsuccessful Attempt to Insert Tom’s Information

The Second Unsuccessful Attempt to Insert Tom’s Information
The Third (Successful) Attempt to Insert Tom’s Information

Two additional memory access were required to resolve the collisions
Sizing the Primary Storage Area Array of a Hashed Structure

• Based on an optimum loading factor, 0.75
  – Too high (≈1) → many collisions and therefore slow operations
  – Too low (< 0.75) → many unused array elements, therefore low density
• And a $4k+3$ prime number, $p_{4k+3}$
  – Using a prime results in less collisions
• Array size, N, is the $p_{4k+3}$ prime just above $n/0.75$
  – An example that sets N to the $p_{4k+3}$ prime 6,863
• Division Hashing function keeps array index in bounds
Optimum Loading Factor Of a Hashed Structure

• *Search length*, $L$, is the number of array access required to find a “correct” location (resolve collisions)

• Appendix B derives the average search length, $L_{avg}$, in terms of the loading factor, $l$, to be: $L_{avg} \leq \frac{1}{1 - l}$
  – As $l$ approaches 1, $L_{avg}$ approaches infinity
  – As $l$ decreases from 1, $L_{avg}$ drops rapidly
  – Below $l = 0.75$, there is very little decrease in search length

• At a loading factor of 0.75
  – $L_{avg} \leq \frac{1}{(1 - 0.75)} = 4$
  – *Density* $\geq 0.8$ for node widths ($w$) $\geq 20$ bytes
Variation Of Average Search Length With Loading Factor

Average Search Length $\leq \frac{1}{(1 - l)}$

Optimum loading Factor, $l = 0.75$ (point of diminishing returns)
Density Variation With Node Width At A loading Factor of 0.75

Density = \( \frac{n \times w}{n \times w + 4N} = \frac{1}{1 + \frac{4}{(l \times w)}} \)

\( n \) is the number of nodes, \( N \) is the array size, \( l \) is the loading factor.

At \( l = 0.75 \), Density = \( \frac{1}{1 + \frac{5.33}{w}} \)

Density of a Hashed Structure With a Loading Factor of 0.75

Density = \( \frac{w}{w + 5.33} \)

Good Density above this node width
4k + 3 Primes

- **Prime numbers** are integers only divisible evenly by themselves and 1
- A prime, p, is a 4k+3 prime if there is an integer, k, such that \( p = 4k + 3 \)
- Test: set \( p = 4k+3 \), solve for k, is k an int?
  - For example, 857 and 859 are primes
    - 857 is *not* a 4k+3 prime: \( k = \frac{(857 - 3)}{4} = 213.5 \)
    - 859 *is* a 4k+3 prime: \( k = \frac{(859 - 3)}{4} = 214.0 \)
Hashed Structure
Primary Storage Array Sizing Example

5,129 nodes will be stored in the structure,
\[ n = 5,129 \]
\[ n / 0.75 = 6838.7, \] first prime above 6838 is 6841.

