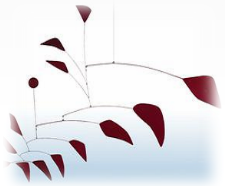
EECS 4101-5101
Advanced Data Structures

Shahin Kamali

Topic 2e - Hash Tables

York University



Picture is from the cover of the textbook CLRS.

Lower bound for search

- The fastest implementations of the dictionary ADT require $O(\log n)$ time to search a dictionary containing n items. Is this the best possible?

Lower bound for search

- The fastest implementations of the dictionary ADT require $O(\log n)$ time to search a dictionary containing n items. Is this the best possible?

- **Theorem**: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-n dictionary.

Lower bound for search

- The fastest implementations of the dictionary ADT require $O(\log n)$ time to search a dictionary containing n items. Is this the best possible?

- **Theorem**: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-n dictionary.

  Proof: Similar to lower bound for sorting.

  - Any algorithm defines a binary decision tree with comparisons at the nodes and actions at the leaves.

    There are at least $n + 1$ different actions (return an item, or "not found").

    So there are $\Omega(n)$ leaves, and therefore the height is $\Omega(\log n)$.

Lower bound for search

- The fastest implementations of the dictionary ADT require $O(\log n)$ time to search a dictionary containing n items. Is this the best possible?

- **Theorem**: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-n dictionary.

  Proof: Similar to lower bound for sorting.

  - Any algorithm defines a binary decision tree with comparisons at the nodes and actions at the leaves.

    There are at least $n+1$ different actions (return an item, or "not found").

    So there are $\Omega(n)$ leaves, and therefore the height is $\Omega(\log n)$.

- We can do better if keys are integers!

Direct Addressing

Requirement: For a given $M \in \mathbb{N}$,
every key k is an integer with $0 \leq k < M$.

- All keys are in $[0, M)$.

Data structure	:	An array of **values** A with size M
$search(k)$	:	Check whether $A[k]$ is empty
$insert(k, v)$	:	$A[k] \leftarrow v$
$delete(k)$	:	$A[k] \leftarrow Null$

Direct Addressing

Requirement: For a given $M \in \mathbb{N}$,
every key k is an integer with $0 \leq k < M$.

- All keys are in $[0, M)$.

Data structure	: An array of **values** A with size M
$search(k)$	: Check whether $A[k]$ is empty
$insert(k, v)$	: $A[k] \leftarrow v$
$delete(k)$	: $A[k] \leftarrow Null$

- E.g., assume student id's are in $[0, 1000)$ and values are pointers to students' records.
  - Maintain an array A of pointers with size 1000.
  - If a student with id k is present in the dictionary, the content of $A[k]$ will be the pointer to that students' record; otherwise it is Null.

Direct Addressing

- Each operation is $O(1)$.

Direct Addressing

- Each operation is $O(1)$.
- Total storage is $O(M)$.

Direct Addressing

- Each operation is $O(1)$.
- Total storage is $O(M)$.
- Direct addressing isn't possible if keys are not integers.
- And the storage is very wasteful if $n \ll M$, e.g., if student numbers are 32-bit integers, you will need an array of size $M = 2^{32}$.

Hashing

- Suppose keys come from some **universe** U.
  Use a **hash function** $h : U \to \{0, 1, \ldots, M-1\}$.
  Generally, h is not injective, so many keys can map to the same integer.

Hashing

- Suppose keys come from some **universe** U.
  Use a **hash function** $h : U \to \{0, 1, \ldots, M - 1\}$.
  Generally, h is not injective, so many keys can map to the same integer.

- **Hash table Dictionary**:
  Array T of size M (the **hash table**).
  An item with key k is stored in $T[h(k)]$.
  search, *insert*, and *delete* should all cost $O(1)$.

Hashing

- Suppose keys come from some **universe** U.
  Use a **hash function** $h : U \to \{0, 1, \ldots, M-1\}$.
  Generally, h is not injective, so many keys can map to the same integer.

- **Hash table Dictionary**:
  Array T of size M (the **hash table**).
  An item with key k is stored in $T[h(k)]$.
  search, *insert*, and *delete* should all cost $O(1)$.

- Challenges:
  - Choosing a good hash function
  - Dealing with **collisions** (when $h(k_1) = h(k_2)$)

Choosing a good hash function

- **Uniform Hashing Assumption**: Each hash function value is equally likely.

