Hash Tables

COMP215: Design & Analysis of Algorithms

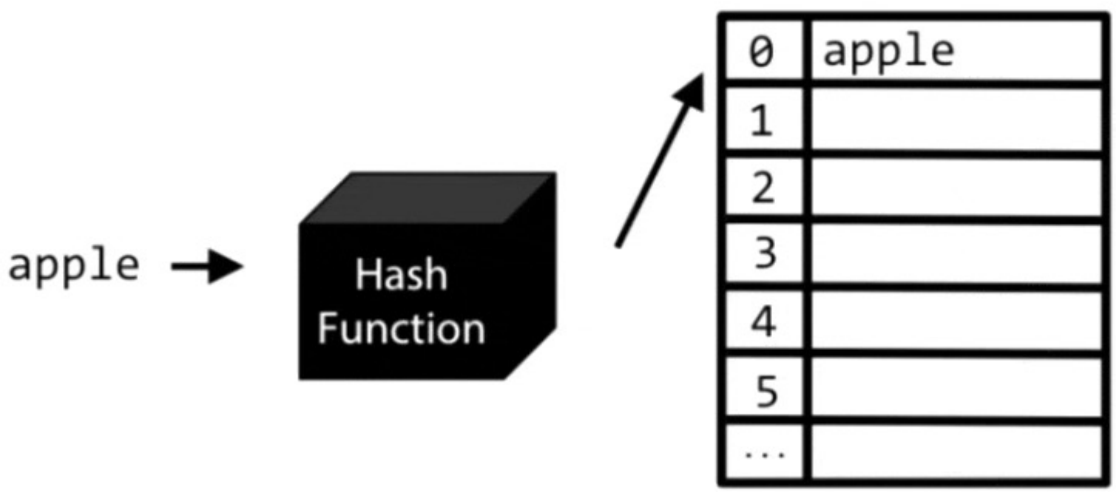
KALAMAZOO COLLEGE

Today

- What is Hash Table?
- Operations for Hash Table.
- Applications.

Hash Tables

- A hash table (or hash map) is a data structure that implements an associative array, mapping keys to values.
- Hash tables maintain an evolving set of objects associated with **keys**.
- They maintain **no ordering** information whatsoever.
- Hash tables facilitate **super-fast** searches, which are also called **lookups** in this context.
- **Key Idea:** It uses a **hash function** to compute an index (also called a **hash code** or hash) into an array of buckets or slots.



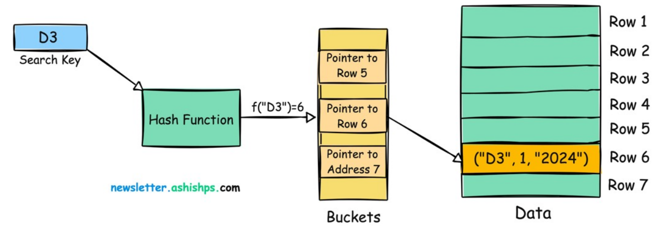
Hash Tables: Supported Operations

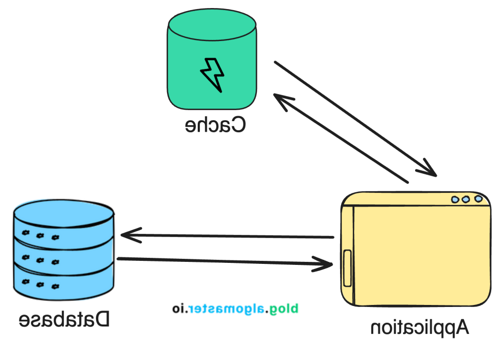
- **Lookup** (Search): for a key k, return a pointer to an object in the hash table with key k (or report that no such object exists).
  - Running time O(1)
- **Insert**: given a new object x, add x to the hash table.
  - Running time O(1)
- **Delete**: for a key k, delete an object with key k from the hash table, if one exists.
  - Running time O(1)

Hash Tables vs. Other Data Structure

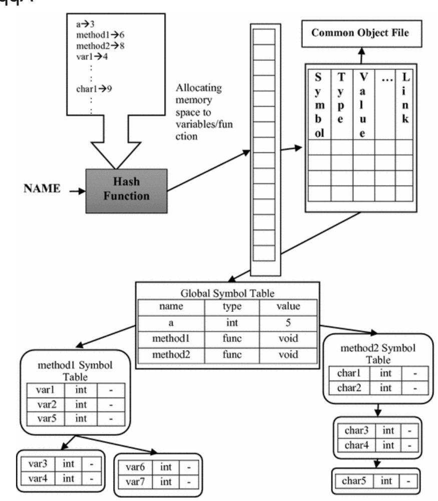
Feature	Hash Table	Array	Linked List	Binary Search Tree
Lookup Time	O(1)	O(n)	O(n)	O(log n)
Insert Time	O(1)	O(n)	O(1)	O(log n)
Delete Time	O(1)	O(n)	O(1)	O(log n)
Sorted Access	No	No	No	Yes