Is 6,841 a 4k+3 prime?
\[ k = (6,841 - 3) / 4 = 1709.5 \] NO

Is next higher prime, 6,857 a 4k+3 prime?
\[ k = (6,857 - 3) / 4 = 1713.5 \] NO

Is next higher prime, 6,863 a 4k+3 prime?
\[ k = (6,863 - 3) / 4 = 1715.0 \] Yes

Array size \( N = 6,863 \)
Prime Numbers Between 4,000 And 7,000

4001, 4003, 4007, 4013, 4019, 4021, 4027, 4049, 4051, 4057, 4073, 4079, 4091, 4093, 4099, 4111, 4127, 4129, 4133, 4139, 4153, 4157, 4159, 4177, 4201, 4211, 4217, 4219, 4229, 4231, 4241, 4243, 4253, 4259, 4261, 4271, 4273, 4283, 4289, 4297, 4327, 4337, 4339, 4349, 4357, 4363, 4373, 4391, 4397, 4409, 4421, 4423, 4441, 4447, 4451, 4457, 4463, 4481, 4493, 4507, 4513, 4517, 4519, 4523, 4547, 4549, 4561, 4567, 4583, 4591, 4597, 4603, 4621, 4637, 4639, 4643, 4649, 4651, 4657, 4663, 4673, 4679, 4691, 4703, 4721, 4723, 4729, 4733, 4751, 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813, 4817, 4831, 4861, 4871, 4877, 4889, 4903, 4909, 4919, 4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969, 4973, 4987, 4993, 4999, 5003, 5009, 5011, 5021, 5023, 5039, 5051, 5059, 5077, 5081, 5087, 5099, 5101, 5107, 5113, 5119, 5147, 5153, 5167, 5171, 5179, 5189, 5197, 5209, 5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297, 5309, 5323, 5333, 5347, 5391, 5387, 5393, 5399, 5407, 5413, 5417, 5419, 5431, 5439, 5441, 5443, 5449, 5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519, 5521, 5527, 5531, 5557, 5563, 5569, 5573, 5581, 5591, 5623, 5639, 5641, 5647, 5651, 5653, 5657, 5659, 5669, 5683, 5689, 5693, 5701, 5711, 5717, 5737, 5741, 5743, 5749, 5779, 5783, 5791, 5801, 5807, 5813, 5821, 5827, 5839, 5843, 5849, 5851, 5857, 5861, 5867, 5869, 5879, 5881, 5897, 5903, 5923, 5927, 5939, 5953, 5961, 5987, 6007, 6011, 6029, 6037, 6043, 6047, 6053, 6067, 6073, 6079, 6089, 6091, 6101, 6113, 6121, 6133, 6133, 6143, 6151, 6163, 6173, 6197, 6199, 6203, 6211, 6217, 6221, 6229, 6247, 6257, 6263, 6269, 6271, 6277, 6287, 6299, 6301, 6311, 6317, 6323, 6329, 6337, 6343, 6353, 6359, 6361, 6367, 6373, 6379, 6389, 6397, 6421, 6427, 6449, 6451, 6469, 6473, 6481, 6491, 6521, 6529, 6547, 6551, 6553, 6563, 6569, 6571, 6577, 6581, 6599, 6607, 6619, 6637, 6653, 6659, 6661, 6673, 6679, 6689, 6691, 6701, 6703, 6709, 6719, 6733, 6737, 6761, 6763, 6779, 6781, 6791, 6793, 6803, 6823, 6827, 6829, 6833, 6841, 6857, 6863, 6869, 6871, 6883, 6899, 6907, 6911, 6917, 6947, 6949, 6959, 6961, 6967, 6971, 6977, 6983, 6991, 6997
Division Hashing Algorithm Keeps Array Index In Bounds

- If the primary storage array has 6,863 elements (N = 6,863)
  - Valid indexes are \(0 \rightarrow 6,862\)
- For *any* numeric pseudokey, \(pk\)
  
  \[
  \text{Array index} = i_p = pk \% 6,863 \text{ yields } 0 \leq i_p \leq 6,862
  \]
- E. g., for \(pk = 10,090,781; i_p = 2,171\)
Preprocessing Algorithms For Hashed Structures

• Generally,
  – Combine a key’s characters into a 32 bit pseudokey
  – The 32 bits are interpreted as an integer
  – The integer is processed by the hashing function

• Good algorithms introduce randomness into the pseudokeys

• Common algorithms are Fold-shifting, Pseudorandom processing, and Digit Extraction

• A preprocessing sequence could be: digit extraction, then fold-shifting, then pseudorandom processing
Fold-shifting Preprocessing
(to generate a 32 bit pseudokey)

• The key is divided into groupings of four characters
• The groupings are interpreted as integers and added; Overflow is ignored
• The sum is the pseudokey
• Often, one of the four character groupings is considered a “pivot”, or first operand
An Example Of Fold-Shifting Preprocessing Of the Key “AlMcAllister”

The key’s characters are grouped: AlMcAllister
The key is folded about the second grouping, Alli
The pseudokey generated is 4,132,249,406

