

Data Structures

Course 3,
Gabriel Istrate

March 18, 2025

- Interface

- ▶ `List-Insert(L, x)` adds element x at beginning of a list L
- ▶ `List-Delete(L, x)` removes element x from a list L
- ▶ `List-Search(L, k)` finds an element whose key is k in a list L

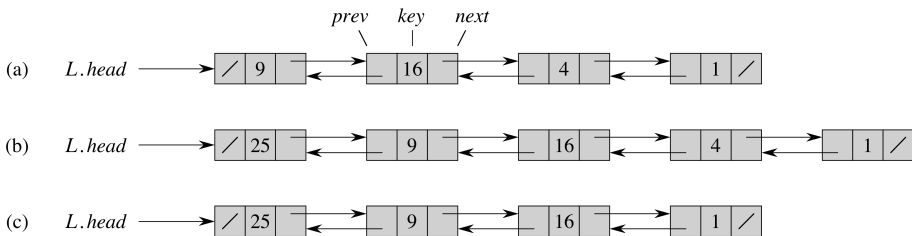
- Interface

- ▶ `List-Insert(L, x)` adds element `x` at beginning of a list `L`
- ▶ `List-Delete(L, x)` removes element `x` from a list `L`
- ▶ `List-Search(L, k)` finds an element whose key is `k` in a list `L`

- Implementation

- ▶ a doubly-linked list
- ▶ each element `x`: two “links” `x.prev` and `x.next` to the previous and next elements, respectively
- ▶ each element `x`: key `x.key`

Linked List: Implementation



- (a). Linked list representing set $S = \{1, 4, 9, 16\}$.
- (b). After $LIST-INSERT(S, 25)$.
- (c). After $LIST-DELETE(S, 4)$.

Linked List: Implementation

List-Init(L)

```
1  L.head = NIL
```

List-Insert(L, x)

```
1  x.next = L.head
2  if L.head  $\neq$  NIL
3      L.head.prev = x
   L.head = x
   x.prev = NIL
```

List-Search(L, k)

```
1  x = L.head.
2  while x  $\neq$  NIL  $\wedge$  x.key  $\neq$  k
3      x = x.next
4  return x
```

Linked List: Implementation (II)

List-Delete(L, x)

```
1  if x.prev  $\neq$  NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next  $\neq$  NIL
5      x.next.prev = x.prev
```

Linked List With a “Sentinel”

- instead of NIL sometimes convenient to have a dummy “sentinel” element $L.nil$
- Simplifies LIST-DELETE .
- Adds more memory \times .

Linked List With a “Sentinel”

List-Init(L)

```
1  L.nil.prev = L.nil  
2  L.nil.next = L.nil
```

List-Insert(L, x)

```
1  x.next = L.nil.next  
2  L.nil.next.prev = x  
3  L.nil.next = x  
4  x.prev = L.nil
```

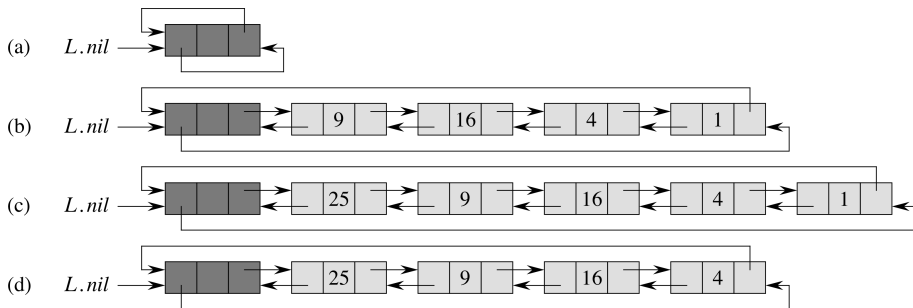
List-Search(L, k)

```
1  x = L.nil.next  
2  while  $x \neq L.nil \wedge x.key \neq k$   
3      x = x.next  
4  return x
```


Linked Lists: Observations on Implementation

- Insert: at the head of the list.
- Possible: insert arbitrary position.

Circular Linked Lists



- Can use nil sentinel as head of the list.
- (a): empty circular list.
- (b): Linked list representing set $S = \{1, 4, 9, 16\}$.
- (c): After `LIST-INSERT(S,25)`.
- (d): After `LIST-DELETE(S,4)`.

