Data Structures

HASHING (CHAPTER 8)

Study Note (1/2)

1. **Definition (1–2 sentences):** What is it? What problem does it solve?
2. **Visualization (hand sketch):** Boxes/arrows for layout (e.g., array indices, linked-list nodes, stack top, queue front/back, tree levels, graph nodes/edges).
3. **Characteristics:**
   Ordering? Indexing? Allows duplicates? Dynamic size? Memory layout (contiguous vs pointers)? Typical operations.
4. **Time/Space (big-O, typical case):** Access/Search/Insert/Delete; extra memory.
5. **Limitations:** When it breaks down or is awkward to use.
6. **Pros/Cons:** When to use vs when not to use.
7. **Use cases:** 2–3 concrete scenarios.

Study Note (2/2)

1. **In class:**
   - Bring the printed template. 2) Fill the **definition**, **sketch**, and **key traits** as we go.
2. **After class (10–15 min):**
   - Add **big-O table**, **pros/cons**, **limitations**, and 2–3 **use cases**.
   - Snap a clear photo/scan of your sketch if needed.
3. **Commit to your repo:**
   - Add **contents** in your repo with structured layout

Dictionary

A book which explains or translates, usually in alphabetical order, the words of a language or languages (or of a particular category of vocabulary), giving for each word its typical spelling, an explanation of its meaning or meanings, and often other information, such as pronunciation, etymology, synonyms, equivalents in other languages, and illustrative examples.

by Oxford English Dictionary

A

¹a \ˈā\ *n, pl* **a's** *or* **as** \ˈāz\ **:** 1st letter of the alphabet

²a \ə, ˈā\ *indefinite article* **:** one or some — used to indicate an unspecified or unidentified individual

aard·vark \ˈärd͵värk\ *n* **:** ant-eating African mammal

aback \əˈbak\ *adv* **:** by surprise

aba·cus \ˈabəkəs\ *n, pl* **aba·ci** \ˈabə͵sī, -͵kē\ *or* **aba·cus·es** **:** calculating instrument using rows of beads

abaft \əˈbaft\ *adv* **:** toward or at the stern

ab·a·lo·ne \͵abəˈlōnē\ *n* **:** large edible shellfish

¹aban·don \əˈbandən\ *vb* **:** give up without intent to reclaim — **aban·don·ment** *n*

²abandon *n* **:** thorough yielding to impulses

aban·doned \əˈbandənd\ *adj* **:** morally unrestrained

abase \əˈbās\ *vb* **abased; abas·ing :** lower in dignity — **abase·ment** *n*

abash \əˈbash\ *vb* **:** embarrass — **abashment** *n*

abate \əˈbāt\ *vb* **abat·ed; abat·ing :** decrease or lessen

abate·ment \əˈbātmənt\ *n* **:** tax reduction

ab·at·toir \ˈabə͵twär\ *n* **:** slaughterhouse

ab·bess \ˈabəs\ *n* **:** head of a convent

ab·bey \ˈabē\ *n, pl* **-beys :** monastery or convent

ab·bot \ˈabət\ *n* **:** head of a monastery

ab·bre·vi·ate \əˈbrēvē͵āt\ *vb* **-at·ed; -at·ing :** shorten — **ab·bre·vi·a·tion** \ə͵brēvēˈāshən\ *n*

ab·di·cate \ˈabdi͵kāt\ *vb* **-cat·ed; -cat·ing :** renounce — **ab·di·ca·tion** \͵abdiˈkāshən\ *n*

ab·do·men \ˈabdəmən, abˈdōmən\ *n* **1 :** body area between chest and pelvis **2 :** hindmost part of an insect — **ab·dom·i·nal** \abˈdämənᵊl\ *adj* — **ab·dom·i·nal·ly** *adv*

ab·duct \abˈdəkt\ *vb* **:** kidnap — **ab·duc·tion** \-ˈdəkshən\ *n* — **ab·duc·tor** \-tər\ *n*

abed \əˈbed\ *adv or adj* **:** in bed

ab·er·ra·tion \͵abəˈrāshən\ *n* **:** deviation or distortion — **ab·er·rant** \aˈberənt\ *adj*

abet \əˈbet\ *vb* **-tt- :** incite or encourage — **abet·tor, abet·ter** \-ər\ *n*

Image credit: https://shop.merriam-webster.com/products/merriam-websters-concise-dictionary-large-print-edition?srsltid=AfmBOoocgYSutYaQ4T7GcVxysA3psclhmwea_BGr3KoUvvchdLiZ1Z4C

Dictionary: Characteristic

A **dictionary** is a collection of *word* $\rightarrow$ *meaning* pairs.

It helps us quickly find the meaning of a word without scanning every page.

It is organized by alphabetical order.

