# Lecture 8: Dictionaries and Hash Tables

Instructor: Saravanan Thirumuruganathan

# Outline

# In-Class Quizzes

- **URL:** `http://m.socrative.com/`
- **Room Name: 4f2bb99e**

# Dictionary ADT

- Stores key-value pairs
- Required Operations:
  - Insert
  - Search (Membership check)
  - Delete

## Caller ID Implementation:

- **Objective:** Given phone number, output Caller's name
- Assume we need to worry about callers from Arlington only
- What is the universe/input space?

# Motivation - I

## Caller ID Implementation:

- **Objective:** Given phone number, output Caller's name
- Assume we need to worry about callers from Arlington only
- What is the universe/input space?
    - Ignore first three digits (why?)
    - Last 7 digits can input numbers between 0 to $10^7 - 1$
    - Number of phone numbers in Arlington way less than $10^7 - 1$

# Motivation - II

## Student ID Lookup:

- **Objective:** Given student id, retrieve student information
- Example: UTA graduate school, TA of this course
- What is the universe/input space?

# Motivation - II

## Student ID Lookup:

- **Objective:** Given student id, retrieve student information
- Example: UTA graduate school, TA of this course
- What is the universe/input space?
  - Ignore four digits (why?)
  - Last 6 digits can input numbers between 0 to $10^6 - 1$
  - Number of students in UTA/5311 is way less than $10^6$

# Potential Implementations

**Possible Candidates:**

# Potential Implementations

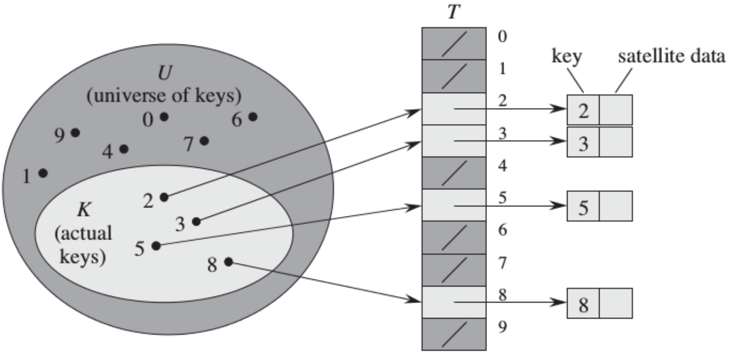
**Possible Candidates:**

- Linked List based
- Array based
- Balanced trees

# Space Vs Time Tradeoff

- All our previous implementations optimized for time given linear storage cost
- What if time is more important than space?
- Think of companies like Google, Facebook, Amazon, AT&T etc

# Direct Address Tables[1]




[1] CLRS Fig 11.1

# Direct Address Tables

```
DAT-Search(T,k):
    return T[k]

DAT-Insert(T,x):
    T[x.key] = x

DAT-Delete(T,x):
    T[x.key] = NULL
```

# Direct Address Tables

- Represent input in an array
- Each position/slot corresponds to a key in universe $U$
- Works well when $U$ is small
- Pro:

# Direct Address Tables

- Represent input in an array
- Each position/slot corresponds to a key in universe $U$
- Works well when $U$ is small
- Pro: Fast
- Con:

# Direct Address Tables

- Represent input in an array
- Each position/slot corresponds to a key in universe $U$
- Works well when $U$ is small
- Pro: Fast
- Con: Lot of space is wasted

# Ideas to Improve DAT

- Let size of universe be $N$
- Let Space budget be $m$ (for eg, $c \cdot \#$ max elements)

# Ideas to Improve DAT

- Let size of universe be $N$
- Let Space budget be $m$ (for eg, $c \cdot \#$ max elements)
- Let $\#$ elements inserted be $n$
- Caller ID Eg: $10^7 - 1$ vs $400K$ (size of Arlington)
- Student ID Eg: $10^6$ Vs 8000
- 5311 Eg: $10^6$ vs 50