  Proving is usually impossible, as it requires knowledge of the input distribution and the hash function distribution.

  We can get good performance by following a few rules.

  A good hash function should:

  - be very efficient to compute
  - be unrelated to any possible patterns in the data
  - depend on all parts of the key

Hash Functions

- The goal of a hash function is to distribute the keys uniformly.
- A hash function takes a key and returns a location in memory that can be accessed in $O(1)$ time.
- A hash function is the composition of two functions:
    - **Hash code map:**
      h_1 : keys $\rightarrow$ integers
    - **Compression map**
      : h_2: integers $\rightarrow [0, M - 1]$
    - The hash code map is applied first and the compression map is then applied on the result: $h(x) = h_2(h_1(x))$

Basic hash functions

- If all keys are integers (or can be mapped to integers), the following two approaches tend to work well:
    - **Division method**: $h(k) = k \bmod M$.
      We should choose M to be a prime not close to a power of 2.
      In the case of non-random data, this ensures the most wide-spread distribution of integers to indices.

Basic hash functions

- If all keys are integers (or can be mapped to integers), the following two approaches tend to work well:

  - **Division method**: $h(k) = k \bmod M$.
    We should choose M to be a prime not close to a power of 2.
    In the case of non-random data, this ensures the most wide-spread distribution of integers to indices.

  - **Multiplication method**: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$,
    for some constant floating-point number A with $0 < A < 1$.
    Knuth suggests $A = \varphi = \dfrac{\sqrt{5}-1}{2} \approx 0.618$.
  - E.g., $k = 2023$, $M = 31$, and $A = 0.618$, we get
    $\lfloor 31(1984 * 0.618 - \lfloor 1984 * 0.618 \rfloor) \rfloor =$
    $\lfloor 31(1226.112 - \lfloor 1226.112 \rfloor) \rfloor = \lfloor 31 * 0.112 \rfloor = 3$

Non-integer keys

- Suppose we have a table capable of holding 5001 records and keys consisting of strings that are 6 characters long. We can apply numeric operations to the ASCII codes of the characters in the string in order to determine a hash index:

```
static int hashFn(String key) {
    int hashCode = 0;
    for (i = 0 ; i < key.length() ; i++)
        hashCode += (int) key.charAt(i);
    return hashCode % 5001; }
```

Non-integer keys

- A better solution via **Horner's Rule**

```
static int hashFn(String key) {
    int hashCode = 0;
    int m = 2; (any int ≠ 1 or 0)
    for (i = 0 ; i < key.length() ; i++)
        hashCode = m * hashCode + (int) key.charAt(i);
    return hashCode % 5001;
}
```

- E.g., for $m = 10$ and string "hope", we get: $\rightarrow 104(h) \rightarrow 10 * 104 + 111(o) \equiv 1151 \rightarrow 10 * 1151 + 112(p) \equiv 1620 \rightarrow 18 * 1620 + 101(e) \equiv 4526$.

Collisions

- If a hash function h maps two different keys x and y to the same index (i.e., $x \neq y$ and $h(x) = h(y)$), then x and y **collide**.
    - A perfect hash function causes no collisions. That is, the function is one-to-one.
    - Unfortunately, creating a perfect hash function requires knowledge of what keys will be hashed.
    - Even a hash function that distributes items randomly can cause collisions, even when the number of items hashed is small.
    - Consequently, we must design a scheme to handle collisions.

Collision Resolution

- Two basic strategies:
  - Allow multiple items at each table location (buckets)
  - Allow each item to go into multiple locations (open addressing)

Collision Resolution

- Two basic strategies:
  - Allow multiple items at each table location (buckets)
  - Allow each item to go into multiple locations (open addressing)
- We will examine the average cost of *search*, *insert*, *delete*, in terms of n, M, and/or the **load factor** $\alpha = n/M$.
- We probably want to rebuild the whole hash table and change the value of M when the load factor gets too large or too small. This is called **rehashing**, and should cost roughly $O(M + n)$.

Chaining

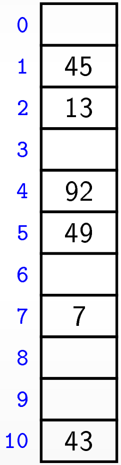
Each table entry is a **bucket** containing 0 or more KVPs.
This could be implemented by any dictionary (even another hash table!).