The addition is performed as:

\[
\begin{align*}
\text{Alli} & \quad 0100 \ 0001 \ 0110 \ 1100 \ 0110 \ 1100 \ 0110 \ 1001 \\
A & \ 0100 \ 0001 \\
\text{AlMc} & \quad 0100 \ 0001 \ 0110 \ 1100 \ 0100 \ 1101 \ 0110 \ 0011 \\
A & \ 0100 \ 0001 \\
\text{ster} & \quad 0111 \ 0011 \\
+ \text{AlMc} & \quad 0100 \ 0001 \ 0110 \ 1100 \ 0100 \ 1101 \ 0110 \ 0011 \\
+ \text{ster} & \quad 0111 \ 0011 \\
\text{Alli} & \quad 1111 \ 0110 \ 0100 \ 1101 \ 0001 \ 1111 \ 0011 \ 1110 = 4,132,249,406
\end{align*}
\]
Pseudorandom Preprocessing

• Used to introduce randomness into
  – Numeric keys, k
  – Numeric pseudokeys, pk, generated from previously preprocessing alphanumeric keys

• The pseudo key pk is:
  \[ pk = p_1 \times k + p_2, \text{ or } pk = p_1 \times pk + p_2 \]
  where: \( p_1 \) and \( p_2 \) are primes, overflow is ignored

• E. g., \( pk = 2,132,249,406; p_1 = 13, p_2 = 53 \)
  \( pk = 2,132,249,406 \times 13 + 53 = \)
  \[ = 1,949,438,555 \text{ (32 bit with overflow ignored)} \]
Digit Extraction Preprocessing

• Like folding, used to reduce the length of multi-character keys (numeric or non-numeric)

• Extracts characters from keys that add no uniqueness (common to all keys)
  – If all keys begin with “Tom”, the first three characters would be eliminated from all keys

• Extracts bits from characters that add no uniqueness
  – If all key characters were lower case letters, the left most 3 bits (000) would be ignored in all characters
Collision Algorithms

• The sequence of locations they generate is called a “collision path”

• Two types
  – **Open addressing** collision algorithms
    • Used when each location in the primary storage area references a single node
  – Non-open addressing collision algorithms
    • Used when each location in the primary storage area can reference multiple nodes
      – A reference to an array of nodes
      – A reference to a linked list of nodes
    • Often used in dynamic implementations
Open Addressing Collision Algorithms

• The **Linear**, **Quadratic**, and **Linear-Quotient** algorithms are open addressing algorithms
  – The Linear-Quotient is the best of these three

• **Bad collision algorithms**
  – Synonyms (keys that hash into the same address) follow the same collision path, called **clustering**;
    • Sequential paths: **primary clustering**
    • Non-sequential paths: **secondary clustering**
  – Generate a location twice before every location is generated once, called **repetitive addressing**
Linear and Quadratic Collision algorithms

- Linear algorithm: \( i_p = (i_p + 1) \mod N \)
- Quadratic algorithm: \( i_p = (i_p + p^2) \mod N \)

where: \( N \) is the array size,
\( p \) is the pass number though the collision algorithm

- The linear algorithm exhibits *primary* clustering
- The quadratic algorithm exhibits *secondary* clustering and *repetitive addressing*
Linear and Quadratic Collision Paths

Array size, N, = 19; hashed ip = 4

Collision paths for pass 1, 2, ...

Linear: \( \text{ip} = (4 + 1) \mod 19 = 5; \quad \text{ip} = (5 + 1) \mod 19 = 6; \ldots \)

Quadratic: \( \text{ip} = (4 + 1 \times 1) \mod 19 = 5; \quad \text{ip} = (5 + 2 \times 2) \mod 19 = 9; \ldots \)

Repetitive addresses: in red; Clustering: same path for all synonyms of 4

<table>
<thead>
<tr>
<th>Pass Number, ( p )</th>
<th>Collision Algorithm</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear</td>
<td>Quadric</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Linear-Quotient Collision Algorithm
For an array of Size $N$

