A **dictionary** is a collection $S$ of **items**, each of which contains a **key** and some **data**, and is called a key-value pair (KVP).

- It is sometimes called an **associative array**, a **map**, or a **symbol table**.
- Keys can be compared and are (typically) unique.
- We often focus on keys; associating data with keys is easy.
Dictionaries

Dictionary ADT

Definition

A dictionary is a collection $S$ of items, each of which contains a key and some data, and is called a key-value pair (KVP).

- It is sometimes called an associative array, a map, or a symbol table.
- Keys can be compared and are (typically) unique.
- We often focus on keys; associating data with keys is easy.

Operations:

- $\text{search}(x)$: return true iff $x \in S$
- $\text{insert}(x, v)$: $S \leftarrow S \cup \{x\}$
- $\text{delete}(x)$: $S \leftarrow S / \{x\}$
- additional: $\text{join}$, $\text{isEmpty}$, $\text{size}$, etc

Examples: student database, symbol table, license plate database
Dictionary is a collection of key-value pairs with the support of search, insert, delete (and possibly some other operations).

- There is a total ordering of elements, i.e., keys are comparable.

- Is dictionary an abstract data type or a data structure?
Dictionaries

Dictionary is a collection of key-value pairs with the support of search, insert, delete (and possibly some other operations).

There is a total ordering of elements, i.e., keys are comparable.

Is dictionary an abstract data type or a data structure?

- It is an abstract data type; we did not discuss implementation.
- Different data structures can be used to implement dictionaries.
Dictionaries

Elementary Implementations

- **Common assumptions:**
  - Dictionary has $n$ KVPs
  - Each KVP uses constant space
    (if not, the “value” could be a pointer)
  - Comparing keys takes constant time

- **Unsorted array or linked list**
  - **search** $O(n)$
  - **insert** $O(1)$
  - **delete** $O(n)$ (need to search)

- **Sorted array**
  - **search** $O(\log n)$
  - **insert** $O(n)$
  - **delete** $O(n)$
There are better data structures (than arrays and lists) for dictionaries:

- Direct addressing (often wasteful of memory)
- **Hash Tables** → Often the fastest and most practical approach
- **Balanced Binary Search Trees** → we will see later.
**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 $M$

- $\text{search}(k)$: Check whether $A[k]$ is empty
- $\text{insert}(k, v)$: $A[k] \leftarrow v$
- $\text{delete}(k)$: $A[k] \leftarrow \text{Null}$

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

**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 $M$

- $\text{search}(k)$: Check whether $A[k]$ is empty
- $\text{insert}(k, v)$: $A[k] \leftarrow v$
- $\text{delete}(k)$: $A[k] \leftarrow \text{Null}$

- 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 isn't possible if keys are not integers. And the storage is very wasteful if $n \ll M$. 
Hashing intro

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 $n \ll M$. 
Hashing intro

Hashing

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

Hashing

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

\textbf{Hash table Dictionary}: Array \( T \) of size \( M \) (the \textbf{hash table}).
An item with key \( k \) is stored in \( T[h(k)] \).
\textit{search}, \textit{insert}, and \textit{delete} should all cost \( O(1) \).
Hashing intro

Hashing

Say keys come from some **universe** $U$.
Use a **hash function** $h : U \rightarrow \{0, 1, \ldots, M - 1\}$.
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 function strategies

Basic hash functions

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

**Division method:** \( h(k) = k \mod M \).
We should choose \( M \) to be a prime not close to a power of 2.

**Multiplication method:** \( h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor \),
for some constant floating-point number \( A \) with \( 0 < A < 1 \).
Knuth suggests \( A = \varphi = \frac{\sqrt{5} - 1}{2} \approx 0.618 \).
Collision Resolution

Even the best hash function may have collisions: when we want to insert \((k, v)\) into the table, but \(T[h(k)]\) is already occupied.
Collision Resolution

Even the best hash function may have collisions: when we want to insert \((k, v)\) into the table, but \(T[h(k)]\) is already occupied.

Two basic strategies:

- Allow multiple items at each table location (buckets)
- Allow each item to go into multiple locations (open addressing)
Collision Resolution

Even the best hash function may have **collisions**: when we want to insert \((k, v)\) into the table, but \(T[h(k)]\) is already occupied.

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** \(\alpha = \frac{n}{M}\).