The simplest approach is to use an unsorted linked list in each bucket.
This is called collision resolution by **chaining**.

- *search*(k): Look for key k in the list at $T[h(k)]$.
- *insert*(k, v): Add (k, v) to the front of the list at $T[h(k)]$.
- *delete*(k): Perform a search, then delete from the linked list.

Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

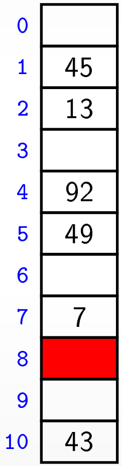


Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

$insert(41)$

$h(41) = 8$

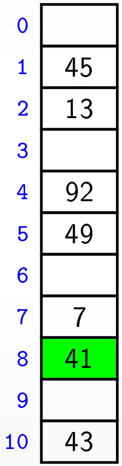


Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

$insert(41)$

$h(41) = 8$

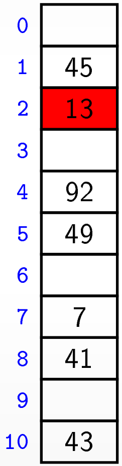


Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

$insert(46)$

$h(46) = 2$

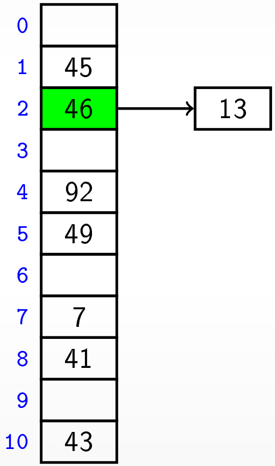


Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

insert(46)

$h(46) = 2$

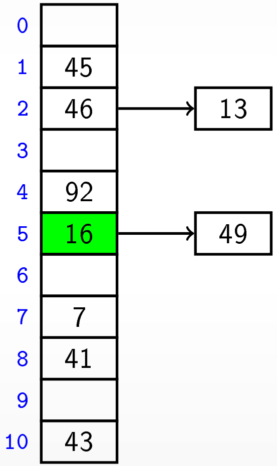


Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

insert(16)

$h(16) = 5$

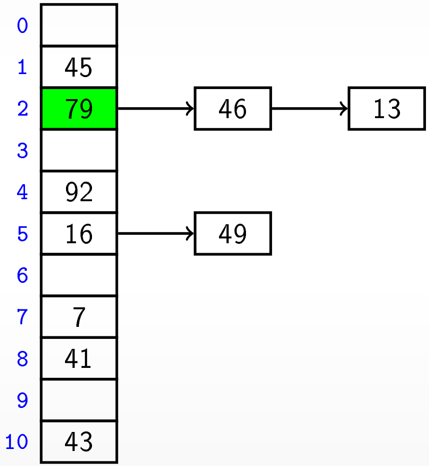


Chaining example

$M = 11, \qquad h(k) = k \bmod 11$

insert(79)

$h(79) = 2$



Complexity of chaining

- Recall the load balance $\alpha = n/M$.
- Assuming uniform hashing, average bucket size is exactly α.
- Analysis of operations:

search	$O(1+\alpha)$ average-case, $O(n)$ worst-case
insert	$O(1)$ worst-case, since we always insert in front.
delete	Same cost as *search*: $O(1+\alpha)$ average, $O(n)$ worst-case

- If we maintain $M \in O(n)$, then average costs are all $O(1)$.
  This is typically accomplished by rehashing whenever $n < c_1 M$ or $n > c_2 M$, for some constants c_1, c_2 with $0 < c_1 < c_2$.

Open addressing

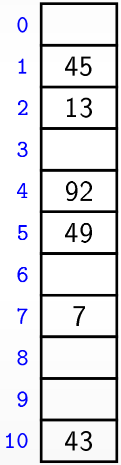
- **Main idea**: Each hash table entry holds only one item, but any key k can go in multiple locations.
- *search* and *insert* follow a **probe sequence** of possible locations for key k: $\langle h(k,0), h(k,1), h(k,2), \ldots \rangle$.
- *delete* is similar to *search* but we must distinguish between **empty** and **deleted** locations.