Index Page of a Book

INDEX

Italic page numbers indicate photos

Dictionary & Index Page

These entries in alphabetical order to make it easy for readers to quickly locate specific information.

Concept: Index (Key)

Index?

Question: Which data structure is inherently index-based?

Answer:

Index?

Question: Which data structure is inherently index-based?

Answer: **Array**

Fruit Dictionary: 2D Array

0	A	Abiu	Acai Palm	Apple	Argan	Atemoya	Avocado
1	B	Babaco	Banana	Bilimbi	Blueberry	Buddha's Hand	Burmese Grape
2	C	Cacao	Cainito	Calahash	Calamansi	Calamondin	Camu Camu
3	D						
4	E						
5	F						
6	G	Gac Fruit	Galia Melon	Genip	Giant Granadilla	Goji Berries	Grape
7	H						
8	I						
9	J	Jaboticaba Fruit	Jambolan	Jamaican Tangelo	Jackfruit	Japanese Plum	Jamaican Nutmeg

Fruit Dictionary: 2D Array

Pros

- …

Cons

- ...

Is there any new design can perform better?

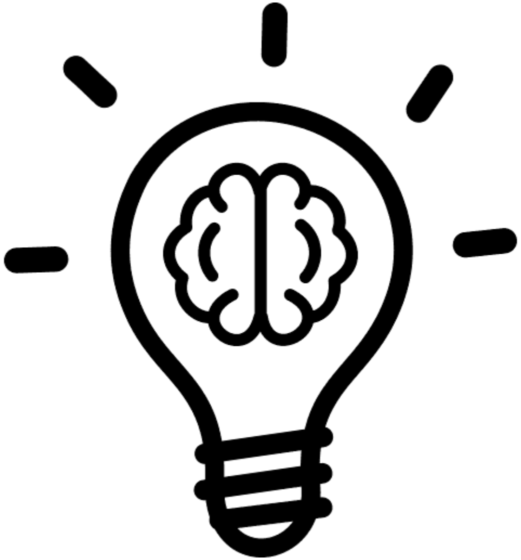
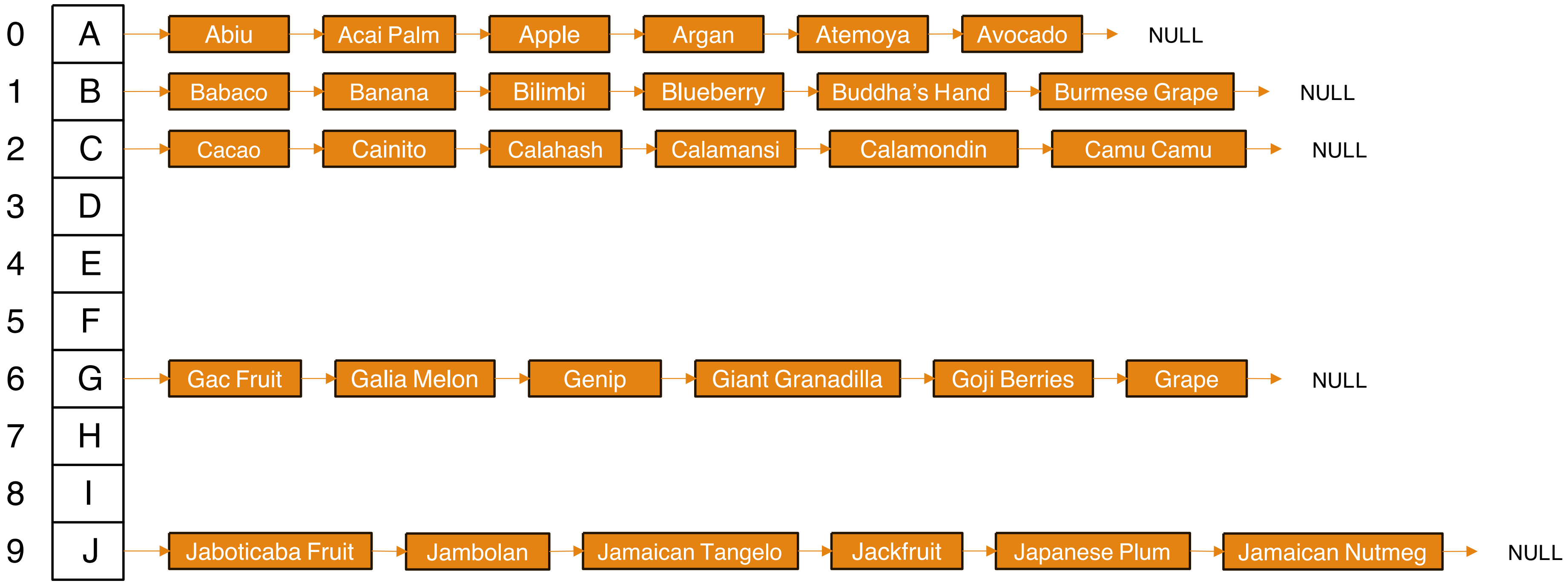