- No repetitive addressing (multiple accesses)
- Minimal secondary clustering
  - only when $q \% N$ is the same for two synonyms (probably $< 1/(N-1)$)
- Minimal primary clustering
  - only when $q == 1$ (probability $1/(N-1)$)

\[
ip = (\text{ip} + \text{offset}) \mod N
\]
where: 
- $\text{offset} = q \textbf{if} (q\%N \neq 0) \textbf{; else} \text{offset} = a 4k+3 \text{ prime}$
- $N$ is the array size, $q = pk / N$

An Example
Linear-Quotient Collision Path For 3 Synonyms

\[ N = 19, \ ip = 4, \ 4k+3 \ prime = 23 \]

**Case 1**, \( pk = 593 \):
\[ q = 593/19 = 31; \quad q \% 19 = 31 \% 19 = 12; \quad i_p = (i_p + 31) \% 19 \]

**Case 2**, \( pk = 5058 \):
\[ q = 5058/19 = 266; \quad q \% 19 = 266 \% 19 = 0; \quad i_p = (i_p + 23) \% 19 \]

**Case 3**, \( pk = 251 \):
\[ q = 251/19 = 13; \quad q \% 19 = 13 \% 19 = 13; \quad i_p = (i_p + 13) \% 19 \]

<table>
<thead>
<tr>
<th>Pass</th>
<th>Case 1, ( pk = 593 )</th>
<th>Case 2, ( pk = 5058 )</th>
<th>Case 3, ( pk = 251 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( i_p = 16 )</td>
<td>( i_p = 8 )</td>
<td>( i_p = 17 )</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>
The Delete Problem

Occurs When A Synonym Is Deleted

Consider an initialized structure. All of primary storage is null.

- Nodes C and D are synonyms
  - Both hash into ip = 4

- C is inserted first
  - A reference to C is stored in data[4]

- D is then inserted
  - Collides with C (for ip = 4), collision algorithm generates ip = 9
  - A reference to D is stored in data[9]

- C is deleted, data[4] is set to null

- D is to be fetched
  - It hashes into ip = 4, but data[4] is null
  - B is assumed to be not in the structure, but it still is
Solution To The Delete Problem

“Remember” That A Synonym Was Deleted

• When Deleting a node
  – Its reference is not set to \texttt{null}, but to the address of a dummy node, \(v_2\)
• \(v_2\) indicates a node was deleted from this reference
• \texttt{null} indicates there was never a node at this reference
Solution To The Delete Problem (continued)

• The Fetch and Delete operations
  – Continue along the collision path ignoring \( v_2 \) references until
    • The node is found, or
    • The element of primary storage contains a \texttt{null}

• The Insert operation continues searching primary storage until a value of \texttt{null} or \( v_2 \) is found
  – Overwriting \( v_2 \) references reclaims the garbage
The LQHashed Data Structure

• Runs at the optimum loading factor
  – Its primary storage size, N, is the first 4k+3 prime above \( n / 0.75 \) (n is the maximum nodes to be stored)

• Its operation algorithms use
  – Folding preprocessing, the Division Hashing function, the Linear-Quotient collision algorithm, and the delete problem solution

• Its implementation is fairly straightforward

• Its performance is always excellent
Operation Algorithms Of the LQHashed Structure

• The operation algorithms are adaptations of the generalized algorithm
  – **Insert** inserts when a **null** or \( v_2 \) is found
  – **Fetch** and **Delete** are very similar
    • Both end unsuccessful when a **null** is found
• **Update** is the Delete algorithm followed by the Insert operation
Hashed Structure Operation Algorithm

Preprocessing algorithm generates pseudo key, pk from the given key, k

hashing algorithm generates the array location, 
\[ i_p = h(pk) \]

Correct location? 