Open addressing

- **Main idea**: Each hash table entry holds only one item, but any key k can go in multiple locations.
- *search* and *insert* follow a **probe sequence** of possible locations for key k: $\langle h(k,0), h(k,1), h(k,2), \ldots \rangle$.
- *delete* is similar to *search* but we must distinguish between **empty** and **deleted** locations.
- Simplest idea: **linear probing**
  $h(k,i) = (h(k) + i) \bmod M$, for some hash function h.

Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$



Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$

insert(41)

$h(41, 0) = 8$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$

insert(84)

$h(84, 0) = 7$

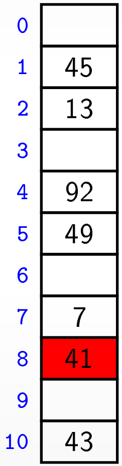
0	
1	45
2	13
3	
4	92
5	49
6	
7	🟥 7
8	41
9	
10	43

Linear probing example

$M = 11, \quad h(k) = k \bmod 11 \quad h(k,i) = (h(k) + i) \bmod M$

insert(84)

$h(84, 1) = 8$



Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$

insert(84)

$h(84, 2) = 9$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$

$insert(20)$

$h(20, 2) = 0$

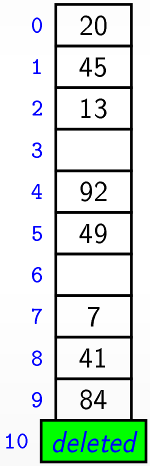
0	🟩 20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	🟥 84
10	🟥 43

Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$

$delete(43)$

$h(43, 0) = 10$

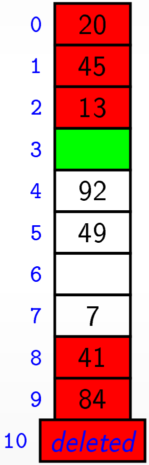


Linear probing example

$M = 11, \qquad h(k) = k \bmod 11 \qquad h(k,i) = (h(k) + i) \bmod M$

search(63)

$h(63, 6) = 3$



Open Addressing: Double Hashing

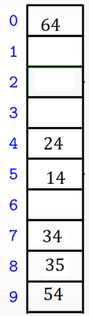
- We have **two** hash functions h_1, h_2 that are **independent**.
- For **double hashing**, define $h(k,i) = h_1(k) + i \cdot h_2(k) \bmod M$.
- *search*, *insert*, *delete* work just like for linear probing, but with this different probe sequence.

Open Addressing: Double Hashing

- Assume we have hash functions: $h_1(x) = x \bmod 10$, $h_2(x) = \lfloor x/10 \rfloor \bmod 10$.
- Recall that $h(k, i) = h_1(k) + i \cdot h_2(k) \bmod M$.
- We want to insert keys: 24, 34, 14, 54, 64, 35, . . .

Open Addressing: Double Hashing

- Assume we have hash functions: $h_1(x) = x \bmod 10$, $h_2(x) = \lfloor x/10 \rfloor \bmod 10$.
- Recall that $h(k,i) = h_1(k) + i \cdot h_2(k) \bmod M$.
- We want to insert keys: 24, 34, 14, 54, 64, 35, ...



Cuckoo hashing

- We have two independent hash functions h_1, h_2.
- We **always** insert a new item into $h_1(k)$.
- This might "kick out" another item, which we then attempt to re-insert into its alternate position.
- Insertion might not be possible if there is a loop. In this case, we have to rehash with a larger M.
- The big advantage is that an element with key k can only be in $T[h_1(k)]$ or $T[h_2(k)]$.

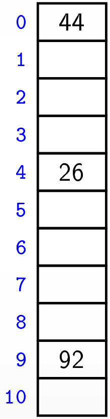
Cuckoo hashing insertion

- Here a **pseudocode** for Cuckoo hashing:

```
cuckoo-insert(T,x)
T: hash table,   x: new item to insert
1.    $y \leftarrow x$,   $i \leftarrow h_1(x.key)$
2.    do at most $n$ times:
3.        $swap(y, T[i])$
4.        if $y$ is "empty" then return "success"
5.        if $i = h_1(y.key)$ then $i \leftarrow h_2(y.key)$
6.        else $i \leftarrow h_1(y.key)$
7.    return "failure"
```

Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$



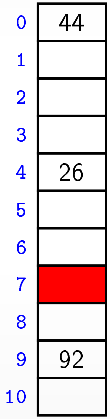
Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(51)$

