Hash Functions and Collision Resolution

Welcome to the lecture on hash functions and collision resolution methods! Today, we will explore one of the most crucial tools in a programmer's arsenal.

Hash Functions and Hash Tables

Separate Chaining Method for Collision Resolution

01

Understanding Hash Functions

We will study the principles of operation and purpose

02

Exploring Collisions

We will analyze problems and solutions

03

Separate Chaining Method

Practical implementation in C++

Lecture Topic

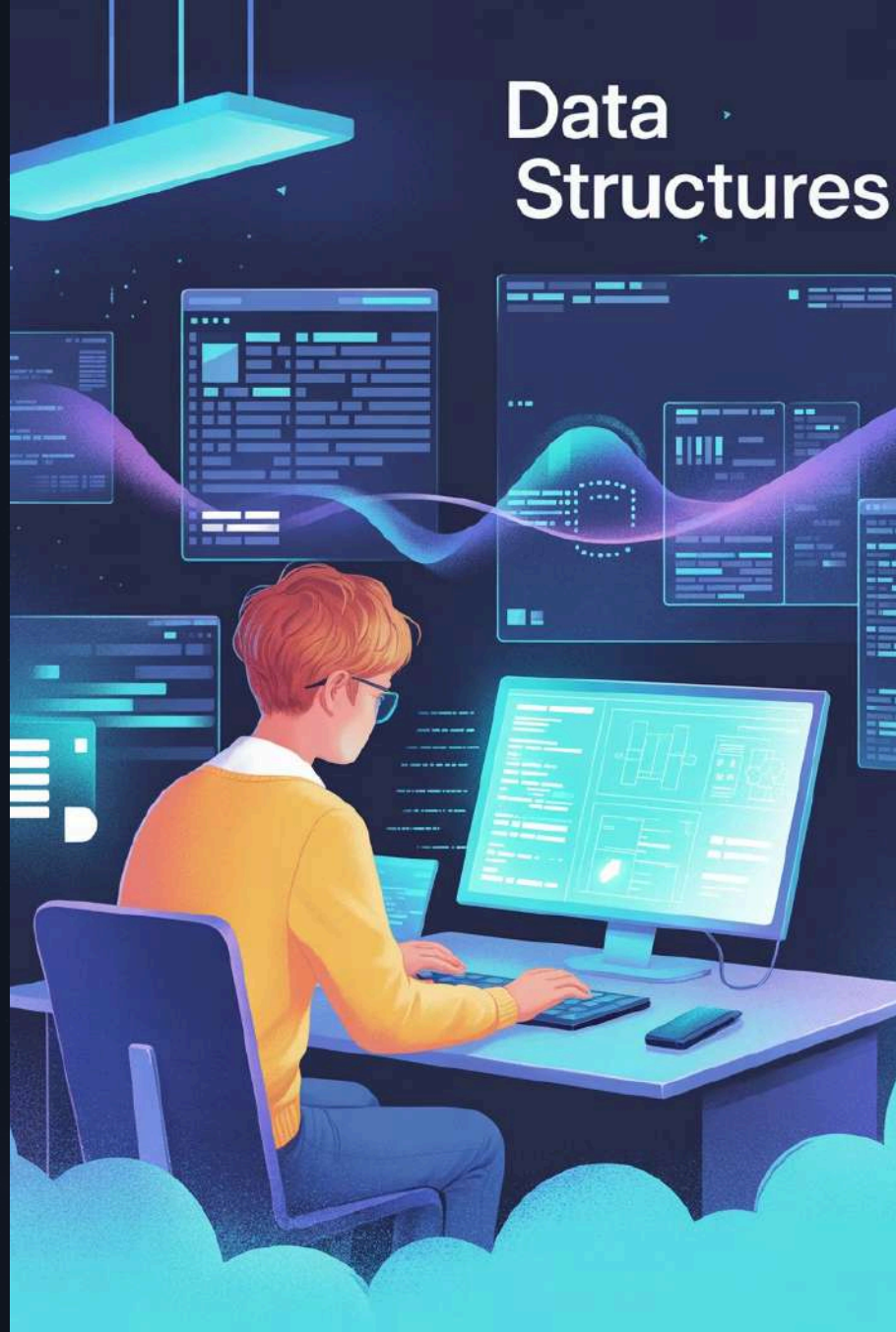
Hash functions and hash tables as the basis for efficient data retrieval