Image credit: https://uxwing.com/idea-icon/

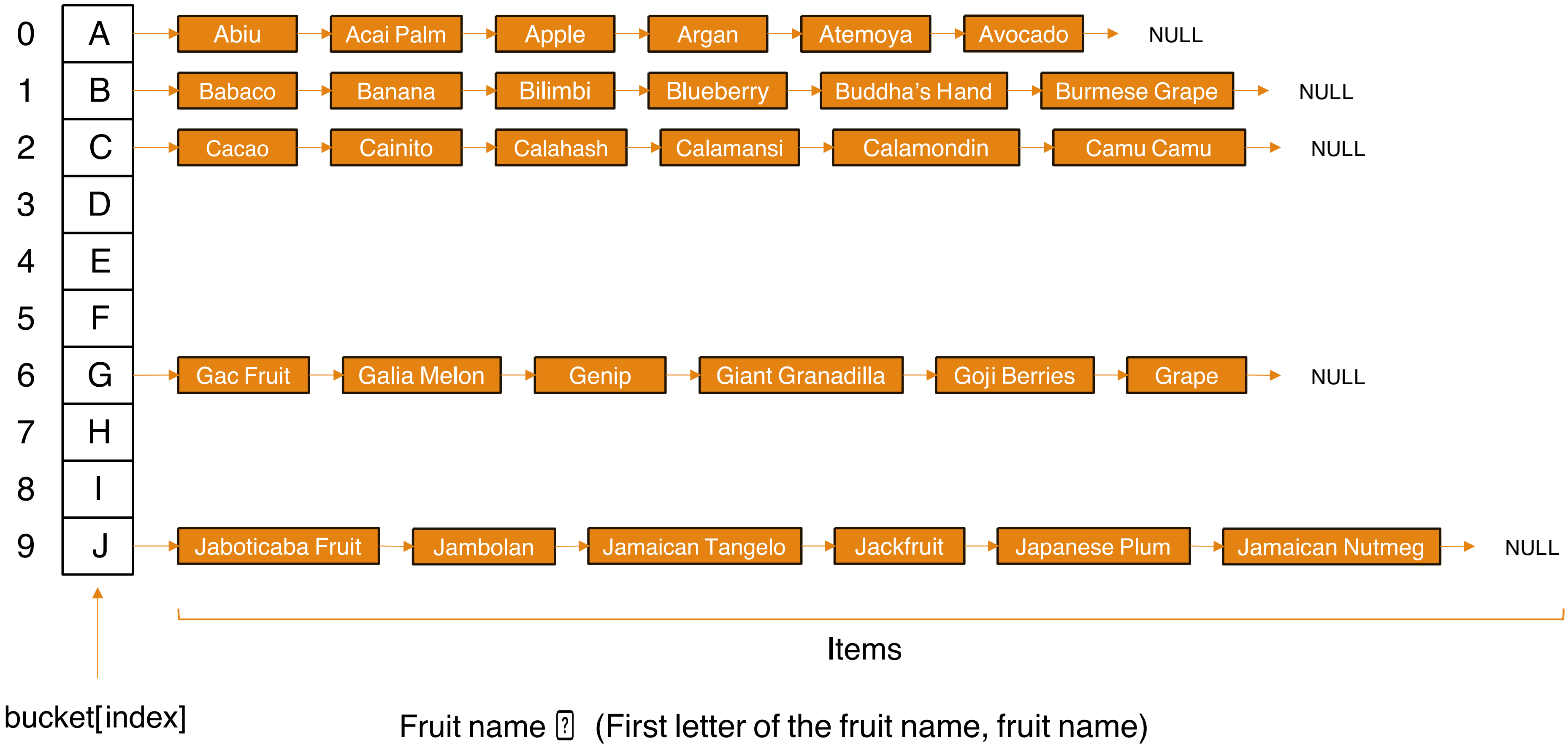
Fruit Dictionary



Fruit Dictionary

Components

- Index (e.g. A, B, C, D, …)
- List of items in some order



Fruit name ? ($\underbrace{\text{First letter of the fruit name}}_{\text{key}}$, $\underbrace{\text{fruit name}}_{\text{value}}$)

Hash Table

Underlying structure:

- Array: used for bucket storage
- Linked List (or other DS): handles collisions

Hybrid = Array + Linked List

Array index = bucket (from hash function)

Each bucket stores a linked list of items with same hash

Efficient Searching by Reducing Search Space

The key to efficiency: **eliminate unnecessary candidates early**.