Yes 

No 

collision algorithm generates new location

perform operation on node at \( i_p \)

end
LQHashed Structure’s Insert Algorithm

Fold preprocess key, k, into pseudo key, pk

- pass = 0
- q = pk / N
- offset = q
- ip = pk % N

- q % N == 0?
  - yes
    - offset = 9,967 (a 4k+3 prime)
  - no
    - pass < N?
      - yes
        - data[ip] == v1 or v2
          - yes
            - insert node at data[ip]
          - no
            - return noError = true and end
        - no
          - return noError = false and end
      - no
        - ip = (ip + offset) % N
          - pass = pass + 1

N is the size of the primary storage area array, data

V1 is null, V2 is the dummy node
LQHashed Structure’s Fetch Algorithm

1. Fold preprocess key, \( k \), into pseudo key, \( pk \)
   
   \[
   \text{pass} = 0; \quad q = pk / N \\
   \text{offset} = q; \quad ip = pk \% N
   \]

2. \( q\%N = 0? \)
   
   - yes
     - \( \text{offset} = 9,967 \) (a \( 4k+3 \) prime)
   
   - no
     - \( \text{pass} < N? \)
       
       - yes
         - \( \text{data}[ip] = v_1 \)
         
         - yes
           - return null and end
         
         - no
           - data[ip]'s key == \( k \)
             
             - yes
               - return node at data[ip] and end
             
             - no
               - \( \text{ip} = (ip + \text{offset}) \% N \)
                 
                 - \( \text{pass} = \text{pass} + 1 \)
LQHashed Structure’s Delete Algorithm

Fold preprocess key, k, into pseudo key, pk

\[
\begin{align*}
\text{pass} & = 0; \quad q = \frac{pk}{N} \\
\text{offset} & = q; \quad ip = pk \mod N
\end{align*}
\]

\[q\%N = 0?\]

\[
\begin{align*}
\text{yes} & \quad \text{offset} = 9,967 \text{ (a } 4k+3 \text{ prime)} \\
\text{no} & \quad \text{ip} = (ip + \text{offset}) \mod N; \quad \text{pass} = \text{pass} + 1
\end{align*}
\]

\[\text{pass} < N?\]

\[
\begin{align*}
\text{yes} & \quad \text{data[ip]} = v_1 \\
\text{no} & \quad \text{data[ip]’s key} = k
\end{align*}
\]

\[\text{yes} \quad \text{return noError} = \text{false} \text{ and end}
\]

\[\text{no} \quad \text{data[ip]} = v_2; \quad \text{return true} \text{ and end}
\]


V_1 \text{ is null, V}_2 \text{ is the dummy node}
Implementation
Of the LQHashed Structure

• The operation methods are the Java equivalent of the operation flowcharts

• Methods are included to
  – Generate a $4k+3$ prime a given percent larger than a given integer
  – Output all the nodes
  – Perform Folding preprocessing on a String key

• A one parameter constructor is included
  – Passed the maximum number of nodes
  – Invokes the $4k+3$ prime generator to size the array
  – Create the dummy node, named deleted

• The client’s node definition class (e.g., Listing) must provide a getKey method
public static int fourKPlus3(int n, int pct) {
    boolean fkp3 = false;
    boolean aPrime = false;
    int prime, highDivisor, d;
    double pctd = pct;

    prime = (int)(n * (1.0 + (pctd / 100.0)));
    if (prime % 2 == 0) // if even make odd
        prime = prime + 1;

    while(fkp3 == false) // not a 4k+3 prime
        {
            while(aPrime == false) // not a prime
            {
                highDivisor = (int)(Math.sqrt(prime) + 0.5);
                for(d = highDivisor; d > 1; d--)
                {
                    if (prime % d == 0)
                        break; // not a prime
                }
                if (d != 1) // prime not found
                    prime = prime + 2;
                else
                    aPrime = true;
            }
        }
    if((prime - 3) % 4 == 0)
        fkp3 = true;
    else
        {
            prime = prime + 2;
            aPrime = false;
        }
    }
    // end of 4k+3 prime search loop
    return prime;
}
Method To Output All The Nodes in the LQHashed Structure

```java
public void showAll()
// deleted is the reference to the dummy node
{
    for (int i = 0; i < N; i++)
        if (data[i] != null && data[i] != deleted)
            data[i].toString();
} // end showAll method
```
Folding Preprocessing Method

