Data Structures in Java

Lecture 12: Introduction to Hashing.

10/19/2015

Daniel Bauer
Homework

• Due Friday, 11:59pm.

• Jarvis is now grading HW3.
Recitation Sessions

• Recitations this week:
  • Review of balanced search trees.
  • Implementing AVL rotations.
  • Implementing maps with BSTs.
  • Hashing (Friday/Next Mon & Tue).
Midterm

• **Midterm next Wednesday (in-class)**
  
  • Closed books/notes/electronic devices.
  
  • Ideally, bring a pen, water, and nothing else.
  
  • 60 minutes
  
  • Midterm review this Wednesday in class.
How to Prepare?

• Midterm will cover all content up to (and including) this week.
  • Know all ADTs, operations defined on them, data structures, running times.
  • Know basics of running time analysis (big-O).
  • Understand recursion, inductive proofs, tree traversals, …
• Practice questions out today. Discussed Wednesday.
• Good idea to review slides & homework!
How to Prepare Even More?

• Optional:
  
  • Solve Weiss textbook exercises and discuss on Piazza.
  
  • Try to implement data structures from scratch.
Map ADT

• A map is collection of \((key, value)\) pairs.

• Keys are unique, values need not be (keys are a Set!).

• Two operations:
  
  • \text{get(key)} \quad \text{returns the value associated with this key}
  
  • \text{put(key, value)} \quad \text{(overwrites existing keys)}

\begin{tabular}{|c|c|}
  \hline
  key1 & value1 \\
  key2 & value2 \\
  key3 & value3 \\
  key4 & \\
  \hline
\end{tabular}
Implementing Maps
Implementing Maps

• Option 1: Use any set implementation to store special (key,value) objects.

• Comparing these objects means comparing the key (testing for equality or implementing the Comparable interface)
Implementing Maps

• Option 1: Use any set implementation to store special (key, value) objects.
  • Comparing these objects means comparing the key (testing for equality or implementing the Comparable interface)

• Option 2: Specialized implementations
  • B+ Tree: nodes contain keys, leaves contain values.
  • Plain old Array: Only integer keys permitted.
  • Hash maps (this week)
Balanced BSTs

- Runtime of BST operations (insert, contains/find, remove, findmin, findmax) depend on height of the tree.

- Balance condition: Guarantee that the BST is always close to a complete binary tree.
  - Then the height of the tree will be $O(\log N)$.
  - All BST operations will run in $O(\log N)$.
  - Map operations get and put will also run in $O(\log N)$.

Can we do better?
Arrays as Maps

- When keys are integers, arrays provide a convenient way of implementing maps.

- Time for `get` and `put` is $O(1)$.
Hash Tables

• Define a table (an array) of some length $TableSize$.

• Define a function $\text{hash(key)}$ that maps key objects to an integer index in the range $0 \ldots TableSize - 1$
Hash Tables

• Define a table (an array) of some length $\text{TableSize}$. 

• Define a function $\text{hash(key)}$ that maps key objects to an integer index in the range $0 \ldots \text{TableSize} - 1$
Hash Tables

- **Lookup/get**: Just hash the key to find the index.
- **Assuming** $\text{hash(key)}$ **takes constant time**, **get and put run in $O(1)$**.
Hash Table Collisions

• Problem: There is an infinite number of keys, but only TableSize entries in the array.

• How do we deal with collisions? (new item hashes to an array cell that is already occupied)

• Also: Need to find a hash function that distributes items in the array evenly.
Choosing a Hash Function

- Hash functions depend on: type of keys we expect (Strings, Integers...) and $TableSize$.

- Hash functions need to:
  - Spread out the keys as much as possible in the table (ideal: uniform distribution).
  - Make sure that all table cells can be reached.
Choosing a Hash Function: Integers

- If the keys are integers, it is often okay to assume that the possible keys are distributed evenly.

\[ \text{hash}(x) = x \% \text{TableSize} \]

```java
class HashFunction {
    public static int hash(Integer key, int tableSize) {
        return key % tableSize;
    }
}
```

e.g. TableSize = 5
hash(0) = 0, hash(1) = 1,
hash(5) = 0, hash(6) = 1
Choosing a Hash Function: Strings - Idea 1

- Idea 1: Sum up the ASCII (or Unicode) values of all characters in the String.

```java
public static int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++)
        hashVal = hashVal + key.charAt(i);
    return hashVal % tableSize;
}
```

e.g. “Anna” → 65 + 2 · 110 + 97 = 382
A → 65, n → 110, a → 97
Choosing a Hash Function: Strings - Problems with Idea 1

- Idea 1 doesn’t work for large table sizes:
  - Assume $TableSize = 10,007$
  - Every character has a value in the range 0 and 127.
  - Assume keys are at most 8 chars long:
    - $hash(key)$ is in the range 0 and $127 \cdot 8 = 1016$.
    - Only the first 1017 cells of the array will be used!
Choosing a Hash Function: Strings - Problems with Idea 1

- Idea 1 doesn’t work for large table sizes:

  - Assume $\text{TableSize} = 10,007$

  - Every character has a value in the range 0 and 127.