Linked Lists: Scorecard

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	$O(1)$ ✓
List-Delete (with pointer)	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	$O(1)$ ✓
List-Delete (with pointer)	$O(1)$ ✓
List-Search	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	$O(1)$ ✓
List-Delete (with pointer)	$O(1)$ ✓
List-Search	$\Theta(n)$ ✗

Linked Lists: to conclude

- Can reimplement Stacks/Queues using Linked Lists.
- Implementation with pointers: **will not pass the class if you don't know it !**

Advanced topic - Skip lists

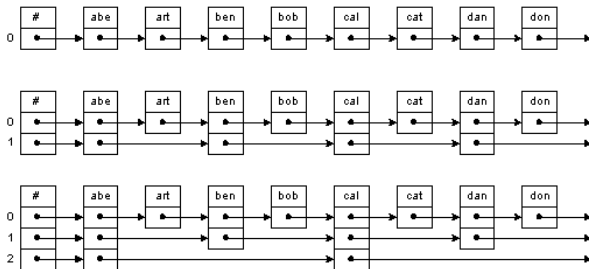
Caution

Topic not in Cormen. See Drozdek for details/C++ implementation.

- Problem with linked list: **search is slow !...** even when elements sorted.
- Solution: **lists of ordered elements** that allow skipping some elements to speed up search.
- **Skip lists**: variant of ordered linked lists that makes such search possible.

More advanced data structure (W. Pugh "Skip lists: a Probabilistic Alternative to Balanced Trees", Communication of the ACM 33(1990), pp. 668-676.) **If anyone curious/interested in data structures/algorithms, can give paper to read; taste how a research article looks like.**

Skip lists



Too theoretical ?

Where does this ever get applied ?

...

Skip lists in real life

According to Wikipedia:

- [MemSQL](#) - skip lists as prime indexing structure for its database technology.
- [Cyrus IMAP server](#) - "skiplist" backend DB implementation
- [Lucene](#) uses skip lists to search delta-encoded posting lists in logarithmic time.
- [QMap](#) (up to Qt 4) template class of Qt that provides a dictionary.
- [Redis](#), ANSI-C open-source persistent key/value store for Posix systems, skip lists in implementation of ordered sets.
- [nessDB](#), a very fast key-value embedded Database Storage Engine.
- [skipdb](#): open-source DB format using ordered key/value pairs.
- [ConcurrentSkipListSet](#) and [ConcurrentSkipListMap](#) in the [Java 1.6 API](#).

Skip lists in real life (II)

According to Wikipedia:

- **Speed Tables**: fast key-value datastore for Tcl that use skiplists for indexes and lockless shared memory.
- **leveldb**, a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values
- **MuQSS** Scheduler for the Linux kernel uses skip lists
- **SkipMap** uses skip lists as base data structure to build a more complex 3D Sparse Grid for Robot Mapping systems.

Skip lists: implementation

What we want

$k = 1, \dots, \lfloor \log_2(n) \rfloor, 1 \leq i \leq \lfloor n/2^{k-1} \rfloor - 1.$

- Item $2^{k-1} \cdot i$ points to item $2^{k-1} \cdot (i + 1).$
 - every second node points to positions two node ahead,
 - every fourth node points to positions four nodes ahead,
 - every eighth node points to positions eighth nodes ahead,
 -, and so on.
-
- Different number of pointers in different nodes in the list !
 - half the nodes only one pointer.
 - a quarter of the nodes two pointers,
 - an eighth of the nodes four pointers,
 -, and so on.
 - $n \log_2(n)/2$ pointers.

Search Algorithm

- ① First **follow pointers on the highest level** until a larger element is found or the list is exhausted.
- ② If a larger element is found, restart search from its predecessor, this time **on a lower level**.
- ③ Continue doing this until element found, or you reach the first level and a larger element or the end of the list.

Inserting and deleting nodes

Major problem

- When inserting/deleting a node, pointers of prev/next nodes have to be restructured.
- Solution: rather than equal spacing, **random spacing** on a level.
- Invariant: **Number of nodes on each level: equal, in expectation to what it would be under equal spacing**

Principle

If you're traveling 10 meters in 10 steps, a step is **on average** one meter.

Inserting and deleting nodes (II)

- Level numbering: start with zero.
- New node inserted: probability $1/2$ on first level, $1/4$ second level, $1/8$ third level, ..., etc.
- Function chooseLevel: chooses randomly the level of the new node.
- Generate random number. If in $[0, 1/2]$ level 1, $[1/2, 3/4]$ level 2, etc.
- To delete node: have to update all links.

Computing the i 'th element faster than in $O(i)$

- If we record “step sizes” in our lists we can even mimic indexing !
- Start on highest level.
- If step too big, restart search from predecessor, this time on a lower level.
- Continue doing this until element found.

Update “step sizes” by insertion/deletion

Easy if you have doubly linked lists.

- On deletion: $\text{pred}[i].\text{size}+ = \text{deleted.size}$ on all levels i .
- On insertion: Simply keep track of predecessors and index of the inserted sequence.