1. **public static int** stringToInt(String aKey) // from Figure 5.18
2. {
3.   int pseudoKey = 0;
4.   int n = 1;
5.   int cn = 0;
6.   char c[] = aKey.toCharArray();
7.   int grouping = 0;
8.   while (cn < aKey.length()) // still more character in the key
9.     {
10.    grouping = grouping << 8; // pack next four characters
11.    grouping = grouping + c[cn];
12.    cn = cn + 1;
13.    if (n == 4 || cn == aKey.length()) // 4 characters are processed
14.       // or no more characters
15.       {
16.          pseudoKey = pseudoKey + grouping; // add grouping to pseudokey
17.          n = 0;
18.          grouping = 0;
19.       }
20.     }
21.   n = n + 1;
22. } // end while
23. return Math.abs(pseudoKey);
24. } // end stringToInt method
The LQHashed Structure’s Constructor

1. public LqHashed(int length)
2.     // loadingFactor is a data member initialized to 0.75
3.     { int pct = (int)((1.0 / loadingFactor - 1) *100.0);
4.         N = fourKPlus3(length, pct);   // size of the array
5.         data = new Listing[N];   // allocate primary storage
6.         deleted = new Listing("","","");  // the dummy node
7.         for(int i = 0; i < N; i++)   // initialize primary storage
8.             data[i] = null;
9.     } // end of constructor
Performance of The LQHashed Structure

• **Speed**
  – Average speed is $O(1)$
  – Only slightly slower than the Direct Hashed structure
  – A problem could slow down Fetch and Delete

• **Density**
  – Usually better than the Direct Hashed structure because of its 0.75 loading factor

• **Overall performance** is excellent
Speed Of The LQHashed Structure

• Each operation
  – uses folding preprocessing on an m character key
    • requires m + 2 memory accesses
  – Averages 4 passes through the collision loop
• Each pass through the collision loop
  – **Insert** makes 1 access to fetch data[ip]
    • Total for insert: m + 2 + 4 = m + 6 accesses
  – **Fetch and Delete** make 2 accesses each
    • One to fetch data[ip], and one to fetch the node’s key
    • Total for Insert or Delete: m + 2 + 8 = m + 10 accesses
A Possible Speed Problem With The LQHashed Structure’s Fetch and Delete

• Develops when lots of nodes have been deleted
  – Leaves the structure with most of the array storing the dummy node’s address, v2
  – Gives the appearance of long collision paths (Fetch and Delete do not terminate on a v2 reference)

• Remedy: total the number of V2 references in the structure (or search length), when excessive
  – Suspend operations and reinsert all nodes into a new initialized array
Algorithm to Remedy the LQHashed Structure’s Fetch and Delete Speed Problem

1. temp = data; // set temp referencing primary storage
2. data = new Listing[N]; // allocate a new array data
3. for (int i = 0; i < N; i++)
4. {
   if(temp[i] != null && temp[i] != v2)
5.      insert(temp[i]); // insert nodes into array data
6. }
7. V2count = 0; // no dummy node references in the array
8. temp = null; // recycle the old array
9. // end of delete problem remedy
Density Of The LQHashed Structure

- Density = information bytes / total bytes
  - Information bytes = n * w
    - n is the number of nodes, w is the bytes per node
  - Overhead = 4N
    - N is the array size, 4 bytes per array element
- Density = n * w / (n * w + 4N)
  = 1 / (1 + 4 N/(n*w)) = 1 / ( 1 + 4 / (l*w))
  - l = n/N is the Loading Factor
- Density = 1 / ( 1 + 5.33 / w) for l ≈ 0.75
  - Density > 0.8 for node widths > 23 bytes per node
Variation In Density with Node Width For the LQHashed Structure
## Performance Of The LQHashed Structure

