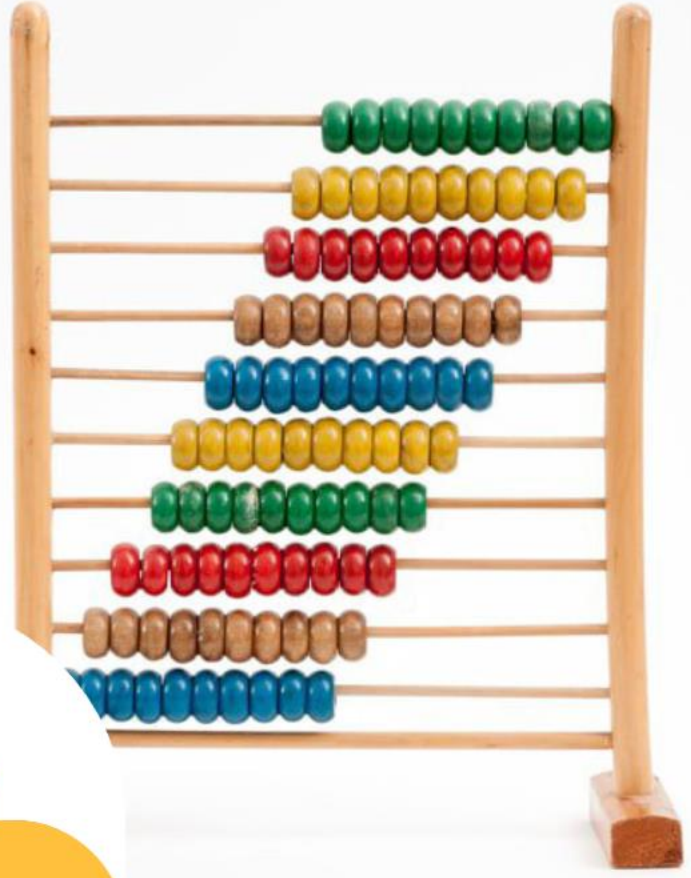
Fast Key Access with Hashing

Why hash tables solve exact-match performance
Map arbitrary keys to fixed indices for expected O(1) lookup, insert, delete; focus on algorithmic hashing, not cryptography.



Abdulaziz Shamsiev
Presenter

Hashing Basics: Keys, Functions, Values, and Indices

How keys map deterministically and uniformly into table slots

Key

01

Arbitrary identifier such as string, integer, or object used for store or lookup

Hash function

02

Deterministic function that converts the key to an integer hash value

Hash value

03

Integer output that may be large and must be reduced to an index range

Index (compress)

04

Reduce hash value to slot range using hash_value mod table_size or alternative compression