# Ideas to Improve DAT

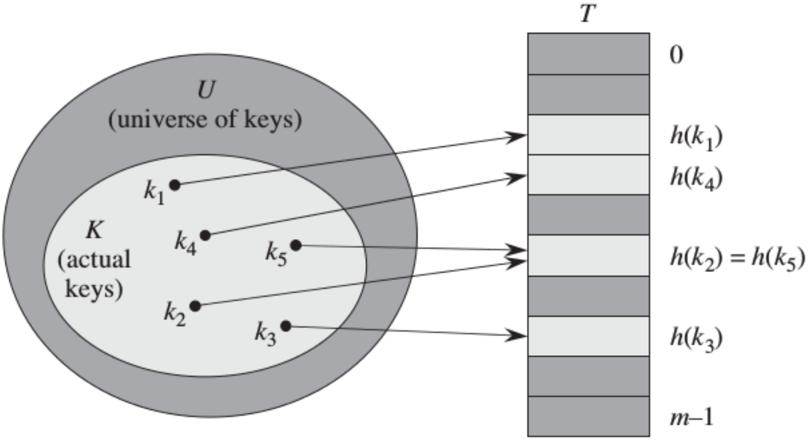
- Let size of universe be $N$
- Let Space budget be $m$ (for eg, $c \cdot \#$ max elements)
- Let $\#$ elements inserted be $n$
- Caller ID Eg: $10^7 - 1$ vs $400K$ (size of Arlington)
- Student ID Eg: $10^6$ Vs 8000
- 5311 Eg: $10^6$ vs 50
- **Insight:** Try to have space proportional to $m$ instead of $N$

# Hash Tables

# Hash Functions

- **Hash Function $h$:** Compute an array index from key value
- **Input:** $1..N$
- **Output:** $0..m-1$
- Formally, $h : U \to \{0, 1, \ldots, m-1\}$
- **Requirement:** (Ideal): Uniformly scramble elements across array
  - Efficient to compute (so peeking into array)
  - Each array position is uniformly likely

# Hash Table[2]




[2] CLRS Fig 11.2

# Hash Function Design: Student ID Example

- Space budget is $m = 100$ (array with 100 slots)
  - Objective: Design hash function $h(student\ id) \in \{0, 1, \dots, 99\}$

# Hash Function Design: Student ID Example

- Space budget is $m = 100$ (array with 100 slots)
  - Objective: Design hash function $h(student\ id) \in \{0, 1, \ldots, 99\}$
  - Last two digits of Student ID
  - Student ID be $h(1000 - 000 - 188) \Rightarrow 88$
  - Any two students with last two digits 88?
- Space budget is $m = 1000$ (array with 1000 slots)

# Hash Function Design: Student ID Example

- Space budget is $m = 100$ (array with 100 slots)
  - Objective: Design hash function $h(student\ id) \in \{0, 1, \ldots, 99\}$
  - Last two digits of Student ID
  - Student ID be $h(1000 - 000 - 188) \Rightarrow 88$
  - Any two students with last two digits 88?
- Space budget is $m = 1000$ (array with 1000 slots)
  - Objective: Design function $h(student\ id) \in \{0, 1, \ldots, 999\}$

# Hash Function Design: Student ID Example

- Space budget is $m = 100$ (array with 100 slots)
  - Objective: Design hash function $h(\mathit{student\ id}) \in \{0, 1, \ldots, 99\}$
  - Last two digits of Student ID
  - Student ID be $h(1000 - 000 - 188) \Rightarrow 88$
  - Any two students with last two digits 88?
- Space budget is $m = 1000$ (array with 1000 slots)
  - Objective: Design function $h(\mathit{student\ id}) \in \{0, 1, \ldots, 999\}$
  - Last three digits of Student ID
  - Student ID be $h(1000 - 000 - 188) \Rightarrow 188$
  - Any two students with last two digits 188?
- Tradeoff between Space and Collisions

# Good and Bad Hash Functions

- 10-digit phone numbers
    - First three digits:

# Good and Bad Hash Functions

- 10-digit phone numbers
  - First three digits: Bad! (why?)
  - Last three digits:

# Good and Bad Hash Functions

- 10-digit phone numbers
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)
- 10-digit UTA student id
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)
- 9-digit SSN
  - First three digits:

# Good and Bad Hash Functions

- 10-digit phone numbers
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)
- 10-digit UTA student id
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)
- 9-digit SSN
  - First three digits: Bad! (why?)
  - Last three digits:

# Good and Bad Hash Functions

- 10-digit phone numbers
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)
- 10-digit UTA student id
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)
- 9-digit SSN
  - First three digits: Bad! (why?)
  - Last three digits: Better (why?)