Skip Lists: Scorecard

Method	Average	Worst-Case
SPACE:	$O(n)$	$O(n \log(n))$
✓		
SEARCH:	$O(\log(n))$	$O(n)$
✓		
INSERT:	$O(\log(n))$	$O(n)$
✓		
DELETE:	$O(\log(n))$	$O(n)$
✓		

- quite practical ! ✓
- Probabilistic, worst-case still bad. ×
- Not completely easy to implement. ×.

Compared to what ?

Binary search trees. Will learn about them later.

- Idea

- ▶ use a table T with $|T| \ll |U|$
- ▶ map each key $k \in U$ to a position in T , using a **hash function**

$$h : U \rightarrow \{1, \dots, |T|\}$$

- ▶ $h(k)$ **easy ($O(1)$) to compute given k**

- Idea

- ▶ use a table T with $|T| \ll |U|$
- ▶ map each key $k \in U$ to a position in T , using a [hash function](#)

$$h : U \rightarrow \{1, \dots, |T|\}$$

- ▶ $h(k)$ [easy \(\$O\(1\)\$ \) to compute given \$k\$](#)

```
Hash-Insert( $T, k$ )  
1   $T[h(k)] = \text{true}$ 
```

```
Hash-Delete( $T, k$ )  
1   $T[h(k)] = \text{false}$ 
```

```
Hash-Search( $T, k$ )  
1  return  $T[h(k)]$ 
```

- Idea

- ▶ use a table T with $|T| \ll |U|$
- ▶ map each key $k \in U$ to a position in T , using a [hash function](#)

$$h : U \rightarrow \{1, \dots, |T|\}$$

- ▶ $h(k)$ [easy \(\$O\(1\)\$ \) to compute given \$k\$](#)

```
Hash-Insert( $T, k$ )  
1   $T[h(k)] = \text{true}$ 
```

```
Hash-Delete( $T, k$ )  
1   $T[h(k)] = \text{false}$ 
```

```
Hash-Search( $T, k$ )  
1  return  $T[h(k)]$ 
```

Are these algorithms always correct?

- Idea

- ▶ use a table T with $|T| \ll |U|$
- ▶ map each key $k \in U$ to a position in T , using a **hash function**

$$h : U \rightarrow \{1, \dots, |T|\}$$

- ▶ $h(k)$ **easy ($O(1)$) to compute given k**

```
Hash-Insert( $T, k$ )  
1   $T[h(k)] = \text{true}$ 
```

```
Hash-Delete( $T, k$ )  
1   $T[h(k)] = \text{false}$ 
```

```
Hash-Search( $T, k$ )  
1  return  $T[h(k)]$ 
```

Are these algorithms always correct? **No!**

- Idea

- ▶ use a table T with $|T| \ll |U|$
- ▶ map each key $k \in U$ to a position in T , using a **hash function**

$$h : U \rightarrow \{1, \dots, |T|\}$$

- ▶ $h(k)$ **easy ($O(1)$) to compute given k**

```
Hash-Insert( $T, k$ )  
1   $T[h(k)] = \text{true}$ 
```

```
Hash-Delete( $T, k$ )  
1   $T[h(k)] = \text{false}$ 
```

```
Hash-Search( $T, k$ )  
1  return  $T[h(k)]$ 
```

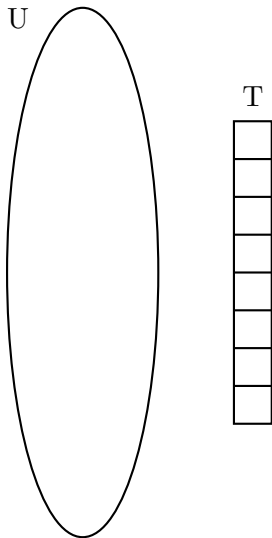
Are these algorithms always correct? **No!** What if two distinct keys $k_1 \neq k_2$ collide? (I.e., $h(k_1) = h(k_2)$)

- Work well "on the average"
- Analogy: throw T balls at random into N bins.
- If $T \ll N$ (in fact $T = o(\sqrt{N})$) then with high-probability no two balls land in the same bin.
- On the average: T/N balls in each bin.
- Want our hash-function to be "random-like": elements of U "thrown out uniformly" by h onto elements of T .

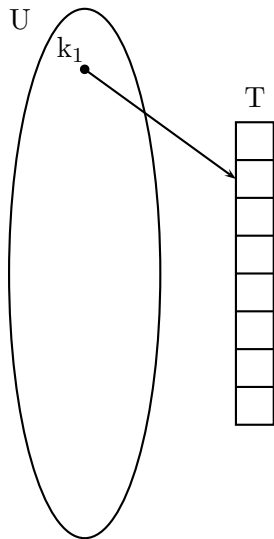
Hashing With Chaining

- Store all objects that map to the same bucket in a linked list.
- "Hope" that hash function is "uniform enough", linked lists are not too large, set operations are efficient.