Applications





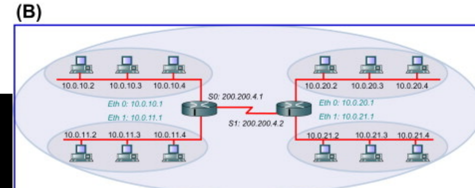
- **Databases Indexing**: Hash tables are used to index data in databases
- **Caching:** They allow for quick storage and retrieval of frequently accessed data.
- **Symbol Tables in Compilers:** Compilers use hash tables to keep track of variables, function names, and other identifiers.
- **Routing Tables in Networks:** Enabling efficient packet forwarding.



(A)

Learned	Network Address	Hop	Interface
C	10.0.10.0	0	Eth0
C	10.0.11.0	0	Eth1
C	200.200.4.0	0	S0
R	10.0.20.0	1	S0
R	10.0.21.0	1	S0

(B)



Applications

De-duplication

De-duplication with a Hash Table

When a new object x with key k arrives:

1. Use LOOKUP to check if the hash table already contains an object with key k.

2. If not, use INSERT to put x in the hash table.

Applications

The 2-SUM Problem

Problem: 2-SUM

Input: An unsorted array A of n integers, and a target integer t.

Goal: Determine whether or not there are two numbers x, y in A satisfying $x + y = t$.[2]

A = {1, 2, 3, 4, 5, 6, 7, 8, 9}
T = 12

2-SUM (Attempt #

Input: array A of n integers and a t
Output: "yes" if $A[i] + A[j] = t$ for
$i, j \in \{1, 2, 3, \ldots, n\}$, "no" otherwise

```
for i = 1 to n do
    y := t − A[i]
    if A contains y then        //
        return "yes"
return "no"
```

O(n^2)

2-SUM (Sorted Array Solution)

Input: array A of n integers and a target int
Output: "yes" if $A[i] + A[j] = t$ for some
$i, j \in \{1, 2, 3, \ldots, n\}$, "no" otherwise.

```
sort A                    // using a sorting sub
for i = 1 to n do
    y := t − A[i]
    if A contains y then          // binary
        return "yes"
return "no"
```

O(nlogn)

2-SUM (Hash Table Solution)

Input: array A of n integers and a target integer t.
Output: "yes" if $A[i] + A[j] = t$ for some
$i, j \in \{1, 2, 3, \ldots, n\}$, "no" otherwise.

```
H := empty hash table
for i = 1 to n do
    INSERT A[i] into H
for i = 1 to n do
    y := t − A[i]
    if H contains y then          // using LOOKUP
        return "yes"
return "no"
```

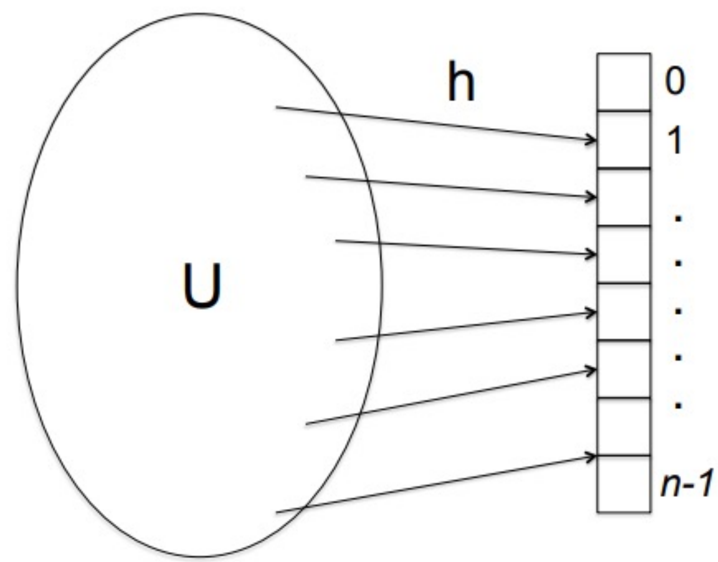
O(n)

Hash function

Hash Functions

A hash function $h : U \rightarrow \{0, 1, 2, \ldots, n-1\}$ assigns every key from the universe U to a position in an array of length n.

