Maps

- A map models a searchable collection of key-value entries
- Characteristics
  - The main operations: search, inserting, and deleting items
  - Multiple entries with the same key are not allowed.
- Applications
  - Address book
  - Student-record database

### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>∅</td>
</tr>
<tr>
<td>put(5, A)</td>
<td>null</td>
<td>(5, A)</td>
</tr>
<tr>
<td>put(7, B)</td>
<td>null</td>
<td>(5, A), (7, B)</td>
</tr>
<tr>
<td>put(2, C)</td>
<td>null</td>
<td>(5, A), (7, B), (2, C)</td>
</tr>
<tr>
<td>put(8, D)</td>
<td>null</td>
<td>(5, A), (7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>get(7)</td>
<td>C</td>
<td>(5, A), (7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>get(4)</td>
<td>null</td>
<td>(5, A), (7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>get(2)</td>
<td>E</td>
<td>(5, A), (7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>size()</td>
<td>4</td>
<td>(5, A), (7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>remove(5)</td>
<td>A</td>
<td>(7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>remove(2)</td>
<td>E</td>
<td>(7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>get(2)</td>
<td>null</td>
<td>(7, B), (8, D)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>false</td>
<td>(7, B), (8, D)</td>
</tr>
</tbody>
</table>
A Simple List-Based Map

- We can efficiently implement a map using an unsorted list.
- We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order.

Methods for Map

- get(key) algorithm

  Algorithm get(k):
  B = S.positions()
  while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k then
      return p.element().getValue()
  return null

- put(key, value) algorithm

  Algorithm put(k,v):
  B = S.positions()
  while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k then
      t = p.element().getValue()
      S.set(p,(k,v))
      return t
  S.addLast((k,v))
  n = n + 1
  return null

- remove(k) Algorithm

  Algorithm remove(k):
  B = S.positions()
  while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k then
      t = p.element().getValue()
      S.remove(p)
      n = n - 1
  return t
  return null

Performance of a List-Based Map

- Time complexity
  - The put operation takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence.
  - The get and remove operations take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key.
  - In our implementation, put also takes $O(n)$ time.

- The unsorted list implementation is effective only
  - For maps of small size or
  - For maps in which puts are the most common operations while searches and removals are rarely performed.
  - This is not true in our implementation, either.
Problems and Solution

• Problems
  - We have lots of data to store.
  - We desire efficient performance, $O(1)$, for insertion, deletion and searching.
  - Too much (wasted) memory is required if we use an array indexed by the data’s key.

• Solution
  - Hashing

Hashing

• Hash function
  - A mathematical function $h$ that, given a key $k$, gives an index $h(k)$ into the hash table (an array), where the item with key $k$ can be found.

• Case
  - Integer keys in the range 1, ..., 32000.
  - Array indices are 0, ..., 249.
  - No more than 250 keys are used altogether.

• Compression map
  - Trivial hash function: $h(k) = k$, which is not plausible
  - We require a hash function that will scale the key down to the allowable range 0, ..., 249.
  - Try $h(k) = k \mod 250$.

Hash Functions and Hash Tables

• A hash function $h$ maps
  - keys of a given type to integers in a fixed interval $[0, N - 1]$.
  - Example: $h(x) = k \mod N$

• Hash value
  - The integer $h(x)$ is called the hash value of key $x$

• A hash table for a given key type consists of
  - Hash function $h(x)$
  - Array (called table) of size $N$
  - When implementing a map with a hash table, the goal is to store item (key, value) at index $i = h(x)$
**Hash Functions**

- Typically specified as the composition of two functions:
  - Hash code $h_1$: key $\rightarrow$ integer
  - Compression function $h_2$: integer $\rightarrow [0, N - 1]$
- Hash function
  - The hash code is applied first.
  - The compression function is applied next on the result.
  - Example: $h(x) = h_2(h_1(x))$
- The goal of the hash function
  - “Disperse” the keys in an apparently random way

**Hash Codes**

- Memory address
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum
  - We partition the bits of the key into components of fixed length (e.g., 8, 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

**Hash Codes (cont.)**

- Polynomial accumulation
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) as $a_0 a_1 ... a_{n-1}$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + ... + a_{n-1} z^{n-1}$$
    at a fixed value $z$, ignoring overflows.
  - Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner’s rule:
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i} + z p_{i-1}(z), (i = 1, 2, ..., n - 1)$$
    $$p(z) = p_{n-1}(z)$$
  - Especially suitable for strings: The choice of $z = 33$ gives at most 6 collisions on a set of 50,000 English words
Compression Functions

- Division:
  - \( h_2(y) = y \mod N \)
  - The size \( N \) of the hash table is usually chosen to be a prime to minimize collision.
- Multiply, Add and Divide (MAD):
  - \( h_2(y) = (ay + b) \mod N \)
  - \( a \) and \( b \) are nonnegative integers such that \( a \mod N \neq 0 \).
  - Otherwise, every integer would map to the same value \( b \).