Example:

- Hash Table
- Binary search on sorted array
- Binary search tree

Real-world Application: DNS Caching

Example.com <-> 192.0.0.15

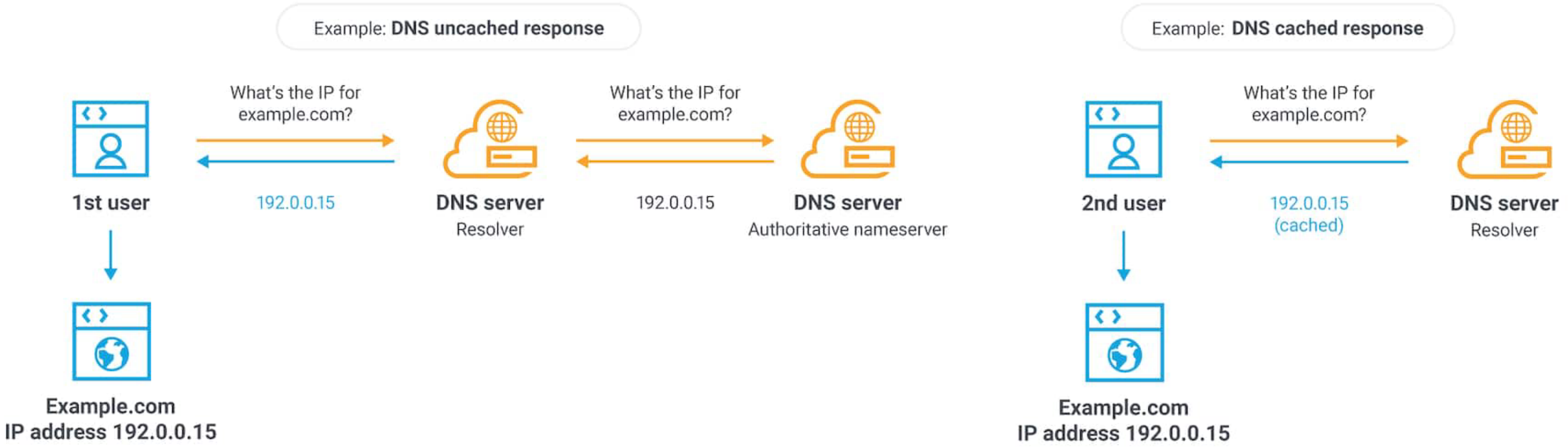


Image credit: https://www.akamai.com/glossary/what-is-dns-caching

Definition: Dictionary

A *dictionary* is a collection of pairs; each pair has a key and an associated item.

- No two pairs have the same key
- Several pairs with the same key

Key-Value Pair

$<Value_1, Value_2>$ Pair$<Key, Value_2>$

- $Value_2$ represents the important information (the data we want to retrieve).
- $Value_1$ provides the source to derive or compute the key

Hashing

A data structure that stores key-value pairs using a hash function to compute an index into an array of buckets.

This allows for average constant-time complexity for insertion, deletion, and lookup operations.

Hash tables handle collisions through techniques like chaining or open addressing, making them ideal for fast data retrieval (average O(1) time).

Concept	Description
Goal	Quickly find data by a *key*, without searching through all elements
Key-Value Mapping	`Key → Hash Function → Index → Value`
Hash Function	Mathematical rule that converts a key to a position in memory (array index)

Table Size

Prefer prime numbers for m to avoid repeating patterns.
Examples: 1009, 10007, 104729, …

For large tables (e.g. 2^{32} buckets), powers of two are fine only if the hash function mixes bits well.

Hash Function

A **hash function** (mapping function) converts a key into an **integer index**.

It should be:

- **Deterministic**: same key → same result
- **Uniform (diverse)**: spread keys evenly across indices
- **Efficient**: computed quickly

Approach

Scenario	Strategy	Description
Ideal Case	One-to-one mapping	Each `Value1` maps to a unique key, ensuring perfect lookup.
Collision Case	Collision handling	Several different `Value1` values map to the same key.

Collision Handling

1	Chaining	Maintain a list of $<\texttt{Value}_1, \texttt{Value}_2>$ pairs under the same key.
2	Open Addressing	Probe another slot (linear, quadratic, or double hashing).
3	Composite Key	Combine multiple attributes (e.g., $\texttt{Key} = \texttt{f}(\texttt{Value}_1, \texttt{Value}_2)$ or $\texttt{Key} = \texttt{f}(\texttt{Value}_1 + \texttt{timestamp})$) to increase uniqueness.
4	Hash Refinement	Redesign `f()` to use better bit-mixing or modulo a large prime number.