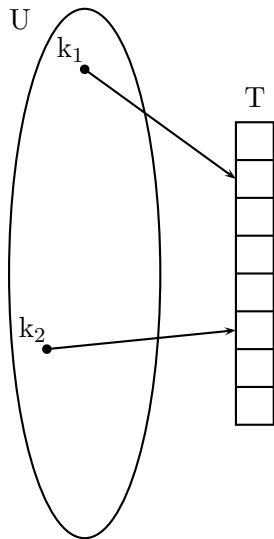
Hash Table: Chaining



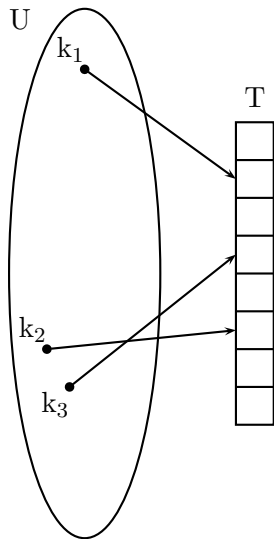
Hash Table: Chaining



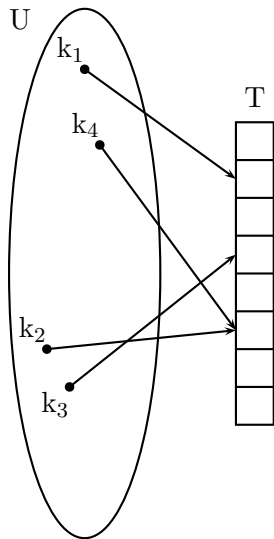
Hash Table: Chaining



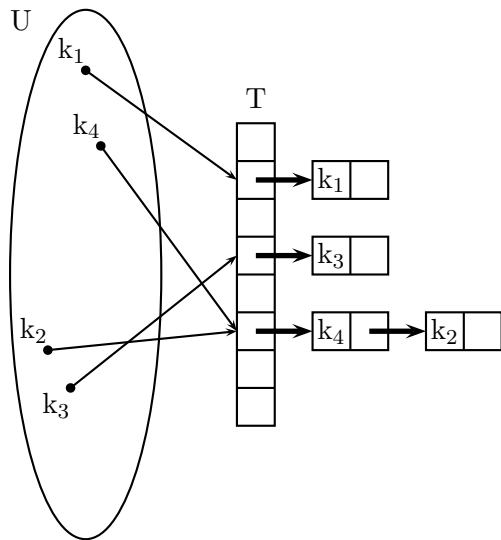
Hash Table: Chaining



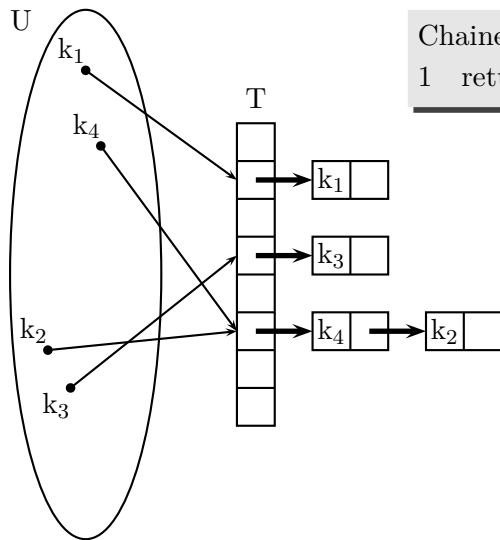
Hash Table: Chaining



Hash Table: Chaining

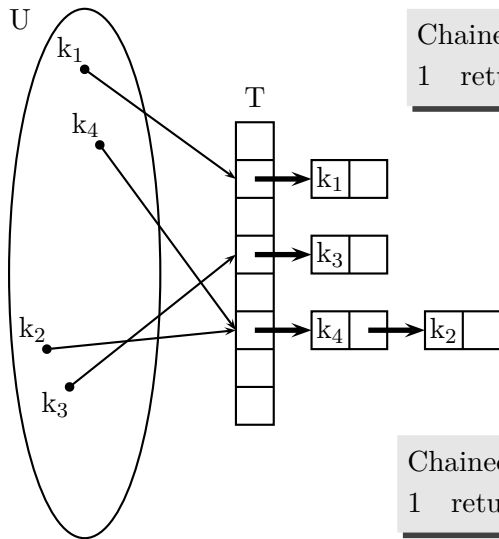


Hash Table: Chaining



```
Chained-Hash-Insert( $T, k$ )  
1  return List-Insert( $T[h(k)], k$ )
```