  - Assume keys are at most 8 chars long:

    - hash(key) is in the range 0 and $127 \cdot 8 = 1016$.

    - Only the first 1017 cells of the array will be used!

- Also: All anagrams will produce collisions: “rescued”, “secured”, ”seducer”
Choosing a Hash Function: Strings - Idea 2

- Idea 2: Spread out the value for each character

```java
public static int hash(Integer key, int tableSize) {
    return (key.charAt(0) +
             27 * key.charAt(1) +
             27 * 27 * key.charAt(2));
}
```
Choosing a Hash Function: Strings - Idea 2

• Idea 2: Spread out the value for each character

```java
public static int hash( Integer key, int tableSize ) {
    return (key.charAt(0) +
            27 * key.charAt(1) +
            27 * 27 * key.charAt(2));
}
```

• Problem: assumes that the all three letter combinations (trigrams) are equally likely at the beginning of a string.

• This is not the case for natural language
  • some letters are more frequent than others
  • some trigrams ( e.g. “xvz”) don’t occur at all.
Choosing a Hash Function: Strings - Idea 3

```java
public static int hash( String key, int tableSize ) {
    int hashVal = 0;
    
    for( int i = 0; i < key.length(); i++ )
        hashVal = 37 * hashVal + key.charAt( i );
    
    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;
    
    return hashVal;
}
```

\[key[N - 1] \cdot 37^N + key[N - 2] \cdot 37^{N-1} + \cdots + key[1] \cdot 37 + key[0]\]

This is what Java Strings use; works well, but slow for large strings.
Combining Hash Functions

• In practice, we often write hash functions for some container class:
  • Assume all member variables have a hash function (Integers, Strings…).
  • Multiply the hash of each member variable with some distinct, large prime number.
  • Then sum them all up.
Combining Hash Functions, Example

```java
public class Person {
    public String firstName;
    public String lastName;
    public Integer age;
}
```
Combining Hash Functions, Example

```java
public class Person {
    public String firstName;
    public String lastName;
    public Integer age;
}

public static int hash(Person key, int tableSize) {
    int hashVal = hash(key.firstName, tableSize) * 127 +
    hash(key.lastName, tableSize) * 1901 +
    hash(key.age, tableSize) * 4591;

    hashVal %= tableSize;
    if(hashVal < 0)
        hashVal += tableSize;
}
```
Why Prime Numbers?

- To reduce collisions, TableSize should not be a factor of any large hash value (before taking the modulo).

Bad example:

TableSize = 8  factors = 2, 4, 6, 8, 16
Why Prime Numbers?

- To reduce collisions, `TableSize` should not be a factor of any large hash value (before taking the modulo).

  Bad example:

  \[
  \text{TableSize} = 8 \quad \text{factors} = 2, 4, 6, 8, 16
  \]

- Good practices:
  
  - Keep `TableSize` a prime number.
  
  - When combining hash values, make the factors prime numbers.
What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.

- Need to make sure that objects that are used as keys cannot be changed at runtime (they are immutable)
What Objects Can be Keys?

• Anything can be a key, we just need to find a good hash function.

• Need to make sure that objects that are used as keys cannot be changed at runtime (they are immutable)
  • Otherwise, if their content changes their hash value should change too!
What Objects Can be Keys?

• Anything can be a key, we just need to find a good hash function.

• Need to make sure that objects that are used as keys cannot be changed at runtime (they are immutable)
  • Otherwise, if their content changes their hash value should change too!

• How would you compute the hash value for a LinkedList or a Binary Tree?
Hash Table Collisions

- Problem: There is an infinite number of keys, but only $TableSize$ entries in the array.
- Need to find a hash function that distributes items in the array evenly.
- How do we deal with collisions? (new item hashes to an array cell that is already occupied)

Anna 555-521-2973

TableSize - 1

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash Value</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anna</td>
<td>hash(key)</td>
<td>555-521-2973</td>
</tr>
<tr>
<td>Alice</td>
<td>hash(key)</td>
<td>555-341-1231</td>
</tr>
<tr>
<td>Bob</td>
<td>hash(key)</td>
<td>555-341-1231</td>
</tr>
</tbody>
</table>
Dealing with Collisions: Separate Chaining

- Keep all items whose key hashes to the same value on a linked list.
- Can think of each list as a *bucket* defined by the hash value.

```
0
1
2
...
TableSize - 1
```

```
Alice 555-341-1231
Bob 555-341-1231
```
Dealing with Collisions: Separate Chaining

- To insert a new key in cell that’s already occupied prepend to the list.