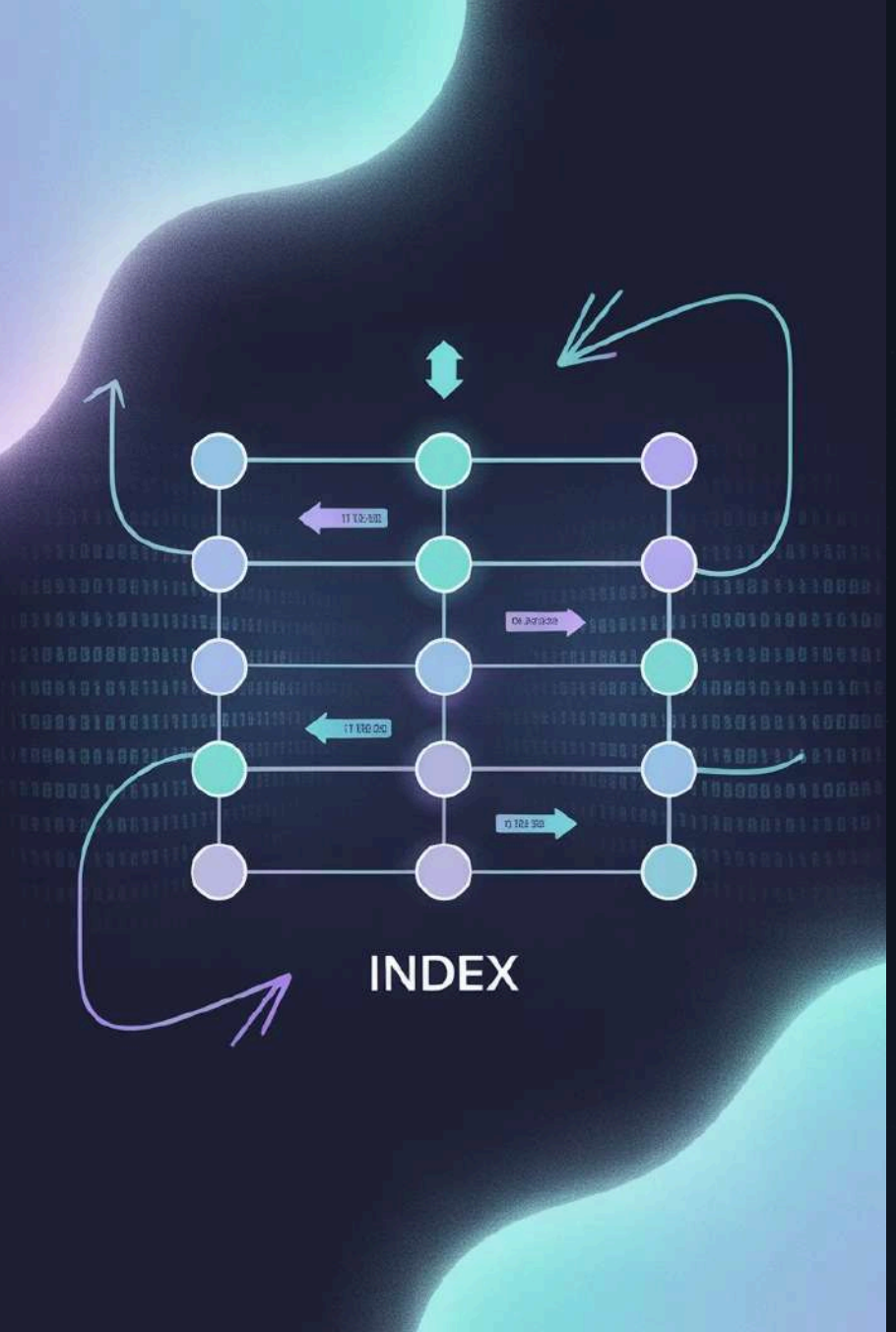
Sub-topic

Chaining method for collision resolution - an elegant solution to an inevitable problem

Learning Objective

Understand the structure of hash functions, the nature of collisions, and how to handle them





Part 1. Hash Functions

What is a Hash Function?