Hash Table: Chaining



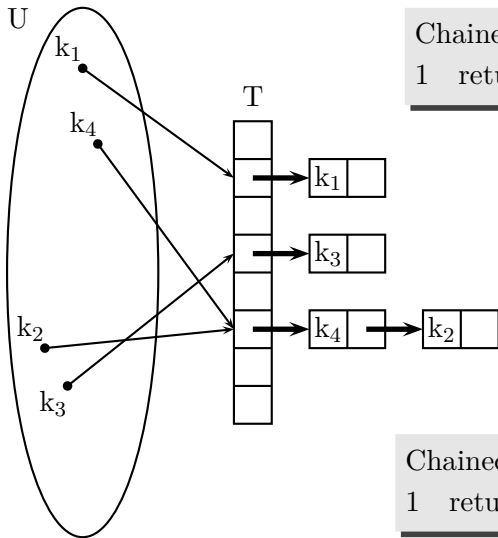
Chained-Hash-Insert(T, k)

1 return List-Insert($T[h(k)], k$)

Chained-Hash-Search(T, k)

1 return List-Search($T[h(k)], k$)

Hash Table: Chaining



Chained-Hash-Insert(T, k)

1 return List-Insert($T[h(k)], k$)

load factor

$$\alpha = \frac{n}{|T|}$$

Chained-Hash-Search(T, k)

1 return List-Search($T[h(k)], k$)

Hashing With Chaining: Analysis

- We assume **uniform hashing** for our hash function
 $h : U \rightarrow \{1 \dots |T|\}$ (where $|T| = T.\text{length}$)

Hashing With Chaining: Analysis

- We assume **uniform hashing** for our hash function

$h : U \rightarrow \{1 \dots |T|\}$ (where $|T| = T.\text{length}$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

(The formalism is actually a bit more complicated.)

Hashing With Chaining: Analysis

- We assume **uniform hashing** for our hash function

$h : U \rightarrow \{1 \dots |T|\}$ (where $|T| = T.length$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

(The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

Hashing With Chaining: Analysis

- We assume **uniform hashing** for our hash function

$h : U \rightarrow \{1 \dots |T|\}$ (where $|T| = T.length$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

(The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time

Hashing With Chaining: Analysis

- We assume **uniform hashing** for our hash function

$h : U \rightarrow \{1 \dots |T|\}$ (where $|T| = T.length$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

(The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time
- Therefore, the complexity of Chained-Hash-Search is

$$\Theta(1 + \alpha)$$

Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists, store all the elements in the table
 - ▶ this implies $\alpha \leq 1$

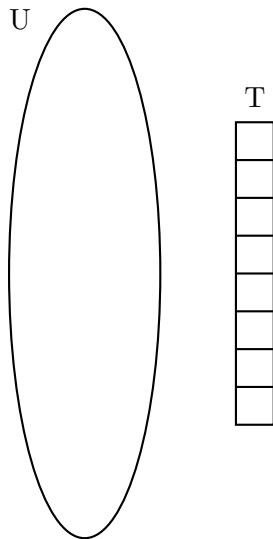
Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists, store all the elements in the table
 - ▶ this implies $\alpha \leq 1$
- When a collision occurs, simply find another free cell in T

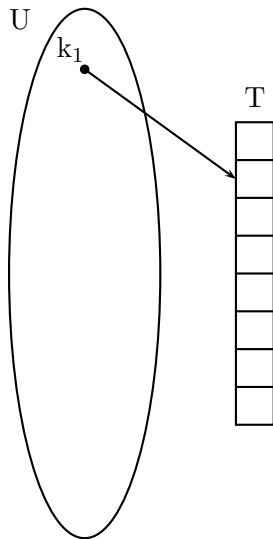
Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists, store all the elements in the table
 - ▶ this implies $\alpha \leq 1$
- When a collision occurs, simply find another free cell in T
- A sequential “probing” method may not be optimal
 - ▶ can you imagine why?