# Hash Function Design

- **Division/Modular:** $h(k) = k \mod m$
  - Alternative: Mod by a prime $P$
  - Java Strings: $P = 31$
- **Multiplication:** $h(k) = \lfloor m(kA \mod 1) \rfloor$
  - $0 < A < 1$
  - Take the fractional part and multiply it by $m$
- Universal hashing

# Hash Table: Sample Usecases

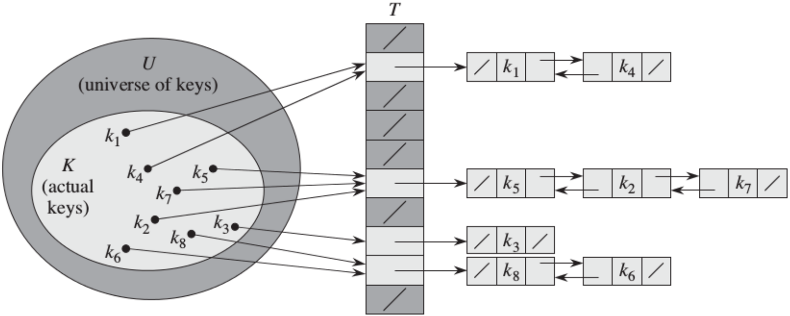
- Frequency of word in a document
- Check if any word in a set is an anagram of another

# Collisions

- When two items are hashed to same slot $h(k_i) = h(k_j)$
- Collision Resolution Techniques
  - Separate Chaining
  - Open Addressing: Linear probing, Quadratic probing, Double Hashing

# Separate Chaining[3]

- Idea: Place all elements that hash to same slot in a linked list




[3]CLRS Fig 11.3

# Separate Chaining

```
Chained-Hash-Insert(T,x):
    Insert x at head of linked list T[h(x.key)]

Chained-Hash-Search(T,k):
    Search for element with key k in T[h(k)]

Chained-Hash-Delete(T,x):
    Delete x from linked list T[h(x.key)]
```

# Open Addressing

- Separate Chaining used an external data structure to store all elements that collide
- Open Addressing
  - Do not use external storage (one element per slot)
  - Use hash table itself to store elements that collide
  - When a new key collides, find an empty slot and put it there
- Handling deletions is very messy - we will not discuss it here

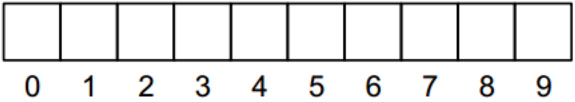
# Linear Probing

**Linear Probing:**

- Using hash function, map key to an array index (say $i$)
- Put element at slot $i$ if it is free
- If not try $i + 1$, $i + 2$, etc
- Roll around to start if needed

# Linear Probing: Example[4]

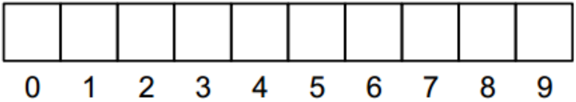
- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Resolve collisions via linear probing
- Hash function $h(k) = k\%10$ (i.e. take last digit)




[4] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[5]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$




- First three elements have no collisions

[5] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[5]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- First three elements have no collisions

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **70** | **81** | | | | | | **97** | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[5] `https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf`

# Linear Probing: Example[6]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 60

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | | | | | | 97 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[6] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[6]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 60

| 70 | 81 | | | | | | 97 | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Check slot 1 - it is full
- Check slot 2 - it is empty, so insert it

| 70 | 81 | 60 | | | | | 97 | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[6] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[7]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 51

| 70 | 81 | **60** | | | | | 97 | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[7] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[7]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 51

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | **60** | | | | | 97 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Check slot 2 - it is full
- Check slot 3 - it is empty, so insert it