Summary

Component	**Meaning**
Key	Computed index derived from `Value`$_1$ using `f(Value`$_1$`)`
Value$_2$	The actual data or information we want
Collision	Multiple `Value`$_1$ mapping to the same key
Goal	Ensure uniqueness and efficient retrieval of `Value`$_2$

Key Concept

Component	Description
Hash Table	The array structure where data (key–value pairs) are stored.
Key	The data or identifier to be stored (e.g., student ID).
Hash Function	Converts the key into an index within the hash table.
Collision	When two keys map to the same index.
Load Factor (α)	`α = number of elements / table size` - measures how full the table is.

Hash Function

Method	Formula / Idea	Example
Division Method	`h(k) = k mod m`	key = 123, m = 10 → index = 3
Multiplication Method	`h(k) = floor(m * (k*A mod 1))`, $0 < A < 1$	$A \approx 0.618$
Folding Method	Split key into parts and add them	Key = 123456 → 12+34+56=102
String Hashing	Polynomial rolling hash	`h(s) = (Σ s[i] * p^i) mod m`

Approach to Managing Hash Table

Static

- **Static Hashing** means the **size of the hash table is fixed** when it is created.

Dynamic

- **Dynamic Hashing** allows the **hash table to grow or shrink automatically** as the number of records changes.
- The hash function or table structure can **adapt dynamically** to maintain good performance.

Comparison

Aspect	Static Hashing	Dynamic Hashing
Table Size	Fixed	Variable (grows/shrinks dynamically)
Hash Function	Constant	Adaptive (changes with size)
Memory Usage	Predictable	May expand dynamically
Performance ($\alpha\uparrow$)	Degrades with high load	Remains efficient
Rehashing	Entire table must be rebuilt	Only local bucket splits
Implementation	Simple	Complex (directory or pointer-based)
Best Use Case	Small, fixed datasets	Large or growing datasets

Collision (in Hashing)

A **collision** occurs in a hash table when **two or more different keys** are mapped by the hash function to the **same index (bucket)** in the table.

Collision = different keys, same hash address.

Several Pairs with the Same Key

Collison: two keys map to same index

Strategies:

- Chaining: Linked list at each bucket
- Open Addressing: Probe for next free slot

Example (chaining):

```
h(15) = 3  →  bucket[3] → [15]
h(23) = 3  →  bucket[3] → [15 -> 23]
h(7)  = 3  →  bucket[3] → [15 -> 23 -> 7]
```

Key Points

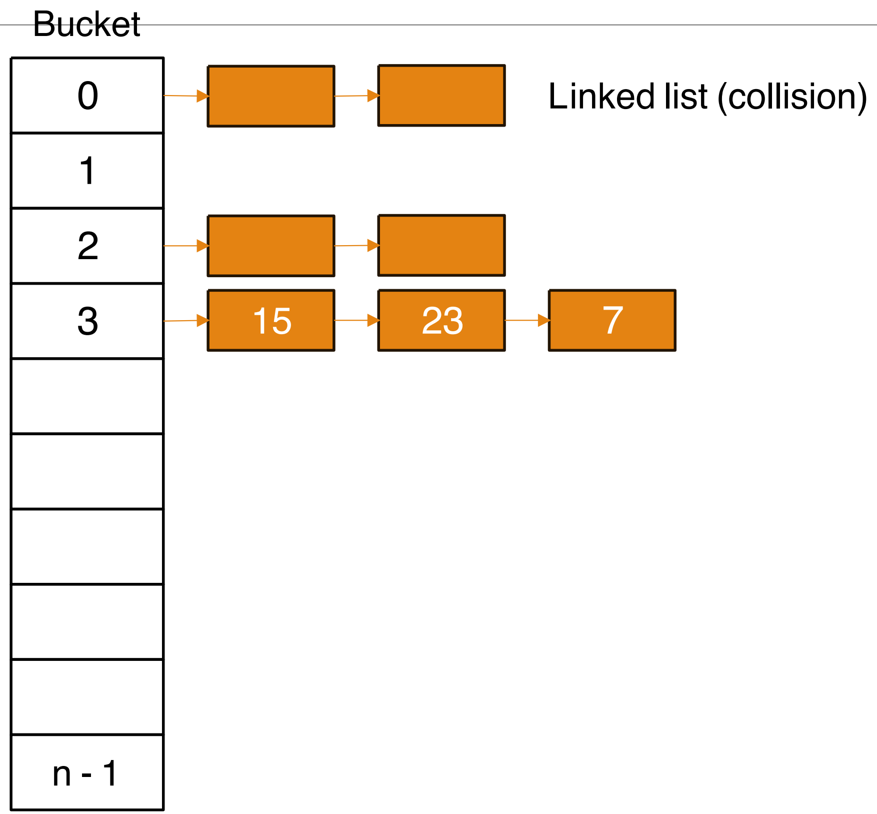
Aspect	**Description**
Collision Source	Multiple keys produce the same hash value.
Unavoidable	Unless the hash space ≥ number of unique keys (rare in practice).
Goal	Minimize collision frequency and resolve them efficiently.