- A function that takes input (key) and returns a fixed-size integer (hash code).
- Function which hashes keys from the universe of potential keys to indices in the array
- Frequently makes use of modulo division
- **Examples of Hash Functions**:
  - Division Method:
    - **hash = key % table_size**
  - Multiplication Method:
    - **hash = (A * key % 1) * table_size**



Examples:

1. Given the keys {“Alice”, “Bob”, “Joseph”, “Chuck” }:
   Hash function: h(k) = len(k) – 1.

   Insert into an empty hash table of size 6

2. Given the Keys: 27, 43, 35, 7, 42, 56

   Hash function: hash(key) = key % table_size

   Insert into an empty hash table with size = 10.

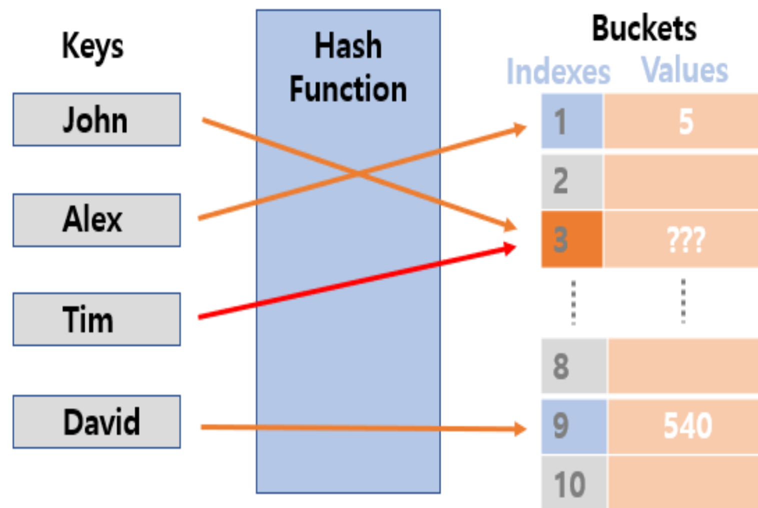
•

Collisions

Collisions

Two keys k_1 and k_2 from U *collide* under the hash function h if $h(k_1) = h(k_2)$.

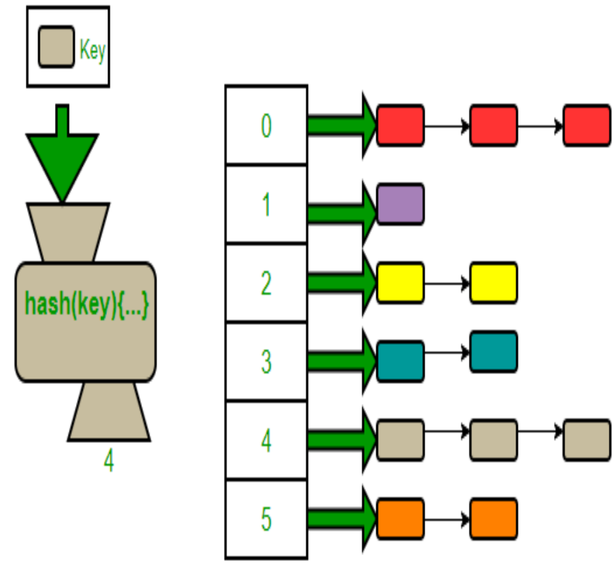
- **What is a Collision?**
  - Occurs when two keys hash to the same index.
- **Problem?**
  - Multiple values would map to the same location.
- **Collision Resolution Techniques:**
  - **Chaining**: Store multiple elements in a linked list at each index.
  - **Open Addressing**: Probe for the **next** available slot:
    - Probing Sequence
    - Double Hashing



Hash Collision

Collision Resolution: Chaining

- **Method**: Each index in the table points to a linked list (or another structure like a binary tree).
- **Example**: If two keys collide, their values are added to the list at that index.
    - **Pros**: Easy to implement, handles large data sets well.
    - **Cons**: Performance can degrade if the linked lists grow too long.

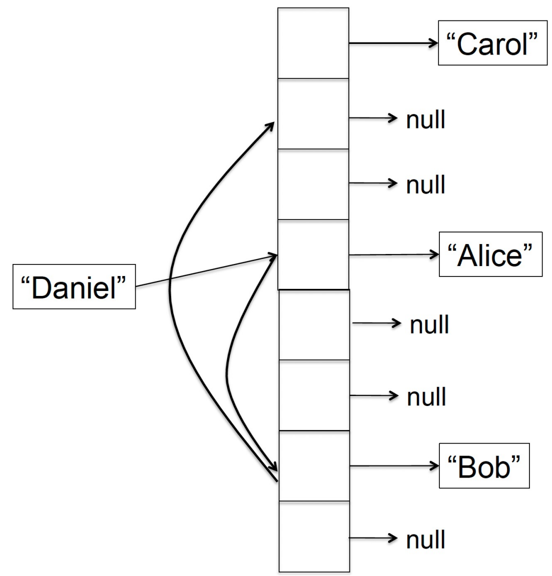


Chaining

1. Keep a linked list in each bucket of the hash table.
2. To LOOKUP/INSERT/DELETE an object with key k, perform LOOKUP/INSERT DELETE on the linked list in the bucket $A[h(k)]$, where h denotes the hash function and A the hash table's array.