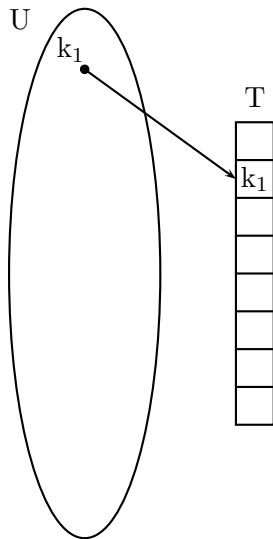
Open-Address Hash Table



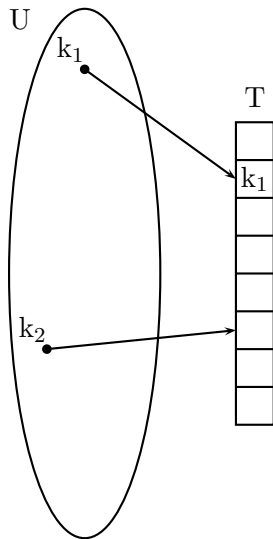
Open-Address Hash Table



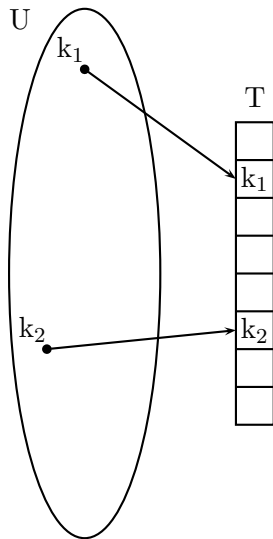
Open-Address Hash Table



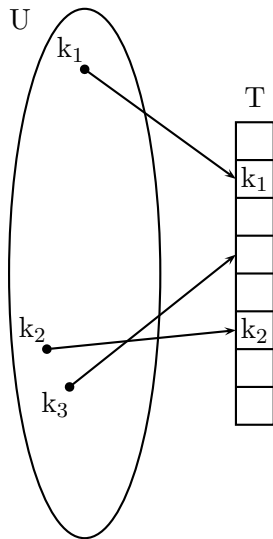
Open-Address Hash Table



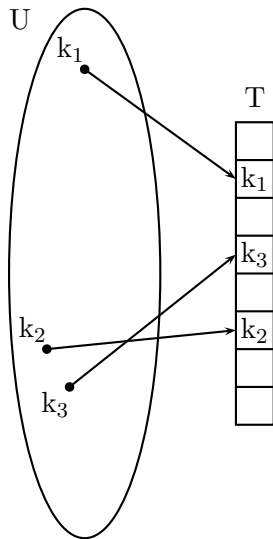
Open-Address Hash Table



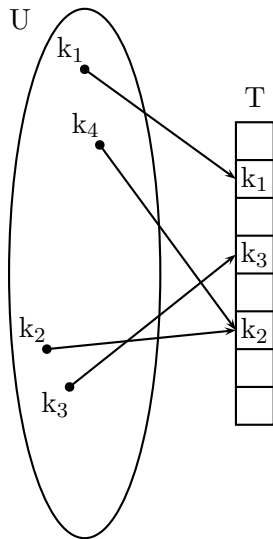
Open-Address Hash Table



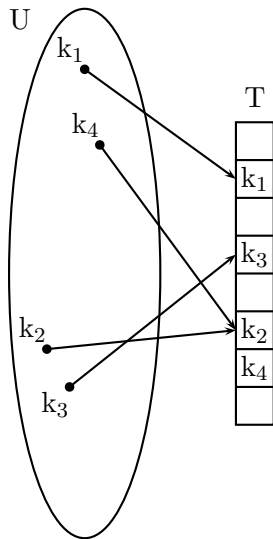
Open-Address Hash Table



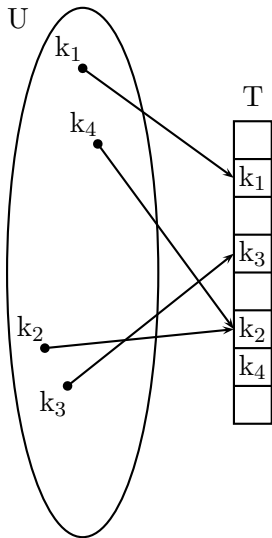
Open-Address Hash Table



Open-Address Hash Table



Open-Address Hash Table



Hash-Insert(T, k)

```
1   $j = h(k)$ 
2  for  $i = 1$  to  $T.length$ 
3      if  $T[j] == \text{nil}$ 
4           $T[j] = k$ 
5          return  $j$ 
6      elseif  $j < T.length$ 
7           $j = j + 1$ 
8      else  $j = 1$ 
9  error "overflow"
```

Open-Addressing (3)

```
Hash-Insert(T, k)
1  for i = 1 to T.length
2    j = h(k, i)
3      if T[j] == nil
4          T[j] = k
5          return j
6  error "overflow"
```

Open-Addressing (3)

```
Hash-Insert(T, k)
1  for i = 1 to T.length
2    j = h(k, i)
3    if T[j] == nil
4      T[j] = k
5    return j
6  error "overflow"
```

- Notice that $h(k, \cdot)$ must be a **permutation**
 - ▶ i.e., $h(k, 1), h(k, 2), \dots, h(k, |T|)$ must cover the entire table T

Procedure HASH-SEARCH

```
HASH-SEARCH( $T, k$ )  
1   $i \leftarrow 0$   
2  repeat  $j \leftarrow h(k, i)$   
3         if  $T[j] = k$   
4             then return  $j$   
5          $i \leftarrow i + 1$   
6  until  $T[j] = \text{NIL}$  or  $i = m$   
7  return NIL
```


Open-address hashing

- Deletion: difficult. Marking NIL does not work.
- Doing so might make it impossible to retrieve any key during whose insertion probed slot i and found it occupied.
- One solution: DELETED instead of NIL. Problem: search time no longer dependent on load factor.
- Techniques for probing: linear probing, quadratic probing and double hashing.
- Linear probing: given auxiliary hash function $h' : U \rightarrow \{0, \dots, m - 1\}$, use hash function

$$h(k, i) = (h'(k) + i) \bmod m.$$

- Easy to implement but suffers from problem called primary clustering.
- Long runs of occupied slots build up, increasing average search time.