Chaining (Separate Lists)

Each index keeps a linked list of all keys mapping to it.

Simple and flexible.

Hash Table



Hash(key) = key mod n

0, …, n-1

Open Addressing (Entire Array Implementation)

If an index is occupied, find another spot.

Methods

- Linear probing
- Quadratic probing
- Double hashing

What is Probing?

Probing is a **collision-resolution technique** used in **open addressing** hash tables.

When two or more keys map to the same hash index (collision), *probing* defines how the algorithm searches for the **next available slot** in the table.

Probing = systematic search for an empty position in a hash table after a collision.

Typing of Probing

Method	Formula	Behavior	Pros / Cons
Linear Probing	$(h(k) + i) \bmod m$	Check next slot sequentially.	Simple Primary clustering
Quadratic Probing	$(h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$	Gaps grow quadratically.	Reduces clustering May skip slots
Double Hashing	$(h_1(k) + i \cdot h_2(k)) \bmod m$	Uses a 2nd hash for step size.	Better spread More computation

i = probe sequence index (0, 1, 2, …)
hash function: $h(k)$, $h_1(k)$, $h_2(k)$
m: table size

Key Property

Property	Description
Deterministic	Same key always probes same sequence.
Bounded	Will examine at most m slots.
Cluster Formation	Some probing methods (e.g. linear) create contiguous filled regions, slowing performance.
Load Factor Sensitivity	As load factor ($\alpha = n/m$) increases, probe lengths and time complexity rise rapidly.

Summary

Term	Definition
Probing	Systematic process of finding the next available slot after a collision.
Purpose	To resolve collisions in open addressing.
Goal	Maintain efficient insertion, search, and deletion with minimal clustering.

Linear Probing

h(k) = k mod m (m = 10)

Collision: index(i) = (h(k) + i) mod m; i incremental while collision again

Probe Step	i	Computed Index	Explanation
1	0	(3 + 0) mod 10 = 3	Try slot[3] (occupied by 23)
2	1	(3 + 1) mod 10 = 4	Next slot[4]
3	2	(3 + 2) mod 10 = 5	Next slot[5]
4	3	(3 + 3) mod 10 = 6	Next slot[6]
...	...	...	Continues sequentially until empty slot

Observation: Slots are checked one by one → simple but causes primary clustering.

Linear Probing

h(k) = k mod m (m = 10)

Collision: index(i) = (h(k) + i) mod m

Insert keys: 23, 33, 43

Key	h(k)	Slot Status
23	3	slot[3] = 23
33	3	collision → probe to slot[4]
43	3	collision → slot[4] taken → probe slot[5]

Primary Clustering

Primary clustering is a phenomenon in **open addressing** hash tables (especially with **linear probing**) where **consecutive occupied slots form a cluster**, causing new keys to **probe longer sequences** and making the cluster grow even larger.

Primary clustering means that once a group of filled slots appears, it tends to attract even more insertions — forming a "crowded neighborhood."

Quadratic Probing

$h(k) = k \bmod 10$

Collision: $index(i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod 10$

Assume $c_1 = c_2 = 1$

Probe Step	i	Computed Index	Explanation
1	0	(3 + 0 + 0) mod 10 = 3	Try slot[3] (occupied)
2	1	(3 + 1 + 1) mod 10 = 5	Try slot[5]
3	2	(3 + 2 + 4) mod 10 = 9	Try slot[9]
4	3	(3 + 3 + 9) mod 10 = 5 → already used	Skip to next
5	4	(3 + 4 + 16) mod 10 = 3	Cycle repeats

Observation: Gaps grow quadratically → reduces clustering but can miss some slots (depends on table size and constants).

Quadratic Probing

$h(k) = k \bmod 10$

Collision: $\text{index}(i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod 10$

Assume $c_1 = c_2 = 1$

Insert keys: 23, 33, 43

Key	h(k)	i	$h(k) + i + i^2$	Slot Status
23	3	0	$(3 + 0 + 0) \bmod 10 = 3$	slot[3] = 23
33	3	1	$(3 + 1 + 1^2) \bmod 10 = 5$	collision → probe to slot[5] = 33
43	3	2	$(3 + 2 + 2^2) \bmod 10 = 9$	collision → slot[5] taken → probe slot[9] = 43

Secondary Clustering

Secondary clustering occurs in **open addressing** hash tables when **different keys** that hash to the **same initial index (h(k))** follow the **same probe sequence** during collision resolution.