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 \(O(M + n)\).
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.
Buckets

Chaining example

\[ M = 11, \quad h(k) = k \mod 11 \]

\[ \begin{array}{c|c}
0 & \\
1 & 45 \\
2 & 13 \\
3 & \\
4 & 92 \\
5 & 49 \\
6 & \\
7 & 7 \\
8 & \\
9 & \\
10 & 43 \\
\end{array} \]
**Buckets**

**Chaining example**

\[ M = 11, \quad h(k) = k \mod 11 \]

**Insert(41)**

\[ h(41) = 8 \]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
0 &   &   &   &   &   &   &   &   &   &   &   \\
1 &   &   &   &   &   &   &   &   &   &   & 45 \\
2 &   &   &   &   &   &   &   &   &   & 13 &   \\
3 &   &   &   &   &   &   &   &   &   &   &   \\
4 &   &   &   &   & 92 &   &   &   &   &   &   \\
5 &   &   &   &   & 49 &   &   &   &   &   &   \\
6 &   &   &   &   &   &   &   &   &   &   &   \\
7 &   &   &   &   &   &   & 7 &   &   &   &   \\
8 &   &   &   &   &   &   &   &   & 43 &   &   \\
9 &   &   &   &   &   &   &   &   &   &   &   \\
10 &   &   &   &   &   &   &   &   &   &   &   \\
\end{array}
\]
Chaining example

$M = 11, \quad h(k) = k \mod 11$

**insert(41)**

$h(41) = 8$

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

Chaining example

\[ M = 11, \quad h(k) = k \mod 11 \]

*insert*(46)

\[ h(46) = 2 \]

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

\[ M = 11, \quad h(k) = k \mod 11 \]

\textit{insert}(46)

\[ h(46) = 2 \]
Buckets

Chaining example

\[ M = 11, \quad h(k) = k \mod 11 \]

*insert*(16)

\[ h(16) = 5 \]

\begin{itemize}
  \item \textcolor{red}{16} \textcolor{green}{\rightarrow 13}
  \item \textcolor{green}{16}
  \item \textcolor{green}{49}
  \item \textcolor{green}{43}
  \item \textcolor{green}{41}
  \item \textcolor{green}{7}
  \item \textcolor{red}{46}
  \item \textcolor{red}{45}
  \item \textcolor{red}{0}
\end{itemize}
**Buckets**

**Chaining example**

\[ M = 11, \quad h(k) = k \mod 11 \]

*insert*(79)

\[ h(79) = 2 \]

```
0  45
1  
2  79  46  13
3  
4  92
5  16
6  
7  7
8  41
9  
10 43
```
Recall the load balance $\alpha = n/M$.

Assuming uniform hashing, average bucket size is exactly $\alpha$.

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.
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* 

$$h(k, i) = (h(k) + i) \mod M,$$

for some hash function $h$. 
Open addressing

Linear probing example

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]
Open addressing

Linear probing example

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

\textit{insert}(41)

\[ h(41, 0) = 8 \]
Open addressing

**Linear probing example**

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

**insert(84)**

\[ h(84, 0) = 7 \]

```
insert(84)
```

```
0  
1  45
2  13
3  
4  92
5  49
6  
7  7  [Red]
8  41
9  
10 43
```
Open addressing

Linear probing example

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

*insert*(84)

\[ h(84, 1) = 8 \]
Open addressing

**Linear probing example**

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

*insert*(84)

\[ h(84, 2) = 9 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>84</td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
Open addressing

**Linear probing example**

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

**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
```
Open addressing

Linear probing example

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

delete(43)

\[ h(43, 0) = 10 \]

\[
\begin{array}{c}
0 & 20 \\
1 & 45 \\
2 & 13 \\
3 & \\
4 & 92 \\
5 & 49 \\
6 & \\
7 & 7 \\
8 & 41 \\
9 & 84 \\
10 & \text{deleted} \\
\end{array}
\]
Open addressing

Linear probing example

\[ M = 11, \quad h(k) = k \mod 11 \quad h(k, i) = (h(k) + i) \mod M \]

**search(63)**

\[ h(63, 6) = 3 \]