- Quadratic probing

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m.$$

- Works much better than linear probing, but to make use of full hash table the values of c_1, c_2, m are constrained.
- Suffers from **secondary clustering**: if two keys have the same initial probe position then their probe sequences are the same.

- Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$

- Among the best methods for open addressing.
- $h_2(k)$ must be relative prime to m . One solution is m a power of two and $h_2(k)$ odd.
- Another one: m prime, $h_2(k) < m$.
- Given an open address hash table with load factor $\alpha = n/m < 1$ the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$, assuming uniform hashing.

Double hashing

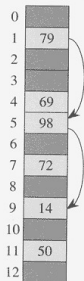


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Good hash functions

Caution

- The area of designing good hash function **huge**.
- Theoreticians and practitioners as well.
- Many hash functions **good for specific goal**.
- **Appearing next does not mean you should blindly use them !**
- Drozdek: discusses some more "practical" examples.
- Here: we follow CORMEN, concentrate on general ideas.

Requirements

- A good hash function satisfies (approximately) the assumption of uniform hashing.
- Unfortunately, usually we don't know probability distribution of the keys, and keys might not be drawn independently.

Good hash functions

- **Good case:** if items are random real numbers k uniformly distributed in $[0, 1)$, $h(k) = \lfloor km \rfloor$ satisfies simple uniform hashing conditions.
- Most hash functions assume universe of keys are the natural numbers.
- E.g. character string = integer in base 128 notation.
- Identifier pt. ASCII $p = 112$, $t = 116$, becomes $112 \cdot 128 + 116 = 14452$.
- **Division method:** $h(k) = k \bmod m$. Avoid some values of m , e.g. powers of two. Indeed, if $m = 2^p$ then $h(k)$ = the p lowest bits of k . Unless we know that p lowest bits of keys are uniform not a good idea.
- Prime not too close to an exact power of two = often a good choice.

- Input: broken into pieces.
- Combined in some way.

Example

- SSN (American CNP): 123456789.
- Divide into three parts: 123, 456, 789.
- Add these: 1368.
- Reduced modulo table-size (1000): 368

Good hash functions: Multiplication method

- Multiplication method: Two stage procedure
- First, multiply key by constant A in range $0 < A < 1$, extract fractional part.
- Then multiply by m , extract floor.
- $h(k) = \lfloor m(kA \bmod 1) \rfloor$.
- Value of m not critical. $m = 2^p$.
- Easy implementation. Restrict $A = s/2^w$, w =machine word size.
- Better with some values of A than other. Knuth suggests $A \sim (\sqrt{5} - 1)/2$ will work well.

Multiplication-method of hashing

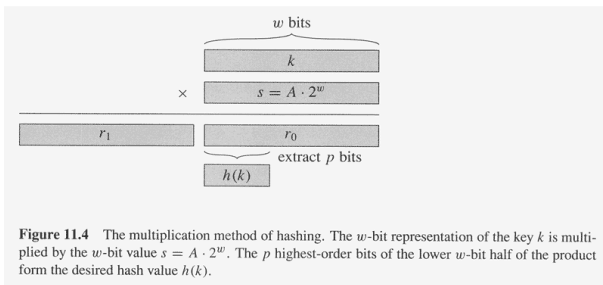


Figure 11.4 The multiplication method of hashing. The w -bit representation of the key k is multiplied by the w -bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w -bit half of the product form the desired hash value $h(k)$.

- Any fixed hash function vulnerable to worst case behavior:
- if "adversary" chooses n keys that all hash to the same slot, this yields average search time of $\theta(n)$.
- Practical example: Crosby and Wallach (USENIX'03) have shown that one can slow down to a halt systems by attacking implementations of hash tables in Perl, squid web proxy, Bro intrusion detection.
- Cause: hashing mechanism known (due to, e.g. publicly available implementation).
- Solution: choose hash function randomly, independent of the keys that are going to be stored.