A hash function is a mathematical mapping that transforms a key of any size (string, number, object) into a **fixed-range array index**.

It's like a magical algorithm that takes arbitrary data and outputs a number that can be used as an address in a table.

$$h(key) \rightarrow [0 \dots m-1]$$

where **m** is the size of the hash table

Everyday examples

Mailboxes

At the post office, the recipient's surname determines the box number alphabetically. This is the simplest example of hashing!

Student Distribution

At university, students are assigned to classrooms based on the first letter of their surname – another example of a hash function.

Programming

std::map in C++ and dictionaries in Python use hashing for fast element lookup by key.

Requirements for a Good Hash Function

Computation Speed

The function should operate in O(1) time, otherwise, the whole point of search optimization is lost.

Uniform Distribution

Keys should be distributed as uniformly as possible across the table, avoiding clustering in individual cells.

Collision Minimization

A good function minimizes cases where different keys produce the same hash.

Examples of Simple Hash Functions:

- h(k) = k mod m for numbers
- For strings: sum of ASCII codes of characters mod m

🔹 Part 2. Hash Tables

What is a hash table?

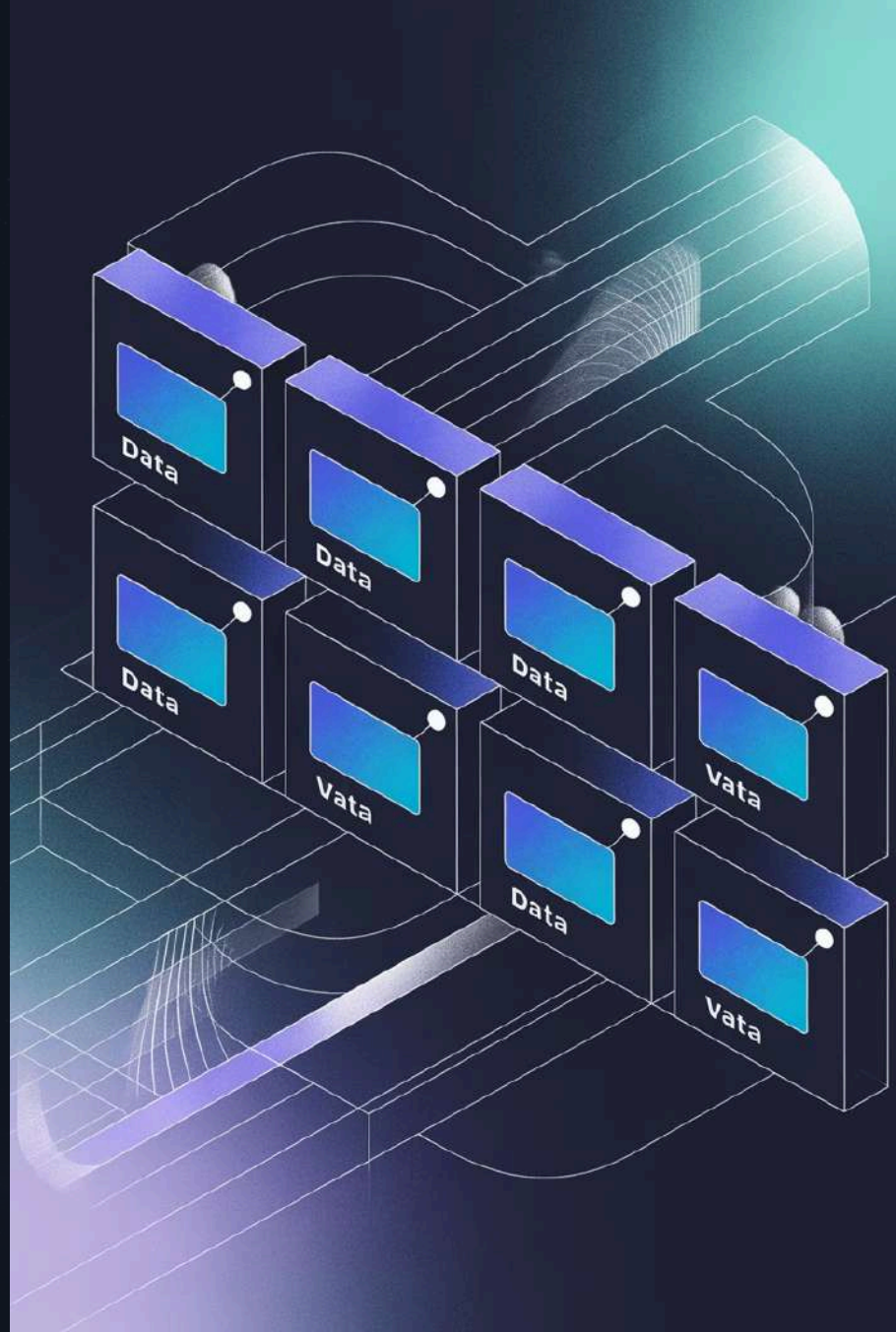
A hash table is a data structure that stores "key → value" pairs and provides fast access to data.