```
0  20
1  45
2  13
3  deleted
4  92
5  49
6  
7  7
8  41
9  84
10 deleted
```
Open Addressing: Double Hashing

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

Open Addressing: Double Hashing

Assume we have hash functions: $h_1(x) = x \mod 10$, $h_2(x) = \lfloor x/10 \rfloor \mod 10$.

Recall that $h(k, i) = h_1(k) + i \cdot h_2(k) \mod M$.

We want to insert keys: 24, 34, 14, 54, 64, 35, ...
Open Addressing: Double Hashing

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

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 $T[h_2(k)]$. 
Here a pseudocode for Cuckoo hashing:

\[
\text{cuckoo-insert}(T,x)
\]
\[
T: \text{hash table, } x: \text{new item to insert}
\]
1. \( y \leftarrow x, \quad i \leftarrow h_1(x.\text{key}) \)
2. \textbf{do} at most \( n \) times:
3. \hspace{1cm} \text{swap}(y, T[i])
4. \hspace{1cm} \textbf{if} \ y \text{ is "empty" then return "success"}
5. \hspace{1cm} \textbf{if} \ i = h_1(y.\text{key}) \textbf{ then } i \leftarrow h_2(y.\text{key})
6. \hspace{1cm} \textbf{else} \ i \leftarrow h_1(y.\text{key})
7. \hspace{1cm} \textbf{return} \ "failure"
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - [\varphi k]) \rfloor \]
Open addressing

**Cuckoo hashing example**

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

**insert(51)**

\( y.key = 51 \)

\( i = 7 \)

\( h_1(y.key) = 7 \)

\( h_2(y.key) = 5 \)
Open addressing

**Cuckoo hashing example**

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

\[
\begin{align*}
\text{insert}(51) \\
y.key &= \quad i = \\
\quad h_1(y.key) = \\
\quad h_2(y.key) = \\
\end{align*}
\]

```
<table>
<thead>
<tr>
<th>0</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>51</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>92</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

**insert(95)**

\( y.key = 95 \)

\( i = 7 \)

\( h_1(y.key) = 7 \)

\( h_2(y.key) = 7 \)
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lceil 11(\varphi k - \lfloor \varphi k \rfloor) \rceil \]

\textit{insert}(95)

\texttt{y.key} = 51

\( i = 5 \)

\( h_1(\texttt{y.key}) = 7 \)

\( h_2(\texttt{y.key}) = 5 \)
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

insert(95)

\[
\begin{align*}
y.key & = \\
i & = \\
h_1(y.key) & = \\
h_2(y.key) & =
\end{align*}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51</td>
<td></td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

**insert(97)**

\[ y.key = 97 \]
\[ i = 9 \]
\[ h_1(y.key) = 9 \]
\[ h_2(y.key) = 10 \]
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

\textit{insert}(97)

\texttt{y.key} = 92

\[ i = 4 \]

\[ h_1(\texttt{y.key}) = 4 \]

\[ h_2(\texttt{y.key}) = 9 \]

\[ \text{92} \]
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \left\lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \right\rfloor \]

**insert(97)**

\[ y\text{.key} = 26 \]

\[ i = 0 \]

\[ h_1(y\text{.key}) = 4 \]

\[ h_2(y\text{.key}) = 0 \]
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

**insert(97)**

\( y.\text{key} = 44 \)

\( i = 2 \)

\( h_1(y.\text{key}) = 0 \)

\( h_2(y.\text{key}) = 2 \)
Open addressing

## Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

**insert(97)**

\[ y.key = \]

\[ i = \]

\[ h_1(y.key) = \]

\[ h_2(y.key) = \]

<table>
<thead>
<tr>
<th>( )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td>92</td>
<td>51</td>
<td>95</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Open addressing

Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \left\lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \right\rfloor \]

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:

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>$\frac{1}{(1-\alpha)^2}$</td>
<td>$\frac{1}{(1-\alpha)^2}$</td>
<td>$\frac{1}{1-\alpha}$</td>
</tr>
<tr>
<td>Double Hashing</td>
<td>$\frac{1}{1-\alpha}$</td>
<td>$\frac{1}{1-\alpha}$</td>
<td>$\frac{1}{\alpha \log \left( \frac{1}{1-\alpha} \right)}$</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td>$1$</td>
<td>$\frac{\alpha}{(1-2\alpha)^2}$</td>
<td>$1$</td>
</tr>
</tbody>
</table>
Open addressing

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 $\alpha$ (load factor) being small (e.g., $\alpha = 1/2$ or $\alpha = 1/100$).
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 $\alpha$ (load factor) being small (e.g., $\alpha = 1/2$ or $\alpha = 1/100$).

Hashing is often the preferred method for implementing dictionaries.