COMP 2140 - Data Structures

Shahin Kamali

Topic 7 - Hash Tables
University of Manitoba

Partially based on notes by S. Durocher.
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.
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:**

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

Examples: student database, symbol table, license plate database
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?
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.
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**: $O(n)$
Data Structures for Dictionaries

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 \( M \)

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

COMP 2140 - Data Structures
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

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 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
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.

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 \(O(M + n)\).
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

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

<table>
<thead>
<tr>
<th>0</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td></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></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
Chaining example

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

\textit{insert}(41)

\[ h(41) = 8 \]
Buckets

Chaining example

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

**insert(41)**

$h(41) = 8$

<table>
<thead>
<tr>
<th>Bucket</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></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
Buckets

Chaining example

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

**insert(46)**

\[ h(46) = 2 \]

<table>
<thead>
<tr>
<th>Bucket</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></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
Buckets

Chaining example

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

*insert*(46)

\[ h(46) = 2 \]

\[
\begin{array}{c}
0 \\
1 \\
2 & 46 \\
3 \\
4 & 92 \\
5 & 49 \\
6 \\
7 & 7 \\
8 & 41 \\
9 \\
10 & 43 \\
\end{array}
\]

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

Chaining example

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

*insert*(16)

\[ h(16) = 5 \]

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c} 
  \hline
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
  \hline
  0 & & & & & \text{45} & & & & & & \\
  1 & & & & & \text{46} & & & & & & \\
  2 & & & & & \text{92} & & & & & & \\
  3 & & & & & & & & & & & \\
  4 & & & & & & & & & & & \\
  5 & & & & & \text{16} & & & & & & \\
  6 & & & & & & & & & & & \\
  7 & & & & & \text{7} & & & & & & \\
  8 & & & & & \text{41} & & & & & & \\
  9 & & & & & & & & & & & \\
 10 & & & & & \text{43} & & & & & & \\
  \hline
\end{array} \]

\[ 13 \]

\[ 49 \]
Chaining example

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

\[ \text{insert}(79) \]

1. \[ h(79) = 2 \]
2. \[ 45 \]
3. \[ 79 \]
4. \[ 92 \]
5. \[ 16 \]
6. \[ 41 \]
7. \[ 7 \]
8. \[ 43 \]
9. 
10. 

\[ 46 \rightarrow 13 \]
\[ 49 \]
Complexity of chaining

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 $0 < c_1 < c_2$. 
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* becomes problematic; 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* becomes problematic; 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

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

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>45</td>
<td>13</td>
<td></td>
<td>92</td>
<td>49</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|   | 8 | 9 | 10|
|---|---|---|
|   |   | 43|
Linear probing example

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

**insert(41)**

$$h(41, 0) = 8$$
Open addressing

Linear probing example

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

\textit{insert}(84)

\[ h(84, 0) = 7 \]
**Open addressing**

**Linear probing example**

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

**Insert** 84

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

<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></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
Open addressing

Linear probing example

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

*insert*(84)

\( h(84, 2) = 9 \)
Open addressing

Linear probing example

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

**Insert(20)**

\[ h(20, 2) = 0 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</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>
Linear probing example

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

\textit{delete}(43)

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

Linear probing example

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

search(63)

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

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|}
\hline
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
h(i) & 20 & 45 & 13 & - & 92 & 49 & - & 7 & 41 & 84 & deleted \\
\hline
\end{array}
\]
Double Hashing

Say we have two hash functions $h_1, h_2$ that are independent.

So, under uniform hashing, we assume the probability that a key $k$ has $h_1(k) = a$ and $h_2(k) = b$, for any particular $a$ and $b$, is

$$\frac{1}{M^2}.$$

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

Cuckoo hashing

This is a relatively new idea from Pagh and Rodler in 2001.

Again, we use two independent hash functions $h_1, h_2$. The idea is to **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 $T[h_2(k)]$. 
Open addressing

Cuckoo hashing insertion

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

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

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></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

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

insert(51)

\( y.key = 51 \)

\( i = 7 \)

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

\[ h_2(y.key) = 5 \]
Cuckoo hashing example

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

\[ \text{insert}(51) \]

\( y\.key = \)
\[ i = \]
\( h_1(y\.key) = \)
\( h_2(y\.key) = \]

\[
\begin{array}{c|c}
0 & 44 \\
1 & \\
2 & \\
3 & \\
4 & 26 \\
5 & \\
6 & \\
7 & 51 \\
8 & \\
9 & 92 \\
10 & \\
\end{array}
\]
Open addressing

Cuckoo hashing example

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

**insert(95)**

\( y.key = 95 \)
\( i = 7 \)