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | **51** | | | | 97 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[7] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[8]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- No collisions when inserting 38 and 89

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | 51 | | | | 97 | **38** | **89** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[8] `https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf`

# Linear Probing: Example[9]

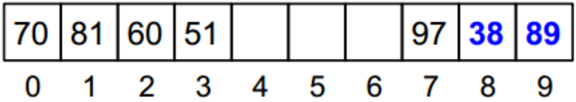
- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 68

| 70 | 81 | 60 | 51 | | | | 97 | **38** | **89** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[9] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[9]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 68




- Check slot 9 - it is full

[9] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[9]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 68

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | 51 | | | | 97 | **38** | **89** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Check slot 9 - it is full
- Wrap around: Check slots $0, 1, 2, 3$
- Insert 68 in slot 4

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | 51 | **68** | | | 97 | 38 | 89 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[9]https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[10]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 68

| 70 | 81 | 60 | 51 | **68** | | | 97 | 38 | 89 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[10] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[10]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 68

| 70 | 81 | 60 | 51 | **68** | | | 97 | 38 | 89 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Check slot 4 - it is full

[10] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[10]

- Objective: Insert elements $\langle 81, 70, 97, 60, 51, 38, 89, 68, 24 \rangle$
- Collision when inserting 68

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | 51 | **68** | | | 97 | 38 | 89 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Check slot 4 - it is full
- Insert 24 in slot 5

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | 51 | 68 | **24** | | 97 | 38 | 89 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

[10] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing: Example[11]

**Searching with Linear Probing:**

| 70 | 81 | 60 | 51 | 68 | 24 | | 97 | 38 | 89 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Easy Case: Search($T$, 81)
- Harder Case I: Search($T$, 60)
- Harder Case II: Search($T$, 68)
- Harder Case III: Search($T$, 80)

[11] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Linear Probing Issues: Primary Clustering[12]

- Remember the scenario of inserting 68

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 70 | 81 | 60 | 51 | | | | 97 | 38 | 89 |

- Had to travel long to find next empty slot
- Once collision happens, new keys are more likely to hash in middle of blocks
- So you have to spend more time to find an empty slot (extending the block size)
- You now increased the chance of a collision in the block!

[12] https://ece.uwaterloo.ca/~cmoreno/ece250/2012-01-30--hash_tables.pdf

# Fixing Primary Clustering

- Idea: Look for empty slots increasingly further away from original slot
- Probe Sequence: The order in which successive slots are checked
- Linear Probing: $h(k, i) = h'(k) + i$
- Probe sequence for Linear Probing:

$$h'(k), h'(k) + 1, h'(k) + 2, \ldots$$

# Fixing Primary Clustering

- Probe sequence for LP: $h'(k), h'(k)+1, h'(k)+2, \ldots$
- Quadratic probing: $h(k,i) = h'(k) + c_1 i + c_2 i^2$
  - Probe sequence when $c_1 = 0, c_2 = 1$,
    $$h'(k), h'(k)+1^2, h'(k)+2^2, \ldots$$
- Double Hashing
  - Choose two hash functions $h_1$ and $h_2$
  - Use $h_1$ first
  - If no collision, all is well
  - Else use the probing sequence
    $$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$
- Search: Follow same procedure till you find the element or an empty slot

# Hash Tables: Practical Advice I

- Load factor $\alpha = n/m$ (#elements / #table size)
    - Low $\alpha$: wasted space
    - High $\alpha$: long time for insert and search
- If you know $n$, pass it to Hash table (e.g. Java, Python)
    - The data structure will be much faster
    - For eg, most languages will set $m = \frac{4}{3}n$ (with $\alpha = \frac{3}{4}$)
- **Re-hashing:** Automatically adjusting number of budgets
    - If $\alpha$ becomes too low or high, **re-hashing** happens (it is bad!)