$h(k) = k \bmod 10$

$index(i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod 10$

Double Hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 7 - (k \bmod 7)$

$index(i) = (h_1(k) + i \times h_2(k)) \bmod 10$

Probe Step	i	$h_2(k)$	Computed Index	Explanation
1	0	$7 - (23 \bmod 7) = 7 - 2 = 5$	$(3 + 0\times5) \bmod 10 = 3$	Try slot[3]
2	1	5	$(3 + 1\times5) \bmod 10 = 8$	Try slot[8]
3	2	5	$(3 + 2\times5) \bmod 10 = 3$	Cycle repeats after full wraparound

Double Hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 7 - (k \bmod 7)$

Collision: $index(i) = (h_1(k) + i \times h_2(k)) \bmod 10$

Insert keys: 23, 33, 43

Key	$h_1(k)$	i	$h_2(k)$	index(i)	slot	Slot Status
23	3	0	7 - (23 mod 7) = 7 – 2 = 5	3	3	slot[3] = 23

Double Hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 7 - (k \bmod 7)$

Collision: $index(i) = (h_1(k) + i \times h_2(k)) \bmod 10$

Insert keys: 23, 33, 43

Key	$h_1(k)$	i	$h_2(k)$	index(i)	slot	Slot Status
23	3	0	7 - (23 mod 7) = 7 – 2 = 5	3	3	slot[3] = 23
33	3	0	7 – (33 mod 7) = 7 – 5 = 2	3	3	slot[3] = 23, occupied
33	3	1	7 – (33 mod 7) = 7 – 5 = 2	5	5	slot[5] = 33

Double Hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 7 - (k \bmod 7)$

Collision: $index(i) = (h_1(k) + i \times h_2(k)) \bmod 10$

Insert keys: 23, 33, 43

Key	$h_1(k)$	i	$h_2(k)$	index(i)	slot	Slot Status
23	3	0	7 - (23 mod 7) = 7 – 2 = 5	3	3	slot[3] = 23
33	3	0	7 – (33 mod 7) = 7 – 5 = 2	3	3	slot[3] = 23, occupied
33	3	1	7 – (33 mod 7) = 7 – 5 = 2	5	5	slot[5] = 33
43	3	0	7 – (43 mod 7) = 7 – 1 = 6	3	3	slot[3] = 23, occupied
43	3	1	7 – (43 mod 7) = 7 – 1 = 6	9	9	slot[9] = 43

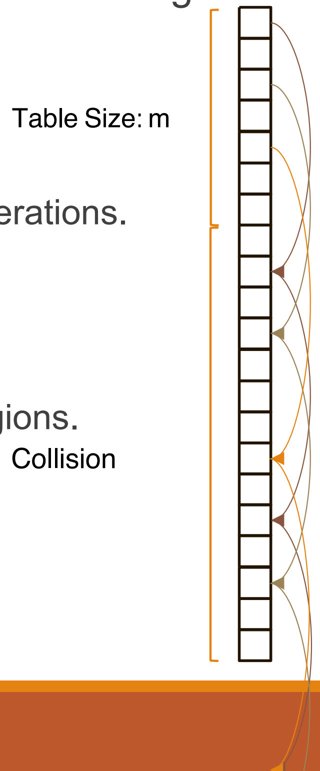
Observation:
Jump size, well-distributed across table. Low clustering and better performance for high load factors.

Open Addressing Hash Table

Observation:
Jump size, well-distributed across table. Low clustering and better performance for high load factors.

Performance issues

- Load Factor Sensitivity — Higher α increases collisions and probe length.
- Collision Frequency — Every collision triggers the probing mechanism, slowing operations.
- Clustering Effects
  - Primary clustering (linear probing)
  - Secondary clustering (quadratic probing)
- Non-uniform Key Distribution — Poor hash spread concentrates keys in certain regions.



Design of Hash Function

A **hash function** is a mathematical formula that converts a **key (data)** into a **table index**.

- A hash function tells you **"where to store"** and **"where to find"** data in the hash table.
- h(k) -> index
  - k = key (number, string, etc)
  - h(k) = hash value
  - index = h(k) mod m, where m is the table size.

Integer Keys

Division Method

$h(k) = k \bmod m$

- Divide the key by the table size (m)
- Use the remainder as the index.

Input keys: 23, 33, 42, 57

Key (k)	Table Size (m=10)	h(k) = k mod 10	Stored Index
23	10	3	3
33	10	3	3 (collision)
42	10	2	2
57	10	7	7

Good Practice: Choose `m` as a **prime number** close to table size to reduce patterns (e.g., 7, 11, 13, 31).

Large Integer Keys

Folding Method

- A hash function design that **divides a numeric key into equal-size parts** , then **adds or combines** those parts to produce the hash value.