Determinism

05

Same key must always map to the same index within a given table configuration

Uniformity

06

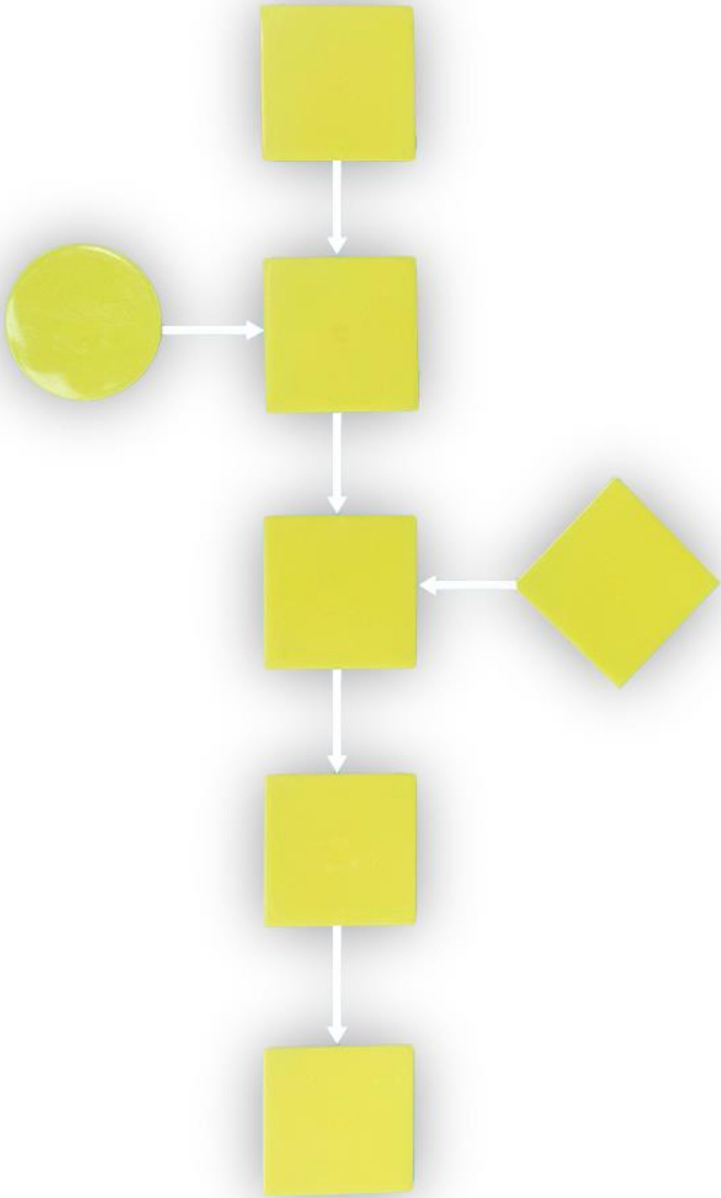
Good hash functions spread keys evenly across indices to reduce clustering

Hash tables

	0	1	2	3	4	5
Values	Ahmed	Ali	John	Jane	Mehmed	Ali
Keys	a10	a11	a12	a13	a14	a15

Collisions in Hash Tables: why they happen and why they matter

Pigeonhole inevitability and measurable performance cost



Why they happen

Pigeonhole principle: finite table slots vs unlimited keys guarantees collisions

Immediate implication

Distinct keys map to same index; extra work needed to remain correct

Performance impact

Longer probes or larger buckets push average time from O(1) toward O(n)