# Hash Tables: Practical Advice II

- Load factor $\alpha = n/m$ (#elements / #table size)
- Chaining can be used when $\alpha < 0.9$
- Linear probing is used when table is sparse ($\alpha \sim 0.5$)
- Double hashing is used when $\alpha < 0.66$
- With **good** hash functions, Hash table outperforms BST, RBT etc
- Double hashing: $h_2(k)$ can never be 0 (else you get infinite loop)

# Distributed Hash Tables (DHTs)

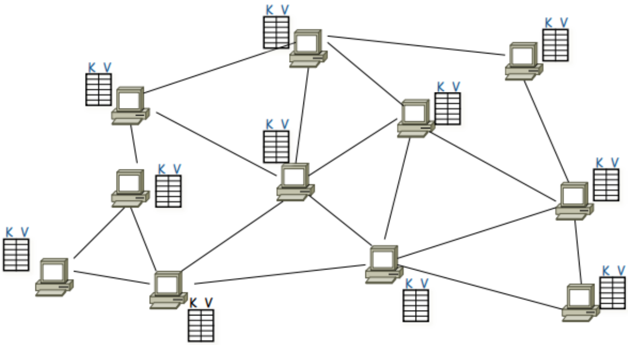
# Distributed Hash Tables (DHTs)

- **Idea:** Distribute the hash table content across many machines (typically a P2P network)
- **Motivation:**
  - Scalability: Eg. CDNs, NoSQL DBs
  - Fault Tolerance: Eg. Robust data archiving
  - Decentralization: Eg. BitTorrent
- **Issue:** We now have to determine which node to store data too!

**Applications:**

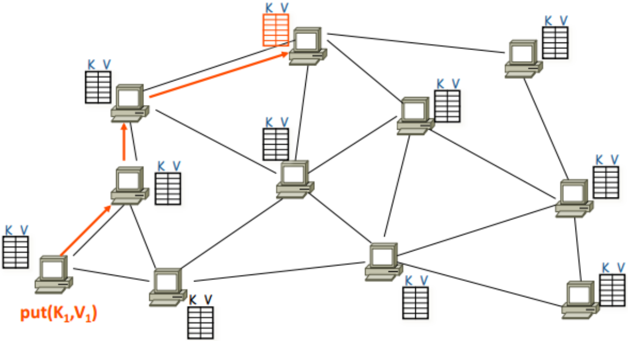
- Any internet scale application would have to use DHT
- Domain Name Service (hierarchical)
- File Sharing and Caching
- Archival/Retrieval of content (Eg. Dropbox: Deduplication)
- BitTorrent and other **trackerless** sharing sites
- Load balancing
- Anonymous web browsing
- Serverless email systems

# DHTs: Visualization[13]



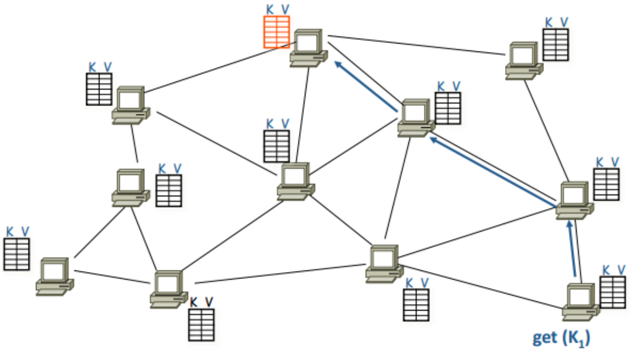

[13] http://www.cs.princeton.edu/courses/archive/spr09/cos461/docs/lec18-dhts.pdf

# DHTs: Put[14]




14 http://www.cs.princeton.edu/courses/archive/spr09/cos461/docs/lec18-dhts.pdf

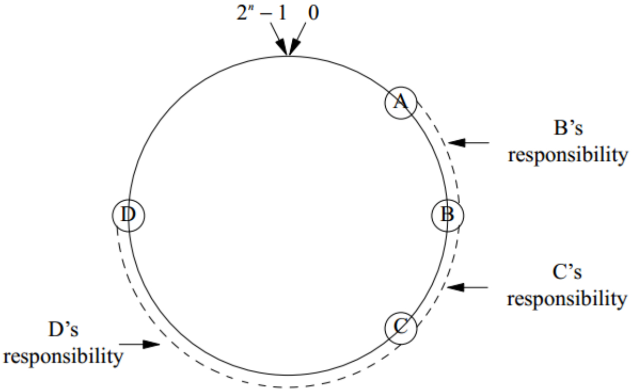
# DHTs: Get[15]