\( h_1(y.key) = 7 \)
\( h_2(y.key) = 7 \)
**Cuckoo hashing example**

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

**insert(95)**

$y.key = 51$

$i = 5$

$h_1(y.key) = 7$

$h_2(y.key) = 5$

```
<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><strong>51</strong></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><strong>95</strong></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

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

\textit{insert}(95)

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

\begin{align*}
\text{y.key} &= 97 \\
i &= 9
\end{align*}

\begin{align*}
h_1(\text{y.key}) &= 9 \\
h_2(\text{y.key}) &= 10
\end{align*}
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)

\[ y.key = 92 \quad i = 4 \]

\[ h_1(y.key) = 4 \quad h_2(y.key) = 9 \]
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)**

\[ y.key = 26 \]
\[ i = 0 \]

\[ h_1(\text{y.key}) = 4 \]
\[ h_2(\text{y.key}) = 0 \]
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)

\[ y.key = 44 \]

\[ i = 2 \]

\[ h_1(y.key) = 0 \]

\[ h_2(y.key) = 2 \]
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)**

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2 (44)</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>y.key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[h_1(y.key) =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[h_2(y.key) =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</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)

\[ h_1(26) = 4 \]
\[ h_2(26) = 0 \]
Open addressing

**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>Strategy</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>$\alpha$</td>
<td>$1$</td>
</tr>
</tbody>
</table>
If we have a very large dictionary that must be stored externally, how can we hash and minimize page faults?

Most hash strategies covered have scattered data access patterns.

**Linear Probing**: All hash table accesses will usually be in the same page. But \( \alpha \) must be kept small to avoid “clustering”, so there is a lot of wasted space. Also, there is a need for frequent rehashing of the entire table.
Extendible Hashing

Say external memory is stored in blocks (or “pages”) of size $S$. 
**The goal:** Use very few blocks, access only 1 page per operation.

**Basic idea:** Similar to a B-tree with height 1 and max size $S$ at the leaves

The **directory** (similar to root node) is stored in internal memory. Contains a hashtable of size $2^d$, where $d$ is called the order.

Each directory entry points to a **block** (similar to leaves) stored in external memory.
Each block contains at most $S$ items, sorted by hash value.

**Parameters:** an integer $L > 0$ and a hash function
$h : U \rightarrow \{0, 1, 2, \ldots, 2^L - 1\}$
(thought of as binary sequences of length $L$.)
Extendible Hashing Directory

Properties of the directory:

- Directory has order \( d \leq L \),
- Directory contains a hashtable with indices 0, 1, \ldots, \(2^d - 1\).
- To look up a key \( k \) in the directory, use the high-order \( d \) bits of \( h(k) \), that is
  \[
  \left\lfloor \frac{h(k)}{2^{L-d}} \right\rfloor.
  \]
- Each directory entry points to a single block.
  (Many entries can point to the same block.)
Extendible Hashing

Extendible Hashing Blocks

Properties of a block $B$:

- $B$ has a local depth $k_B \leq d$ and size $n_B \leq S$.
- $B$ stores $n_B$ KVPs, sorted by the hash values of the keys.
- Hash values in $B$ agree on the high-order $k_B$ bits. Call this the block index $i_B$, where $0 \leq i_B < 2^{k_B}$.
- Every key key in $B$ satisfies
  \[ i_B \cdot 2^{L-k_B} \leq h(key) < (i_B + 1) \cdot 2^{L-k_B}. \]
- Exactly $2^{d-k_B}$ directory entries point to block $B$. 
Searching is done in the directory, then in a block:

- Given a key \( k \), compute \( h(k) \).
- Lookup a block \( B \) in the directory with index first \( d \) bits of \( h(k) \): \( \lfloor h(k)/2^{L-d} \rfloor \).
- Perform a binary search in \( B \) for all items with hash value \( h(k) \).
Searching in extendible hashing

Searching is done in the directory, then in a block:

- Given a key $k$, compute $h(k)$.
- Lookup a block $B$ in the directory with index first $d$ bits of $h(k)$: $\left\lfloor h(k)/2^{L-d} \right\rfloor$.
- Perform a binary search in $B$ for all items with hash value $h(k)$.

**Cost:**

- CPU time: $O(\log S)$
- Page faults: 1 (directory resides in internal memory)
Insertion in Extendible Hashing

\textit{insert}(k, v) \textit{is done as follows:}

- Search for \( h(k) \) to find the proper block \( B \) for insertion
