Module 5: Hashing

CS 240 - Data Structures and Data Management

There Biedl   Daniela Maftuleac

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2015

Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ 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 $\Theta(\log n)$ 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 $\Theta(\log n)$ 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 $\Omega(\log n)$.

$\square$

Direct Addressing

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

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) : $A[k] \leftarrow \mathit{empty}$

Direct Addressing

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

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) : $A[k] \leftarrow$ *empty*

Each operation is $\Theta(1)$.
Total storage is $\Theta(M)$.

What sorting algorithm does this remind you of?

Direct Addressing

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

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) : $A[k] \leftarrow \mathit{empty}$

Each operation is $\Theta(1)$.
Total storage is $\Theta(M)$.

What sorting algorithm does this remind you of? *Counting Sort*

Hashing

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

Say 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

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

Say 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 (later)
- Dealing with *collisions* (when $h(k_1) = h(k_2)$)

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 $\Theta(M + n)$.

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

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

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

Chaining example

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

insert(41)

$h(41) = 8$

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

Chaining example

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

insert(41)

$h(41) = 8$

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

Chaining example

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

insert(46)

$h(46) = 2$

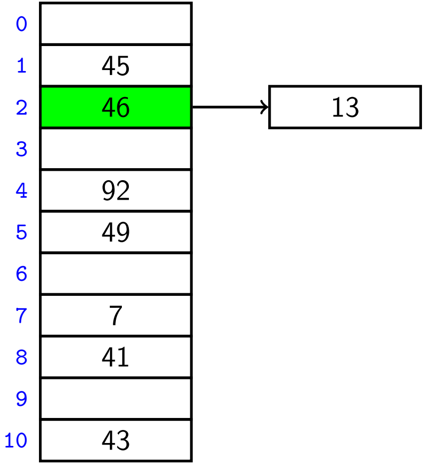
0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Chaining example

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

insert(46)

$h(46) = 2$

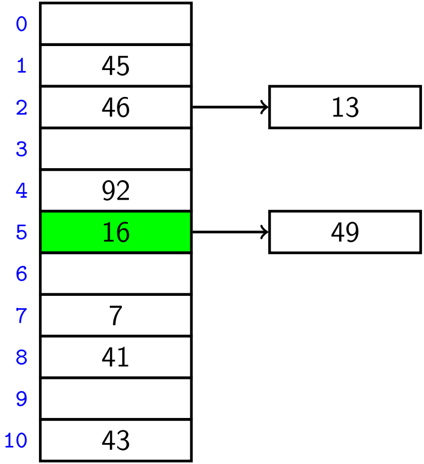


Chaining example

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

insert(16)

$h(16) = 5$

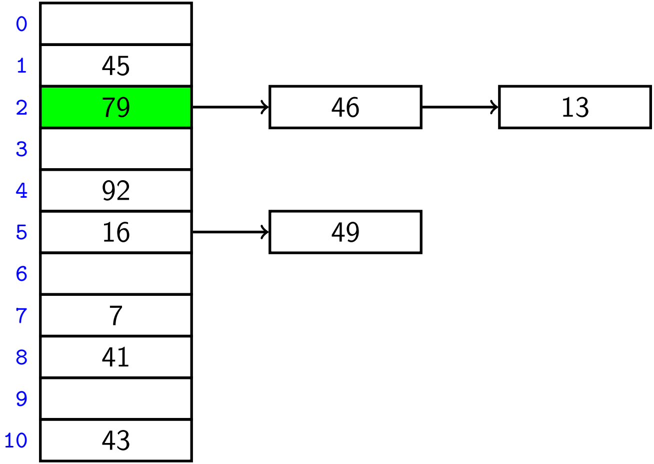


Chaining example

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

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 $\Theta(1+\alpha)$ average-case, $\Theta(n)$ worst-case

insert $O(1)$ worst-case, since we always insert in front.

delete Same cost as *search*: $\Theta(1+\alpha)$ average, $\Theta(n)$ worst-case

If we maintain $M \in \Theta(n)$, 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 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*
$h(k,i) = (h(k) + i) \bmod M$, for some hash function h.

Linear probing example

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

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

Linear probing example

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

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

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

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

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

insert(84)

$h(84, 1) = 8$

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

Linear probing example

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

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

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

insert(20)

$h(20, 0) = 9$

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

Linear probing example

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

insert(20)

$h(20, 1) = 10$

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

Linear probing example

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

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

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

delete(43)

$h(43, 0) = 10$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 0) = 8$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 1) = 9$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 2) = 10$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 3) = 0$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 4) = 1$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 5) = 2$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

Linear probing example

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

search(63)

$h(63, 6) = 3$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

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 $h(k, i) = h_1(k) + i \cdot h_2(k) \bmod M$.

search, *insert*, *delete* work just like for linear probing, but with this different probe sequence.