.hash(key). Anna 555-521-2973

Alice 555-341-1231

Bob 555-341-1231
Dealing with Collisions: Separate Chaining

- To insert a new key in cell that’s already occupied prepend to the list.
Analyzing Running Time for Separate Chaining (1)

• Time to find a key = time to compute hash function + time to traverse the linked list.

• Assume hash functions computed in O(1).

• How many elements do we expect in a list on average?
• Let $N$ be the number of keys in the table.
• Define the load factor as

$$\lambda = \frac{N}{\text{TableSize}}$$

• The average length of a list is $\lambda$. 

Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed.
Analyzing Running Time for Separate Chaining (2)

- If lookup fails (table miss):
  - Need to search all $\lambda$ nodes in the list for this hash bucket.

- If lookup succeeds (table hit):
  - There will be about $\lambda$ other nodes in the list.
  - On average we search half the list and the target key, so we touch $\lambda/2 + 1$ nodes.

Design rule: keep $\lambda \approx 1$. If load becomes too high increase table size (rehash).
Problems with Separate Chaining

- Requires allocation of new list nodes, which introduces overhead.

- Requires more code because it requires a linked list data structure in addition to the hash table itself.
Hash Tables without Linked Lists: Probing

- When a collision occurs put item in an empty cell of the hash table itself.
Hash Tables without Linked Lists: Probing

- When a collision occurs put item in an empty cell of the hash table itself.
Hash Tables without Linked Lists: Probing

- When a collision occurs put item in an empty cell of the hash table itself.
Hash Tables without Linked Lists: Probing

- When a collision occurs put item in an empty cell of the hash table itself.
Hash Tables without Linked Lists: Probing

- When a collision occurs put item in an empty cell of the hash table itself.

\[ x \% 11 \]

```
0  21
1
2
3
4
5
6
7
8  40
9  18
10 29
11  9
```
Hash Tables without Linked Lists: Probing

- When a collision occurs put item in an empty cell of the hash table itself.
Hash Tables without Linked Lists: Probing

• To look up a key, we search the table, starting from the cell the key was hashed to.

With a Probing Hash Table $\lambda \leq 1$. Table is full if $\lambda = 1$. 

![Diagram of hash table with probing]

29 \rightarrow \text{hash(key)} \rightarrow \begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10
\end{array}

\begin{array}{c}
21 \\
40 \\
18 \\
29 \\
9
\end{array}
Probing: Collision Resolution Strategies (1)

- To insert an item, we probe other table cells in a systematic way until an empty cell is found.

- To look up a key, we probe in a systematic way until the key is found.

- Different strategies to determine the next cell
  - Example: Just try cells sequentially (with wraparound).
Collision Resolution Strategies (2)

• Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is

$$ (\text{hash}(x) + f(i)) \mod TableSize. $$
Collision Resolution Strategies (2)

• Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is

$$ (\text{hash}(x) + f(i)) \% \text{TableSize}. $$

• Linear Probing (previous example):
  • $f(i)$ is some linear function of $i$, usually $f(i) = i$.

If $\text{hash}(x) = 7$, try cell 7 first, then try cell $7+f(1)=8$, cell $7+f(2)=9$, cell $7+f(3)=10$, ...
Collision Resolution Strategies (2)

- Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is $\text{hash}(x) + f(i) \mod \text{TableSize}$.
- Linear Probing (previous example):
  - $f(i)$ is some linear function of i, usually $f(i) = i$.
    
    If $\text{hash}(x) = 7$, try cell 7 first, then try cell $7 + f(1) = 8$, cell $7 + f(2) = 9$, cell $7 + f(3) = 10$, ...

- Quadratic probing $f(i) = i^2$
- Double hashing $f(i) = i \cdot \text{hash}_2(x)$
Linear Probing \( f(i) = i \)

- Can always find an empty cell (if there is space in the table).

- Problem: **Primary Clustering**.
  - Full cells tend to cluster, with no free cells in between.
  - Time required to find an empty cell can become very large if the table is almost full (\( \lambda \) is close to 1).
Linear Probing \( f(i) = i \)

- Can always find an empty cell (if there is space in the table).