$h(k) = (\Sigma \text{ parts of key}) \bmod m$

Key = 123456 → 12+34+56=102

h(123456) = 102 mod 10 = 2

Non-Integer Keys

For Character or String Keys

- Convert each character to its **ASCII code** and combine them.

Key: "CAT"

C = 67, A = 65, T = 84

Sum = 67 + 65 + 84 = 216

h("CAT") = 216 mod 10 = 6

The **decimal** set:

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

Non-Integer Keys

Weighted String Hash (better spread); Polynomial rolling hash

h(s) = (Σ s[i] * p^i) mod m

h("CAT") = $(C \times 31^2 + A \times 31^1 + T \times 31^0)$ mod m

Homework Assignment

Design Your Own Hash Function!!

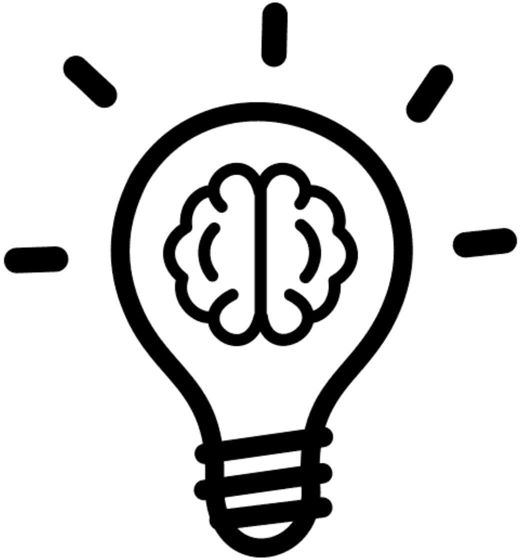


Image credit: https://uxwing.com/idea-icon/

Time Complexity

Separate Chaining

Operation	Best	Average	Worst	Remarks
Search	O(1)	$O(1 + \alpha)$	O(n)	Average-case constant if α small
Insert	O(1)	O(1)	O(n)	Append to short chain
Delete	O(1)	O(1)	O(n)	Search + unlink node

$T_{avg} \approx O(1+n/m) = O(1+\alpha)$

Time Complexity

Open Addressing

- Collisions resolved by probing (linear, quadratic, or double hashing).

Operation	Average ($\alpha \le 0.7$)	Worst	Notes
Search	O(1)	O(n)	At high load factor, probe chain length ↑
Insert	O(1)	O(n)	May require several probes
Delete	O(1)	O(n)	Needs careful slot marking (“ lazy delete”)

ADT: Dictionary

ADT *Dictionary* is

objects:

A collection of n > 0 pairs, each pair has a key and an associated item

functions:

for all $d \in Dictionary$, $item \in Item$, $k \in Key$, $n \in$ integer

Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(*d*, *n*)	::=	**if** (*n* > 0) **return** TRUE **else return** FALSE
Element Search(*d*, *k*)	::=	**return** *item* with key *k*. **return** NULL if no such element.
Element Delete(*d*, *k*)	::=	delete and return item (if any) with key *k*.
void Insert(d, item, k)	::=	insert *item* with key *k* into *d*.

end *Dictionary*

ADT: HashTable with Separate Chaining

ADT *HashTable* is

objects:

A finite set of pairs <key, value> where key is unique. Keys are distributed across m buckets using hash function h: key → [0, m-1]. Each bucket contains a chain (linked list) of key-value pairs.

parameters:

m: number of buckets (positive integer)

h: hash function (deterministic, uniform distribution)

λ: load factor = n/m where n = number of stored pairs

MAX_LOAD_FACTOR: threshold for triggering resize (default: 0.75)

functions:

for all $h \in HashTable$, $k \in Key$, $v \in Value$

HashTable Create(*m*)	::=	precondition: m > 0 postcondition: return empty hash table with m buckets, λ = 0
Boolean IsEmpty(*h*)	::=	**return** (size(*h*) == 0)
Insert(h, k, v)	::=	i = *h*(*k*) mod m if *k* exists in bucket[i]: replace existing value with v else: add < *k*, *v*> to front of bucket[*i*], increment size if λ > MAX_LOAD_FACTOR: resize(H, 2* m)
value Retrieve(h, k)	::=	i = h(k) mod m search bucket[i] for key k if found: return associated value **else** throw KeyNotFoundException
Boolean Delete(h, k)	::=	i = h(k) mod m if k exists in bucket[i]: remove <k,v>, decrement size, **return** TRUE **else return** false
Boolean Search(h, k)	::=	i = h(k) mod m **return** (k exists in bucket[i])
Iterator Traverse(h)	::=	**return** iterator that visits all key-value pairs order: bucket[0] to bucket[m-1], within bucket: insertion order

end *HashTable*