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Insert</th>
<th>Delete</th>
<th>Fetch</th>
<th>Update = Delete + Insert</th>
<th>Average $^{[1]}$</th>
<th>Big-O Average</th>
<th>Average for $n = 10^7$</th>
<th>Condition for Density $&gt; 0.8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted-Optimized Array</td>
<td>3</td>
<td>$\leq n$</td>
<td>$\leq n$</td>
<td>$\leq n + 3$</td>
<td>$(3n+6)/4 = 0.75n + 1.5$</td>
<td>O(n)</td>
<td>0.75x$10^7 + 1.5$</td>
<td>w &gt; 16</td>
</tr>
<tr>
<td>Stack and Queue</td>
<td>5</td>
<td>combined with fetch</td>
<td>4.5</td>
<td>not supported</td>
<td>$9.5/2 = 5$</td>
<td>O(1)</td>
<td>5</td>
<td>w &gt; 16</td>
</tr>
<tr>
<td>Singly Linked List</td>
<td>6</td>
<td>1.5n</td>
<td>1.5n</td>
<td>1.5n + 6</td>
<td>$(4.5n+12)/4 = 1.13n + 3$</td>
<td>O(n)</td>
<td>1.13x$10^7 + 3$</td>
<td>w &gt; 33</td>
</tr>
<tr>
<td>Direct Hashed (with pre-processing)</td>
<td>1 or (3)</td>
<td>2 or (4)</td>
<td>1 or (3)</td>
<td>3 or (7)</td>
<td>$7/4 = 1.75$ or $(17/4 = 4.25)$</td>
<td>O(1)</td>
<td>1.75 or (4.25)</td>
<td>w*1 &gt; 16</td>
</tr>
<tr>
<td>LQHashed</td>
<td>m+6</td>
<td>m + 10</td>
<td>m + 10</td>
<td>2m+16</td>
<td>$(5m+42)/4$</td>
<td>O(1)</td>
<td>1.25m + 11</td>
<td>w &gt; 23</td>
</tr>
</tbody>
</table>

$^{[1]}$ assumes all operations equally probable; w is node width, l is the loading factor; m is the key length
Generic Hashed Structures

• Use the previously discussed
  – *Generic coding guidelines* to code the node definition and the data structure class
  – *Four-step methodology* to convert the data structure class to a generic implementation

• Separate the preprocessing method from the data structure class
  – Code it in the key’s class
  – Typically its signature is: `public int hashcode()`
  – Java’s String and wrapper classes code this method

• The data structure’s preprocessing becomes:
  ```java
  int pk = newListing.getKey().hashCode();
  ```
Dynamic Hashed Structures

• Can expand beyond an initially predicted maximum number of nodes

• Various schemes include
  – Expanding the primary storage area array’s size
  – Providing additional storage external to the array
  – A hybrid approach (Java’s Hashtable class)
Expanded External Storage Approach To Dynamic Hashed Structures

• External storage is a set of linked lists
  – Each primary storage element is a linked list header
  – The keys of all the nodes in a given linked list are synonyms (all map into the same element)

• Collisions are resolved during
  – An Insert operation by adding a node to the front of a linked list (expanding the structure)
  – Fetch and Delete operations by a sequential search through the key’s linked list

• The performance is typically very good
Performance Of A Linked Dynamic Hashed Structure

• **Speed** is excellent, $O(1)$
  – Slightly faster than the LQHashed structure assuming the linked lists are balanced

• **Density** is good, but not as good as the LQHashed structure
  – Density $\geq 0.8$ for node width $> 37$

• Still, the overall performance is excellent, and it is a dynamic structure
Speed Of A Linked Hashed Structure

- Each operation
  - uses folding preprocessing on an \textbf{m character key}
    - requires \( m \) memory accesses
- Insert
  - requires 5 additional memory access to insert at the front of a linked list
  - Total is \( m + 5 \)
- Fetch and Delete
  - require an additional 3 accesses for \( n/N = 3 \)
  - Total is \( m + 3 \)
    - on average, assuming linked lists are balanced
    - \( n \) is the maximum number of nodes stored
    - \( N \) is the size of the primary storage area array
Density Of A Linked Hashed Structure

- Density = \( \frac{1}{(1 + \frac{8}{w}) + \left(\frac{1}{l_p}\right) \times \left(\frac{4}{w}\right)} \)
  - \( l_p \) is \( n/N \) (a pseudo loading factor)
  - \( w \) is the node width in bytes