- Problem: **Primary Clustering**.

  - Full cells tend to cluster, with no free cells in between.

  - Time required to find an empty cell can become very large if the table is almost full \( (\lambda \text{ is close to } 1) \).
Primary Clustering

40 \rightarrow \text{hash(key)} \rightarrow x \% 11

7 \rightarrow 7 \rightarrow 40

0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10
Primary Clustering

\[x \% 11\]

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

0 1 2 3 4 5 6 7 8 9 10

0 1 2 3 4 5 6 7 8 9 10

40 51
Primary Clustering

• Cells 7-9 are occupied with keys that hash to 7. The entire block is unavailable to keys that hash to k<7.
Primary Clustering

- Cells 7-8 are occupied with keys that hash to 7. The entire block is unavailable to keys that hash to $k<7$. 

$$\text{hash(key)} \equiv x \mod 11$$
Primary Clustering

- Cells 7-8 are occupied with keys that hash to 7. The entire block is unavailable to keys that hash to k<7.
Primary Clustering

• This becomes really bad if $\lambda$ is close to 1

$\frac{x}{11} \mod 11$
Linear Probing vs. Choosing a Random Cell

Figure 5.12  Number of probes plotted against load factor for linear probing (dashed) and random strategy (S is successful search, U is unsuccessful search, and I is insertion)

Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed.
Quadratic Probing

$$\text{hash}(x) + f(i) \pmod{\text{TableSize}}$$

$$f(i) = i^2$$

25 → hash(key) → 3 → 25

$$x \mod 11$$
Quadratic Probing

\[(\text{hash}(x) + f(i)) \mod \text{TableSize}\]

\[f(i) = i^2\]

14 \rightarrow \text{hash(key)}

x \mod 11

3 \rightarrow f(1) = 1

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10 \\
\end{array} \\
\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10 \\
\end{array} \\
\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10 \\
\end{array}
\]

25

3
Quadratic Probing

\[(\text{hash}(x) + f(i)) \mod \text{TableSize}\]

\[f(i) = i^2\]
Quadratic Probing

\[(\text{hash}(x) + f(i)) \% \text{TableSize} \quad f(i) = i^2\]
Quadratic Probing

$$(hash(x) + f(i)) \mod TableSize \quad f(i) = i^2$$

- Primary clustering is not a problem.
Quadratic Probing

- Important: With quadratic probing, TableSize should be a prime number! Otherwise it is possible that we won’t find an empty cell, even if there is plenty of space.

```
19 \rightarrow \text{hash(key)} \rightarrow 3
```

```
i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 \\
---|---|---|---|---|---|---|---|---
3 + f(i) \% 8 | 4 | 7 | 4 | 3 | 4 | 7 | 4 | 3 | ...
```

```
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
0 & 9 \\
1 & 3 \\
2 & 20 \\
3 & 11 \\
\hline
\end{array}
```
Quadratic Probing

- Problem: If the table gets too full ($\lambda > 0.5$), it is possible that empty cells become unreachable, even if the table size is prime.
Quadratic Probing Theorem

If $\text{TableSize}$ is prime, then the first $\frac{\text{TableSize}}{2}$ cells visited by quadratic probing are distinct. Therefore we can always find an empty cell if the table is at most half full.
Quadratic Probing Theorem

If \( TableSize \) is prime, then the first \( \frac{TableSize}{2} \) cells visited by quadratic probing are distinct. Therefore we can always find an empty cell if the table is at most half full.

- Let \( TableSize \) be some prime greater than 3.
- Let \( hash(x) = h \)
- If there was a slot visited twice during the first probing steps, then there must be two numbers \( 0 \leq i < j \leq \frac{TableSize}{2} \) such that

\[
(h + i^2) \mod TableSize = (h + j^2) \mod TableSize
\]
Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first probing steps, then there must be two numbers \( 0 \leq i < j \leq \frac{\text{TableSize}}{2} \) such that

\[
(h + i^2) \mod \text{TableSize} = (h + j^2) \mod \text{TableSize}
\]

\[
h + i^2 = h + j^2
\]

\[
i^2 = j^2
\]

\[
i^2 - j^2 = 0
\]

\[(i - j)(i + j) = 0\]
Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first probing steps, then there must be two numbers \(0 \leq i < j \leq \frac{TableSize}{2}\) such that

\[
(h + i^2) \mod TableSize = (h + j^2) \mod TableSize
\]

\[
h + i^2 = h + j^2
\]

\[
i^2 = j^2
\]

\[
i^2 - j^2 = 0
\]

\[
(i - j)(i + j) = 0
\]

either \((i - j)(i + j) = TableSize\)