[15] http://www.cs.princeton.edu/courses/archive/spr09/cos461/docs/lec18-dhts.pdf

# DHTs: Sample Implementation[16]

**Chord Ring:** Routing, Joining, Replication




[16] http://www.ietf.org/old/2009/proceedings/06mar/slides/plenaryt-2.pdf

**Consistent Hashing function:**

- Special type of hashing function
- When $m$ (#slots) or $n$ (#item) is changed, at most $O(\lg n)$ items have to be moved
- Great idea for P2P systems with node arrival and removal

# Bloom Filters

# Bloom Filters

**Bloom Filters:**

- Probabilistic data structure - false positives (FP) but no false negatives (FN)
- Approximate set membership
- *Extremely* space efficient - $\sim$ 10 bits per element for 1% FP
- Only insert and membership check - no deletion
- Extensions for counting and deletion
- FP proportional to #elements - so becomes bad with more elements

# Bloom Filters: Applications

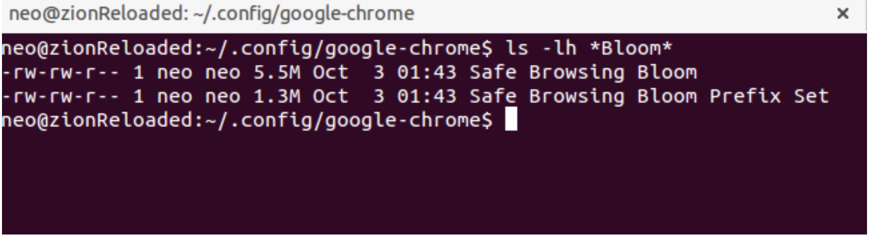
**Typical Scenarios:**

- Membership checking (yes or no) is sufficient
- Space is at a premium
- No need to list the items

**Applications:**

- Distributed Web caches (most popular application)
- Web proxy (such as Squid)
- Google BigTable and others
- Detecting malicious urls (Google SafeBrowse)
- Dictionary of weak passwords
- Spell-Check in phones

# Bloom Filter in Google Chrome



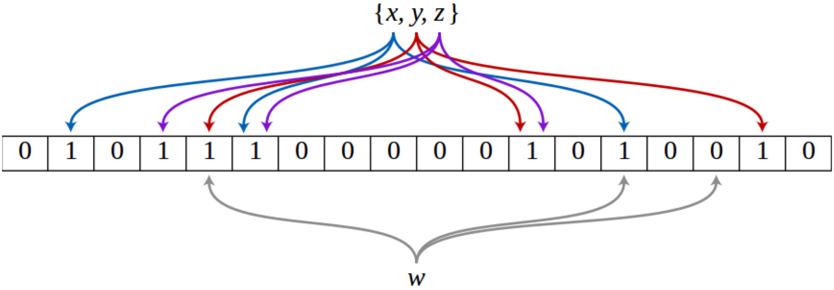

- Stores close to 1M malicious websites
- Windoze: C:\Users\%USERNAME%\AppData\Local\Google\Chrome\User Data\
- Mac: ~/Library/Application Support/Google/Chrome
- Linux: ~/.config/google-chrome

# High Level Idea

- Create an array $T$ with $m$ bits
- Select $k$ hashing functions that return value between $\{0, 1, \ldots, m-1\}$
- **Insert($x$):** $T[h_i(x)] = 1 \quad \forall i \in \{1, 2, \ldots, k\}$
- **Lookup($x$):**
  return $T[h_1(x)] == 1 \wedge T[h_2(x)] == 1 \wedge \ldots \wedge T[h_k(x)] == 1$

# Bloom Filter: Visualization[17]

**Settings:** $m = 18$ and $k = 3$




[17] http://en.wikipedia.org/wiki/Bloom_filter

# Summary

## Major Concepts:

- Dictionary ADT
- Hash Tables
- Hashing, Collision Resolution
- DHTs, Bloom Filters