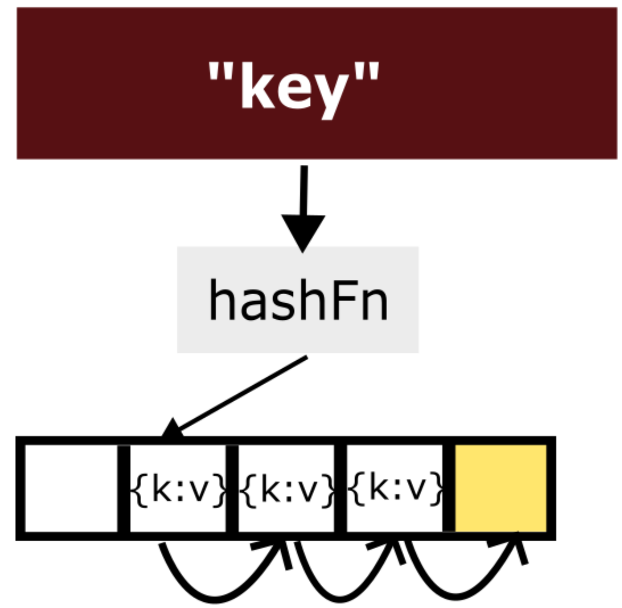
Collision Resolution: Open Addressing

- **Method**:
  - If a collision occurs, find another slot within the table by **probing**.
  - Each position of the array stores 0 or 1 objects, rather than a list



Open Addressing :Probe Sequences

- The object is stored in the first unoccupied position of its key’s probe sequence
- **Linear Probing:** This method uses one hash function h, and defines the probe sequence for a key k as **h(k)**,followed by **h(k)+1**, followed by **h(k)+2**, and so on.



Open Addressing

1. INSERT: Given an object with key k, iterate through the probe sequence associated with k, storing the object in the first empty position found.

2. LOOKUP: Given a key k, iterate through the probe sequence associated with k until encountering the desired object (in which case, return it) or an empty position (in which case, report “none”).[8]

Collision Resolution: Double Hashing

- Uses **two** hash functions.
  - The first **($h_1(k)$)** tells you the first position of the probe sequence
  - The second **($h_2(k)$)** indicates the offset for subsequent positions.

- For example, if **$h_1(k) = 15$** and **$h_2(k) = 22$:**
  - The first place to look for an object with key k is position 15; failing that:
    - position 37 (15+ 22); failing that,
      - position 59 (37+ 22); failing that,
        - Position 81 (59 +22); and so on.

Collisions

- *What Makes for a Good Hash Function?*

No matter which collision-resolution strategy we employ, hash table performance degrades with the number of collisions.

Pathological Data Sets

For *every* hash function $h : U \to \{0, 1, 2, \ldots, n-1\}$, there exists a set S of keys of size $|U|/n$ such that $h(k_1) = h(k_2)$ for every $k_1, k_2 \in S$.[10]

Load vs. Performance

$$\text{load of a hash table} = \frac{\text{number of objects stored}}{\text{array length } n}.$$

Which hash table strategy is feasible for loads larger than 1?

Idealized performance of a hash table as a function of its load α and its collision-resolution strategy

Collision-Resolution Strategy	Idealized Running Time of LOOKUP
Chaining	$O(\lceil \alpha \rceil)$
Double hashing	$O\left(\frac{1}{1-\alpha}\right)$
Linear probing	$O\left(\frac{1}{(1-\alpha)^2}\right)$

Comparison of Lookup Times (With Examples):

Here's a comparison of lookup times with $\alpha = 0.9$:

Strategy	Lookup Time	Example Explanation
Chaining	$O(\alpha)$, i.e., $O(0.9)$	Look through up to 1 key on average per bucket, only slight degradation with collisions.
Double Hashing	$O\left(\frac{1}{1-\alpha}\right)$, i.e., $O(10)$	Several probes needed to resolve collisions at high load factors (9 out of 10 slots filled).
Linear Probing	$O\left(\frac{1}{(1-\alpha)^2}\right)$, i.e., $O(100)$	Linear probing suffers due to clustering; high probing times near full capacity.