or \((i - j) = 0\) or \((i + j) = 0\)
Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first probing steps, then there must be two numbers such that

\[ 0 \leq i < j \leq \frac{TableSize}{2} \]

such that

\[ (h + i^2) \% TableSize = (h + j^2) \% TableSize \]

\[ h + i^2 = h + j^2 \]

\[ i^2 = j^2 \]

\[ i^2 - j^2 = 0 \]

\[ (i - j)(i + j) = 0 \]

impossible because TableSize is prime

either \( (i - j)(i + j) = TableSize \)

or \( (i - j) = 0 \) or \( (i + j) = 0 \)

impossible because \( i < j \)

impossible because \( i < j \leq TableSize/2 \)
Quadratic Probing Theorem (2)

Proof by contradiction:
If there is an index visited twice during the first probing steps, then there must be two numbers such that

\[ 0 \leq i < j \leq \frac{\text{TableSize}}{2} \]

such that

\[ (h + i^2) \mod \text{TableSize} = (h + j^2) \mod \text{TableSize} \]

\[ h + i^2 = h + j^2 \]
\[ i^2 = j^2 \]
\[ i^2 - j^2 = 0 \]
\[ (i - j)(i + j) = 0 \]

The assumption must be false!

impossible because TableSize is prime
impossible because \( i < j \)
impossible because \( i < j \leq \text{TableSize}/2 \)

Contradiction!
Double Hashing

\[ f(i) = i \cdot \text{hash}_2(x) \]

Compute a second hash function to determine a linear offset for this key.

40 \xrightarrow{\text{hash}(\text{key})} x \mod 11

7 \xrightarrow{5 - x \mod 5}

0 1 2 3 4 5 6 7 8 9 10

40
Double Hashing

\[ f(i) = i \cdot hash_2(x) \]

Compute a second hash function to determine a linear offset for this key.

\[ f(1) = 1 \cdot \text{hash}_2(x) = 1 \]
Double Hashing

\[ f(i) = i \cdot \text{hash}_2(x) \]

Compute a second hash function to determine a linear offset for this key.

\[ f(1) = 1 \cdot \text{hash}_2(x) = 3 \]

\[
\begin{align*}
\text{hash(key)} & \quad 62 \\
\text{x \% 11} & \quad 7 \\
\text{hash}_2(key) & \quad 3 \\
5 - x \% 5 & \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10
\end{align*}
\]

\begin{align*}
40 \\
84 \\
62
\end{align*}
Double Hashing

\[ f(i) = i \cdot hash_2(x) \]

Compute a second hash function to determine a linear offset for this key.

\[ f(1) = 1 \cdot hash_2(x) = 1 \]
\[ f(2) = 2 \cdot hash_2(x) = 2 \]
Choosing a Secondary Hash Function

- Need to choose $\text{hash}_2$ wisely!
- What happens with the following function?

```plaintext
hash(key) = x % 11
hash_2(key) = x % 11
```

<table>
<thead>
<tr>
<th>x % 11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
</tr>
</tbody>
</table>
Choosing a Secondary Hash Function

- Need to choose $hash_2$ wisely!
- What will happen with the following function?

\[
\begin{align*}
f(1) &= 1 \cdot hash_2(x) = 0 \\
f(2) &= 2 \cdot hash_2(x) = 0 \\
&\vdots
\end{align*}
\]
Double Hashing

- A good choice for integers is $\text{hash}_2(x) = R - (x \% R)$

- As with quadratic hashing, we need to choose the table size to be prime (otherwise cells become unreachable too quickly).

- Properly implemented, double hashing produces a good distribution of keys over table cells.
Rehashing

• Separate Chaining Hash Tables become inefficient if the load factor becomes too large (lists become too long).

• Hash Tables with Linear Probing become inefficient if the load factor approaches 1 (primary clustering) and eventually fill up.

• Hash Tables with Quadratic Probing and Double Hashing can have failed inserts if the table is more than half full.

• Need to copy data to a new table.
Rehashing

• Allocate a new table of twice the size as the original one.

• For probing hash tables, we cannot simply copy entries to the new array.
  • Different modulo wraparound won’t cause the same collisions.
  • Since the hash function is based on the TableSize, keys won’t be in the correct cell, anyway.

• Remove all N items and re-insert into the new table. This operation takes O(N), but this cost is only incurred in the rare case when rehashing is needed.
Rehashing Running Time

• Remove all N items and re-insert into the new table.
• Every insert is $O(1)$, so rehashing takes $O(N)$.
• But rehashing is relatively rare, we need to do it only after every TableSize/2 inserts.