Collision Handling

- Collisions occur when different elements are mapped to the same cell.
- Separate Chaining
  - Let each cell in the table point to a linked list of entries that map there.
  - Simple, but requires additional memory outside the table.

Map with Separate Chaining

Algorithm get(k):
return A[h(k)].get(k)

Algorithm remove(k):
t = A[h(k)].remove(k)
if t \neq null then // k was found
    n = n - 1
return t

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = null then // k is a new key
    n = n + 1
return t

Linear Probing

- Open addressing
  - The colliding item is placed in a different cell of the table
- Linear probing
  - Handles collisions by placing the colliding item in the next (circularly) available table cell
  - Each table cell inspected is referred to as a “probe”
  - Colliding items lump together, causing future collisions to cause a longer sequence of probes.
- Example:
  - \( h(x) = x \mod 13 \)
  - Insert keys in the following order: 18, 41, 22, 44, 59, 32, 31, 73
Search with Linear Probing

- **get(key)** method
  - Consider a hash table A that uses linear probing
  - We start at cell \( h(x) \)
  - We probe consecutive locations until one of the following occurs:
    - An item with key is found, or
    - An empty cell is found, or
    - All cells have been unsuccessfully probed

  \[
  \text{Algorithm } \text{get}(k) \nn  i \leftarrow h(k) \nn  p \leftarrow 0 \nn  \text{repeat} \nn  \quad c \leftarrow A[i] \nn  \quad \text{if } c = \emptyset \nn  \quad \quad \text{return } \text{null} \nn  \quad \text{else if } c.getKey() = k \nn  \quad \quad \text{return } c.getValue() \nn  \quad \text{else} \nn  \quad \quad i \leftarrow (i + 1) \mod N \nn  \quad \quad p \leftarrow p + 1 \nn  \text{until } p = N \nn  \text{return } \text{null}
  \]

Updates with Linear Probing

- **put(key, value)** method
  - We throw an exception if the table is full.
  - We start at cell \( h(x) \)
  - We probe consecutive cells until one of the following occurs:
    - A cell is found that is either empty or stores AVAILABLE, or
    - All cells have been unsuccessfully probed.
  - We store \((key, value)\) in the found empty or AVAILABLE cell.

Double Hashing

- **Double hashing**
  - Uses a secondary hash function \( d(k) \)
  - Handles collisions by placing an item in the first available cell of the series \((i + jd(k)) \mod N\) for \(j = 0, 1, \ldots, N - 1\).
  - The secondary hash function \(d(k)\) cannot have zero values.
  - The table size \(N\) must be a prime to allow probing of all the cells.
  - Common choice of compression function for the secondary hash function:
    - \(d_2(k) = q - k \mod q\) where \(q < N\) and \(q\) is a prime
    - The possible values for \(d_2(k)\) are \(1, 2, \ldots, q\).
Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - \( N = 13 \)
  - \( h(k) = k \mod 13 \)
  - \( d(k) = 7 - k \mod 7 \)
- Example
  - Insert keys in the following order: 18, 41, 22, 44, 59, 32, 31, 73

<table>
<thead>
<tr>
<th>( k )</th>
<th>( h(k) )</th>
<th>( d(k) )</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Performance of Hashing

- Worse case time complexity on hash table
  - Insertion: \( O(n) \)
  - Removal: \( O(n) \)
  - The worst case occurs when all the keys inserted into the map collide.
- Our expectation
  - The load factor \( \alpha = \frac{n}{N} \) affects the performance of a hash table.
  - Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is \( \frac{1}{1 - \alpha} \).
- Expected running time
  - All operations in a hash table: \( O(1) \)
  - In practice, hashing is very fast provided the load factor is not close to 100%. 