Attack when hashing mechanism known

dos.pdf

Universal hashing

- \mathcal{H} finite collection of hash functions that map universe U into $\{0, 1, \dots, m - 1\}$.
- Such a collection is called **universal** if for every keys $k \neq l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$.
- Suppose a hash function h is chosen from a universal collection of hash functions, and is used to hash n keys into a table T of size m (using chaining).
- If key is not in the table expected length of the list that k hashes to is at most α .
- If key is not in the table expected length of the list that k hashes to is at most $1 + \alpha$.

A universal class of hash functions

- Due to Carter and Wegman.
- $a \equiv b \pmod{p}$ if $p|(a - b)$.
- Z_p : integers modulo p . p prime.
- How do we choose p ? So that all keys are in the range 0 to $p - 1$.
- m : number of slots in the hash table.
- $a \in Z_p^*$, $b \in Z_p$.
- $h_{a,b}(k) = ((ak + b) \pmod{p}) \pmod{m}$.
- $\mathcal{H}_{p,m} = \{h_{a,b} : a \in Z_p^*, b \in Z_p\}$.
- Other applications of this set of hash functions: pseudo-random generators.

Perfect hashing

- Hashing can provide **worst-case** performance when the set of keys is **static**: once stored in the table, the set of keys never changes.
- Example: set of files on a DVD-R (finished).
- **Perfect hashing**: the worst-case number of accesses to perform a search is $O(1)$.
- Idea: **two-level hashing with universal hashing at each level**.
- **First level**: the n keys are hashed into m slots using a hashing function chosen from a family of universal hash functions.
- Instead of chaining: **Use (small) secondary table S_j with an associated hash function h_j** .
- **Choosing h_j carefully guarantees no collisions**.

Perfect hashing with chaining

fig1-9-eps-converted-to.pdf

Perfect hashing: design

- n_j = number of elements that hash to slot j .
- We let $m_j = |S_j| = n_j^2$.
- Idea: if $m = n^2$ and we store n keys in a table of size $m = n^2$ using a hash function randomly chosen from a set of universal hash function then the collision probability is at most $1/2$.
- Find a good hash function using $O(1)$ trials.
- Expected amount of memory $O(n)$.
- Why this works: proof omitted (see Cormen if curious).

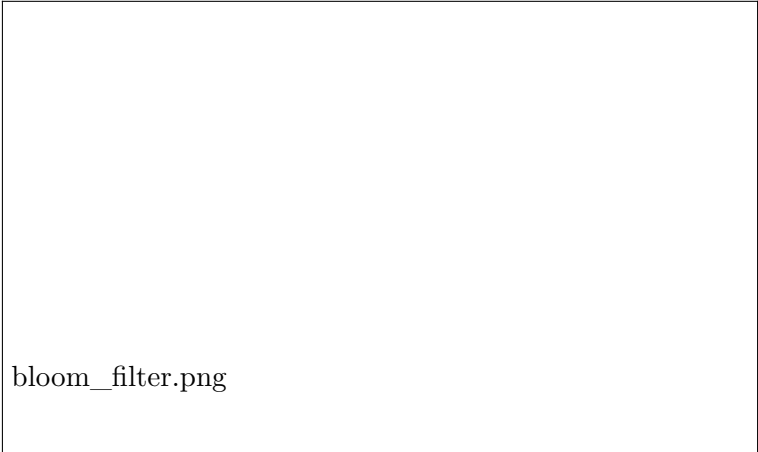
Hashing: there is more

- **Cryptographic hash functions:** hash functions with good security properties.
- Most well-known cryptographic hash function: **md5** (Rabin). You probably have encountered it if you downloaded anything large from the web.
- (sha-1), sha-2, sha-3.
- U.S. Government standards.
- (Some) attacks on sha-1 (CWI Amsterdam, 2017)

SHA1("The quick brown fox jumps over the lazy dog") gives
hexadecimal: 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12 gives
Base64 binary to ASCII text encoding:
L9ThxnotKPzthJ7hu3bnORuT6xI=

Bloom filters

- Probabilistic data structure. Used to test membership of an element in a dataset.
- NO answer: correct
- YES answer: possibly false positive.



bloom_filter.png

Hashing: where to go from here

- MapReduce model for grid computing: **programming with hash functions**.
- Data: **key-value pairs**.
- Map: applied in parallel to every pair (keyed by k1). Produces a list of pairs (keyed by k2) for each call.
- Mapreduce collects all pairs with the same k2 and groups them together.
- Reduce: applied in parallel to each group, which in turn produces a collection of values in the same domain.

Hashing: where to go from here

- **Locality-sensitive hashing**: reduces the dimensionality of high-dimensional data. LSH hashes input items so that similar items map to the same 'buckets' with high probability.



mining.jpg

Hashing in programming languages

- Python: dictionaries.
- Hash tables in STL: `std::hash`.
- Also: two implementations, `std::unordered_map` and `std::unordered_set`.