$y.key = 51$
$i = 7$

$h_1(y.key) = 7$
$h_2(y.key) = 5$



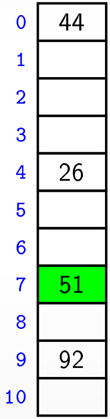
Cuckoo hashing example

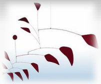
$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(51)$

$y.key =$
$i =$

$h_1(y.key) =$
$h_2(y.key) =$





Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(95)

$y.key = 95$
$i = 7$

$h_1(y.key) = 7$
$h_2(y.key) = 7$

0	44
1	
2	
3	
4	26
5	
6	
7	51
8	
9	92
10	

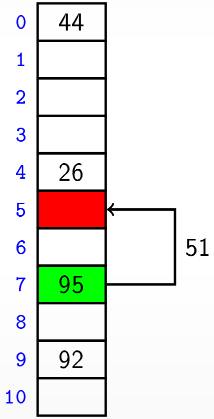
Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(95)$

$y.key = 51$
$i = 5$

$h_1(y.key) = 7$
$h_2(y.key) = 5$



Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(95)

$y.key =$

$i =$

$h_1(y.key) =$

$h_2(y.key) =$

0	44
1	
2	
3	
4	26
5	51
6	
7	95
8	
9	92
10	

Cuckoo hashing example

$M = 11,\qquad h_1(k) = k \bmod 11,\qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 97$
$i = 9$

$h_1(y.key) = 9$
$h_2(y.key) = 10$

0	44
1	
2	
3	
4	26
5	51
6	
7	95
8	
9	92
10	

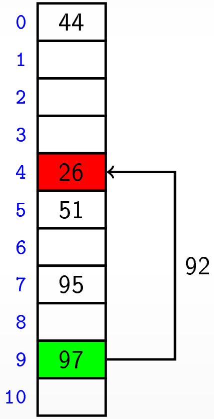
Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(97)$

$y.key = 92$
$i = 4$

$h_1(y.key) = 4$
$h_2(y.key) = 9$



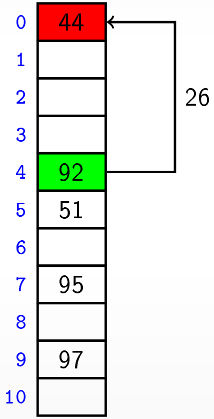
Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 26$
$i = 0$

$h_1(y.key) = 4$
$h_2(y.key) = 0$



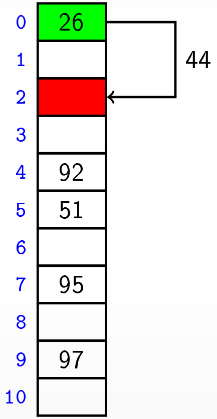
Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 44$
$i = 2$

$h_1(y.key) = 0$
$h_2(y.key) = 2$



Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key =$

$i =$

$h_1(y.key) =$

$h_2(y.key) =$

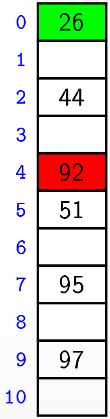
0	26
1	
2	44
3	
4	92
5	51
6	
7	95
8	
9	97
10	

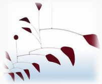
Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

search(26)

$h_1(26) = 4$
$h_2(26) = 0$





Complexity of open addressing strategies

We won't do the analysis, but just state the costs.

For any open addressing scheme, we **must** have $\alpha < 1$ (why?).
Cuckoo hashing requires $\alpha < 1/2$.

The following gives the **big-Theta** cost of each operation for each strategy:

	search	*insert*	*delete*
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$
Double Hashing	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$
Cuckoo Hashing	1	$\frac{\alpha}{(1-2\alpha)^2}$	1

Hashing Summary

- When the size of the hash table *M* is sufficiently large all search, insert, deleted operations can be done in constant time.
- This requires having α (load factor) being small (e.g., $\alpha = 1/2$ or $\alpha = 1/100$.

Hashing Summary

- When the size of the hash table *M* is sufficiently large all search, insert, deleted operations can be done in constant time.
- This requires having α (load factor) being small (e.g., $\alpha = 1/2$ or $\alpha = 1/100$.
- Hashing is often the preferred method for implementing dictionaries.