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor k/2 \rfloor \bmod 11$

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

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor k/2 \rfloor \bmod 11$

insert(41)

$h_1(41) = 8$

$h(41, 0) = 8$

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

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor k/2 \rfloor \bmod 11$

insert(84)

$h_1(84) = 7$

$h(84, 0) = 7$

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

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor k/2 \rfloor \bmod 11$

insert(84)

$h_1(84) = 7$

$h(84, 0) = 7$

$h_2(84) = 9$

$h(84, 1) = 5$

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

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor k/2 \rfloor \bmod 11$

insert(84)

$h_1(84) = 7$

$h(84, 0) = 7$

$h_2(84) = 9$

$h(84, 1) = 5$

$h(84, 2) = 3$

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

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 $T[h_2(k)]$.

Cuckoo hashing insertion

cuckoo-insert(T,x)

T: hash table, x: new item to insert

$y \leftarrow x$, $i \leftarrow h_1(x.key)$

do at most n times:

  swap$(y, T[i])$

  if y is "empty" **then return** "success"

  if $i = h_1(y.key)$ **then** $i \leftarrow h_2(y.key)$

  else $i \leftarrow h_1(y.key)$

return "failure"

Cuckoo hashing example

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

0	44
1	
2	
3	
4	26
5	
6	
7	
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(51)

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

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

0	44
1	
2	
3	
4	26
5	
6	
7	
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(51)

$y.key =$
$i =$

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

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)

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

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

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

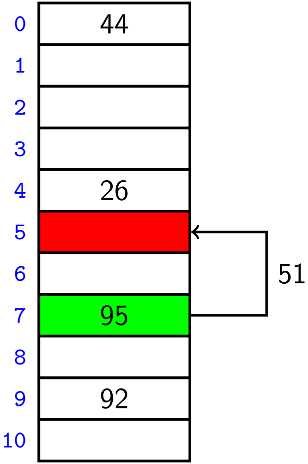
Cuckoo hashing example

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

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	

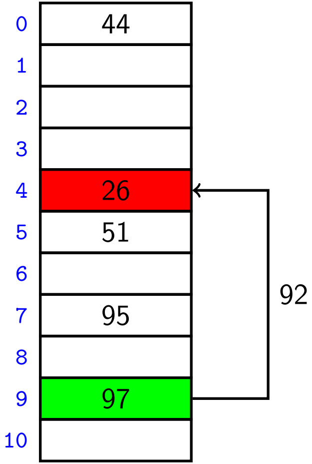
Cuckoo hashing example

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

insert(97)

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

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



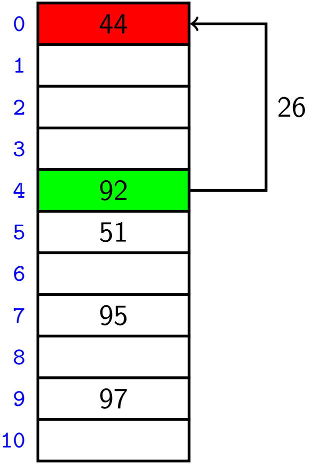
Cuckoo hashing example

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

insert(97)

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

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



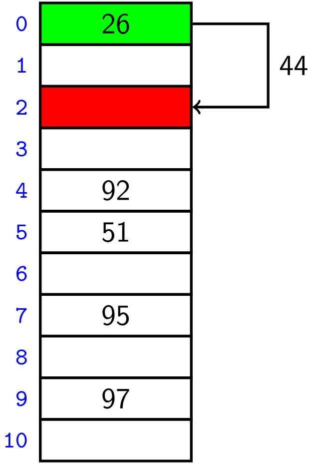
Cuckoo hashing example

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

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)

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

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

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

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

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.

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

Multi-dimensional Data

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

Standard approach is to *flatten* string w to integer $f(w) \in \mathbb{N}$, e.g.

$$\begin{aligned} A \cdot P \cdot P \cdot L \cdot E &\rightarrow 65 \cdot 80 \cdot 80 \cdot 76 \cdot 69 \quad \text{(ASCII)} \\ &\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 68R^0 \\ &\qquad \text{(for some radix } R\text{, e.g. } R = 255) \end{aligned}$$

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 $\Omega(\text{length of } w)$ time.

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 loads.

Hashing in External Memory

If we have a *very large* dictionary that must be stored externally, how can we hash and minimize disk transfers? Say external memory is

stored in blocks (or "pages") of size S.

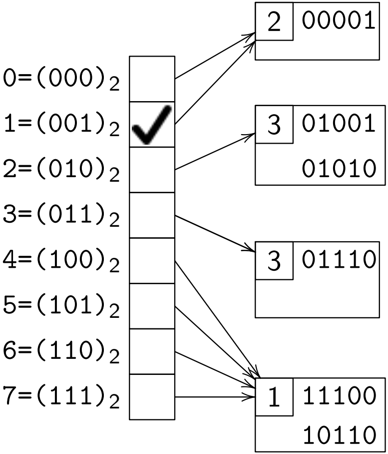
Most hash strategies covered access many pages (data is scattered).