- If the \( B \) has space \( (n_B < S) \), then put \( (k, v) \) there.
Insertion in Extendible Hashing

\( \text{insert}(k, \nu) \) is done as follows:

- Search for \( h(k) \) to find the proper block \( B \) for insertion.
- If the \( B \) has space (\( n_B < S \)), then put \((k, \nu)\) there.
- Elseif the block is full and \( k_B < d \), perform a block split:
  - Split \( B \) into \( B_0 \) and \( B_1 \).
  - Separate items according to the \((k_B + 1)\)-th bit:
    - if \( h(k) \mod 2^{L-k_B} < 2^{L-k_B-1} \), then \( k \) goes in \( B_0 \), else \( k \) goes in \( B_1 \).
  - Set local depth in \( B_0 \) and \( B_1 \) to \( k_B + 1 \).
  - Update references in the directory.
Insertion in Extendible Hashing

\(\text{insert}(k, \nu)\) is done as follows:

- Search for \(h(k)\) to find the proper block \(B\) for insertion.
- If the \(B\) has space \((n_B < S)\), then put \((k, \nu)\) there.
- Elself the block is full and \(k_B < d\), perform a block split:
  - Split \(B\) into \(B_0\) and \(B_1\).
  - Separate items according to the \((k_B + 1)\)-th bit:
    - if \(h(k) \mod 2^{L-k_B} < 2^{L-k_B-1}\), then \(k\) goes in \(B_0\), else \(k\) goes in \(B_1\).
  - Set local depth in \(B_0\) and \(B_1\) to \(k_B + 1\).
  - Update references in the directory.
- Elself the block is full and \(k_B = d\), perform a directory grow:
  - Double the size of the directory \((d \leftarrow d + 1)\).
  - Update references appropriately.
  - Then split block \(B\) (which is now possible).
Extendible hashing conclusion

\textit{delete}(k) is performed in a reverse manner to \textit{insert}:

- Search for block \( B \) and remove \( k \) from it
- If \( n_B \) is too small, then we perform a \textbf{block merge}
- If every block \( B \) has local depth \( k_B \leq d - 1 \), perform a \textbf{directory shrink}
Extendible Hashing

Extendible hashing conclusion

\textit{delete}(k) is performed in a reverse manner to \textit{insert}:

- Search for block \( B \) and remove \( k \) from it
- If \( n_B \) is too small, then we perform a \textit{block merge}
- If every block \( B \) has local depth \( k_B \leq d - 1 \), perform a \textit{directory shrink}

Cost of \textit{insert} and \textit{delete}:

\begin{itemize}
  \item \textbf{CPU time:} \( O(S) \) without a directory grow/shrink (which rarely happen)
  \item \textbf{Directory grow/shrink costs} \( O(2^d) \).
  \item \textbf{Page faults:} 1 or 2, depending on whether there is a block split/merge.
\end{itemize}
Summary of extendible hashing

- Directory is much smaller than total number of stored keys and should fit in main memory.
- Only 1 or 2 external blocks are accessed by any operation.
- To make more space, we only add a block. Rarely do we have to change the size of the directory. Never do we have to move all items in the dictionary (in contrast to normal hashing).
- Space usage is not too inefficient: can be shown that under uniform hashing, each block is expected to be 69% full.
- Main disadvantage:
Conclusion

Summary of extendible hashing

- Directory is much smaller than total number of stored keys and should fit in main memory.
- Only 1 or 2 external blocks are accessed by any operation.
- To make more space, we only add a block. Rarely do we have to change the size of the directory. Never do we have to move all items in the dictionary (in contrast to normal hashing).
- Space usage is not too inefficient: can be shown that under uniform hashing, each block is expected to be 69% full.
- Main disadvantage: extra CPU cost of $O(\log S)$ or $O(S)$
## Conclusion

**Hashing vs. Balanced Search Trees**

### Advantages of Balanced Search Trees
- $O(\log n)$ worst-case operation cost
- Does not require any assumptions, special functions, or known properties of input distribution
- No wasted space
- Never need to rebuild the entire structure

### Advantages of Hash Tables
- $O(1)$ cost, but only on average
- Flexible load factor parameters
- Cuckoo hashing achieves $O(1)$ worst-case for search & delete

### External memory:
Both approaches can be adopted to minimize page faults.
Multi-dimensional Data

What if the keys are multi-dimensional, such as strings?

Say $U = \{0, 1, \ldots, s - 1\}^d$, length-$d$ strings from size-$s$ alphabet.

Standard approach is to flatten to integers using a function $f : U \rightarrow \mathbb{N}$.

We combine this with a standard hash function $h : \mathbb{N} \rightarrow \{0, 1, 2, \ldots, M - 1\}$.

With $h(f(k))$ as the hash values, we then use any standard hash table.

**Note:** computing each $h(f(k))$ takes $\Omega(d)$ time.