Practical note

Small load factor increases sharply raise collision rate and latency

Memory Layout and Cache Trade-offs for Hash Tables

Compare separate chaining versus open addressing for in-memory performance

VS

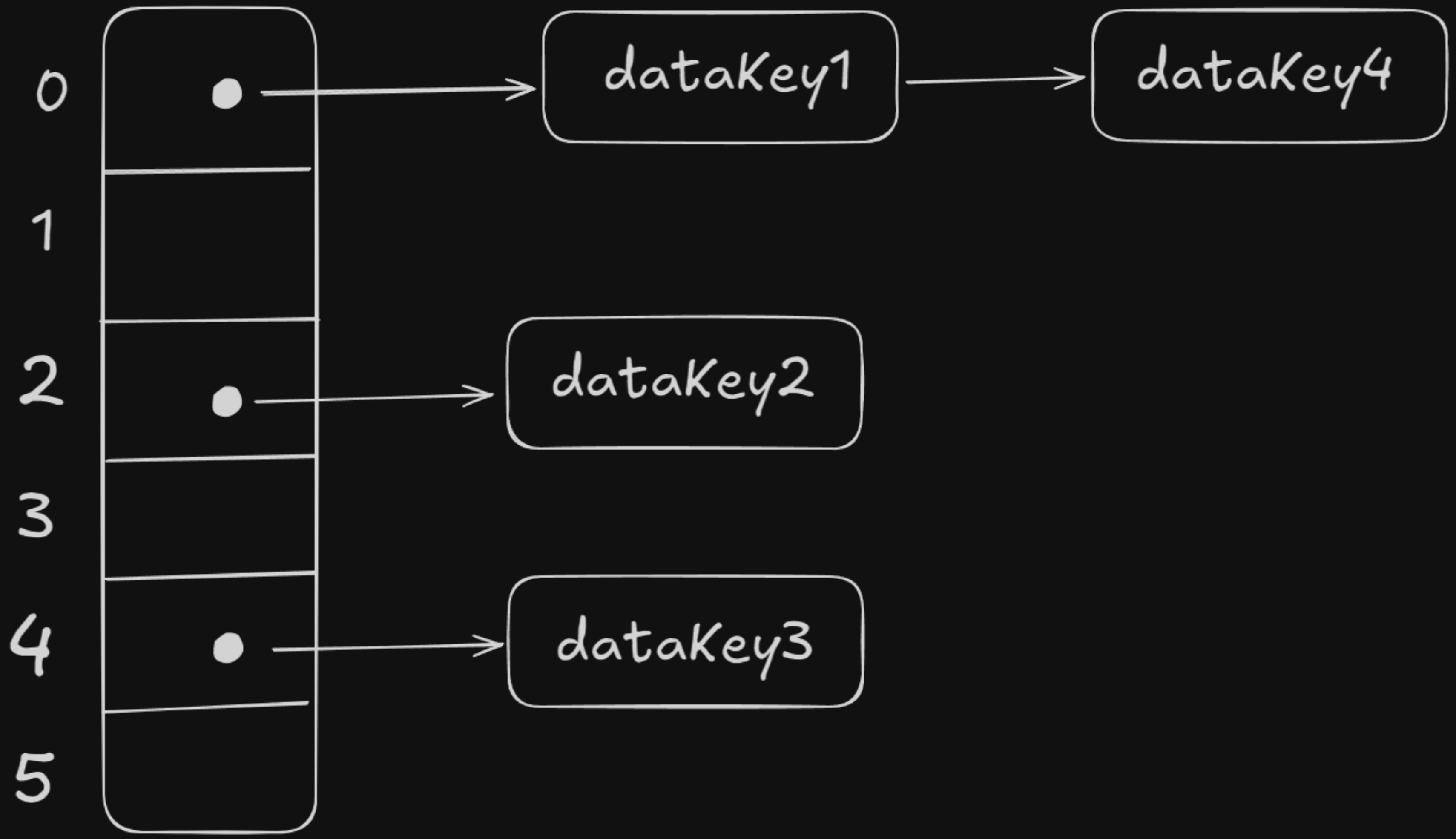
Separate Chaining

- Memory: pointers per entry increase footprint
- Cache: poor locality because of pointer chasing
- Use when: bucket sizes remain small or deletions frequent

Open Addressing

- Memory: compact contiguous array lowers overhead
- Cache: better locality and fewer cache misses at moderate load
- Use when: read-heavy workloads and load factor controlled

Chaining example



Code example

```c
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10

// Node structure for chaining
struct Node {
    int key;
    struct Node* next;
};

// Hash table (array of pointers)
struct Node* hashTable[TABLE_SIZE];

```

Code example

```c
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert key into hash table
void insert(int key) {
    int index = hashFunction(key);

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->next = hashTable[index];

    hashTable[index] = newNode;
}

// Search key in hash table
int search(int key) {
    int index = hashFunction(key);
    struct Node* current = hashTable[index];

    while (current != NULL) {
        if (current->key == key)
            return 1; // Found
        current = current->next;
    }
    return 0; // Not found
}
```

Collision

```c

// Main function
int main() {
    insert(15);
    insert(25);
    insert(35); // collision with 15 and 25

    if (search(25))
        printf("Key 25 found in hash table.\n");
    else
        printf("Key 25 not found.\n");

    return 0;
}
```

Thanks for Listening — Key Takeaways on Hashmaps

Questions about hashing or performance?