Exception: **Linear Probing**: All hash table accesses will usually be in the

same page.
But α 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*.
New Idea: **Extendible Hashing**: Similar to a B-tree with height 1 and

max size S at the leaves

Extendible Hashing Overview



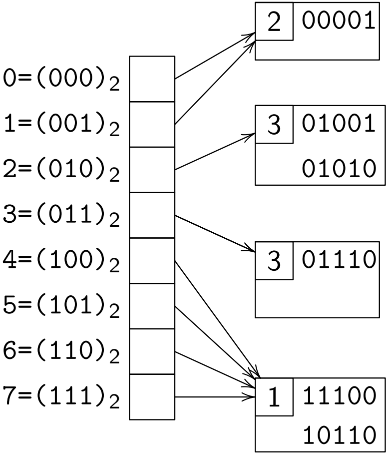
Assumption: Hash-function has values in $\{0, 1, \ldots, 2^L - 1\}$.

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

Each directory entry points to a *block* stored in *external memory*.
Each block contains at most S items.
(Many entries can point to the same block.)

To look up a key k in the directory, use the d leading bits of $h(k)$.

Extendible Hashing Details



Blocks are shared by entries in a specific manner:

- Every block B stores a *local depth* $k_B \leq d$.
- Hash values in B agree on leading k_B bits.
- All directory entries with the same k_B leading bits point to B.
- So 2^{d-k_B} directory entries point to block B.

Searching in extendible hashing

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

- Given a key k, compute $h(k)$.
- Leading d digits of $h(k)$ give index in directory.
- Load block B at this index into main memory.
- Perform a search in B for all items with hash value $h(k)$.
- Search among them for the one with key k.

Searching in extendible hashing

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

- Given a key k, compute $h(k)$.
- Leading d digits of $h(k)$ give index in directory.
- Load block B at this index into main memory.
- Perform a search in B for all items with hash value $h(k)$.
- Search among them for the one with key k.

Cost:

CPU time: $\Theta(\log S)$ if B stores items in sorted order.

Disk transfers: 1 (directory resides in internal memory)

Insertion in Extendible Hashing

insert(k, v) is done as follows:

- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.

Insertion in Extendible Hashing

insert(k, v) is done as follows:

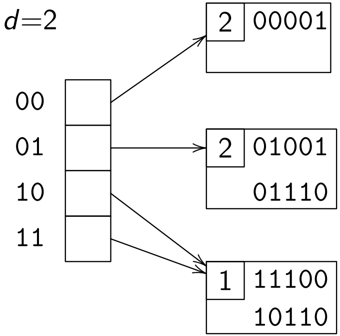
- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.
- ElseIf the block is full and $k_B < d$, perform a *block split*:
  - Split B into two blocks B_0 and B_1.
  - Separate items according to the $(k_B + 1)$-th bit.
  - Set local depth in B_0 and B_1 to $k_B + 1$
  - Update references in the directory

Insertion in Extendible Hashing

insert(k, v) is done as follows:

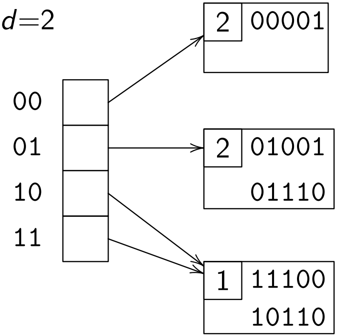
- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.
- ElseIf the block is full and $k_B < d$, perform a *block split*:
  - Split B into two blocks B_0 and B_1.
  - Separate items according to the $(k_B + 1)$-th bit.
  - Set local depth in B_0 and B_1 to $k_B + 1$
  - Update references in the directory
- ElseIf 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).

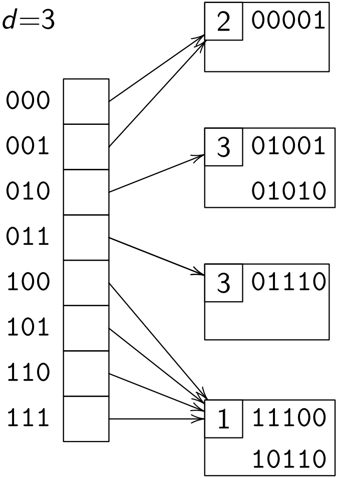
Extendible hashing insert example



- `Insert( 00100 )`
- `Insert( 01010 )`

Extendible hashing insert example





- `Insert( 00100 )`
- `Insert( 01010 )`

Extendible hashing conclusion

delete(k) is performed in a reverse manner to *insert*:

- Search for block B and remove k from it
- If block becomes too empty, then we perform a *block merge*
- If every block B has local depth $k_B \leq d - 1$, perform a *directory shrink*

But most likely just do *lazy deletion*.

Extendible hashing conclusion

delete(k) is performed in a reverse manner to *insert*:

- Search for block B and remove k from it
- If block becomes too empty, then we perform a *block merge*
- If every block B has local depth $k_B \leq d - 1$, perform a *directory shrink*

But most likely just do *lazy deletion*.

Cost of *insert* and *delete*:

CPU time: $\Theta(S)$ without a directory grow/shrink
Directory grow/shrink costs $\Theta(2^d)$ (but very rare).

Disk transfers: 1 or 2, depending on whether there is a block split/merge.

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:

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 $O(S)$