![Linked Hash Structure](chart.png)
### Linked Hashed Structure’s Performance

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Insert</th>
<th>Delete</th>
<th>Fetch</th>
<th>Update = Delete + Insert</th>
<th>Average[^1^]</th>
<th>Big-O</th>
<th>Average for n = 10^7</th>
<th>Condition for Density &gt; 0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted-Optimized Array</td>
<td>3</td>
<td>≤ n</td>
<td>≤ n</td>
<td>≤ n +3</td>
<td>(3n+6)/4 = 0.75n + 1.5</td>
<td>O(n)</td>
<td>0.75x10^7 + 1.5</td>
<td>w &gt; 16</td>
</tr>
<tr>
<td>Stack and Queue</td>
<td>5</td>
<td>combined with fetch</td>
<td>4.5</td>
<td>not supported</td>
<td>9.5/2 = 5</td>
<td>O(1)</td>
<td>5</td>
<td>w &gt; 16</td>
</tr>
<tr>
<td>Singly Linked List</td>
<td>6</td>
<td>1.5n</td>
<td>1.5n</td>
<td>1.5n +6</td>
<td>(4.5n+12)/4 = 1.13n + 3</td>
<td>O(n)</td>
<td>1.13x10^7 + 3</td>
<td>w &gt; 33</td>
</tr>
<tr>
<td>Direct Hashed (with pre-processing)</td>
<td>1 or (3)</td>
<td>2 or (4)</td>
<td>1 or (3)</td>
<td>3 or (7)</td>
<td>7/4 = 1.75 or (17/4 = 4.25)</td>
<td>O(1)</td>
<td>1.75 or (4.25)</td>
<td>w*1 &gt; 16</td>
</tr>
<tr>
<td>LQHashed</td>
<td>m+6</td>
<td>m + 10</td>
<td>m + 10</td>
<td>2m+16</td>
<td>(5m+42)/4</td>
<td>O(1)</td>
<td>1.25m + 11</td>
<td>w &gt; 23</td>
</tr>
<tr>
<td>Linked hashed lp=3</td>
<td>m+5</td>
<td>m + 3</td>
<td>m + 3</td>
<td>2m+8</td>
<td>(5m+19)/4</td>
<td>O(1)</td>
<td>1.25m + 5</td>
<td>w &gt; 37</td>
</tr>
</tbody>
</table>

[^1^] assumes all operations equally probable; w is node width, l is the loading factor; m is the key length
Java’s Hashtable Class

• In the package java.util
  • http://java.sun.com/j2se/1.3/docs/api/java/util/Hashtable.html

• A hybrid dynamic hashed structure
  – Uses expandable linked lists to resolve collisions
  – Primary storage also expands when the loading factor exceeds a specified maximum

• An unencapsulated generic structure

• Key can be any type of object
  – Key class should overwrite the method equals and hashCode (the String and wrapper classes do this)
  – Hashtable’s operation methods invoke these methods
Use of the HashTable Class

1. **public static void** main(String[] args)
2. {
3.     Hashtable <String, Listing> dataBase = **new** Hashtable<String, Listing>();
4.     Listing b, t;
5.     Listing bill = **new** Listing("Bill", "1st Avenue", "999 9999");
6.     Listing tom = **new** Listing("Tom", "2nd Avenue", "456 8978");
7.     Listing newBill = **new** Listing("William", "99th Street", "123 4567");
8.     // inserts
9.     dataBase.put("Bill", bill);
10.    dataBase.put("Tom", tom);
11.    // fetches
12.    b = dataBase.get("Bill");    t = dataBase.get("Tom");
13.    System.out.println(b, + "\n" t);
14.    // effectively an update of Bill's address
15.    dataBase.put("Bill", newBill);      b = dataBase.get("Bill");  // fetches
16.    System.out.println(b);
17.    // demonstration of the lack of encapsulation. Client can change node contents
18.    newBill.setAddress("18 Park Avenue");
19.    b = dataBase.get("Bill");
20.    System.out.println(b);
21.    // delete operation
22.    dataBase.remove("Bill");   b = dataBase.get("Bill");
23.    System.out.println(b)
24. }
Return to, Overview, primary storage area, hashing functions, perfect hashed structures, non-perfect hashed structures, generic conversion, dynamic hashed structures, Java’s Hashtable class