It's like a library catalog: by the book's title (key), we instantly find its location (value).

Key operations:

- `insert(key, value)` - insertion
- `search(key)` - search
- `delete(key)` - deletion

Average complexity: O(1)



Collisions

A collision occurs when two or more different keys produce the same hash code and claim the same table cell.

Inevitability

Collisions are inevitable according to the "birthday paradox" principle – they will definitely occur even with a good hash function.

Problem

We need to decide where to place the second element if its "ideal" spot is already taken.

Solutions

There are several methods for collision resolution, and we will explore one of the most popular.

Part 3. Chaining Method

Chaining Method

In the chaining method, each cell of the hash table contains not a single value, but a **linked list** of all elements that have the same hash.

Imagine a coat check: if several coats have the same hanger number, we simply hang them one after another on the same hanger.

1

Hashing

Calculate cell index

2

Find Chain

Locate the correct list

3

Operation

Add/search/delete in the list

Visualization of the Chaining Method

Consider a hash table of size m = 5 with a simple hash function h(k) = k mod 5:

```
Хэш-таблица (m = 5):
0: → [15] → [20]
1: → [6]
2: (пусто)
3: → [18]
4: → [9] → [14]
```

It is clear that elements 15 and 20 fell into the same cell (0), since 15 mod 5 = 0 and 20 mod 5 = 0. They form a chain in the zero cell.

Similarly, 9 and 14 form a chain in the fourth cell, as both yield a remainder of 4 when divided by 5.

C++ Implementation - Class Structure

```
#include
#include
#include
#include
using namespace std;

struct HashTable {
 int size;
 vector> table;

 HashTable(int s) : size(s), table(s) {}

 int hashFunction(int key) {
 return key % size;
 }

 void insert(int key) {
 int idx = hashFunction(key);
 table[idx].push_back(key);
 }

 bool search(int key) {
 int idx = hashFunction(key);
 for (int val : table[idx])
 if (val == key) return true;
 return false;
 }

 void remove(int key) {
 int idx = hashFunction(key);
 table[idx].remove(key);
 }

 void display() {
 for (int i = 0; i < size; i++) {
 cout << i << ": ";
 for (int val : table[i])
 cout << val << " -> ";
 cout << "NULL" << endl;
 }
 }
};
```

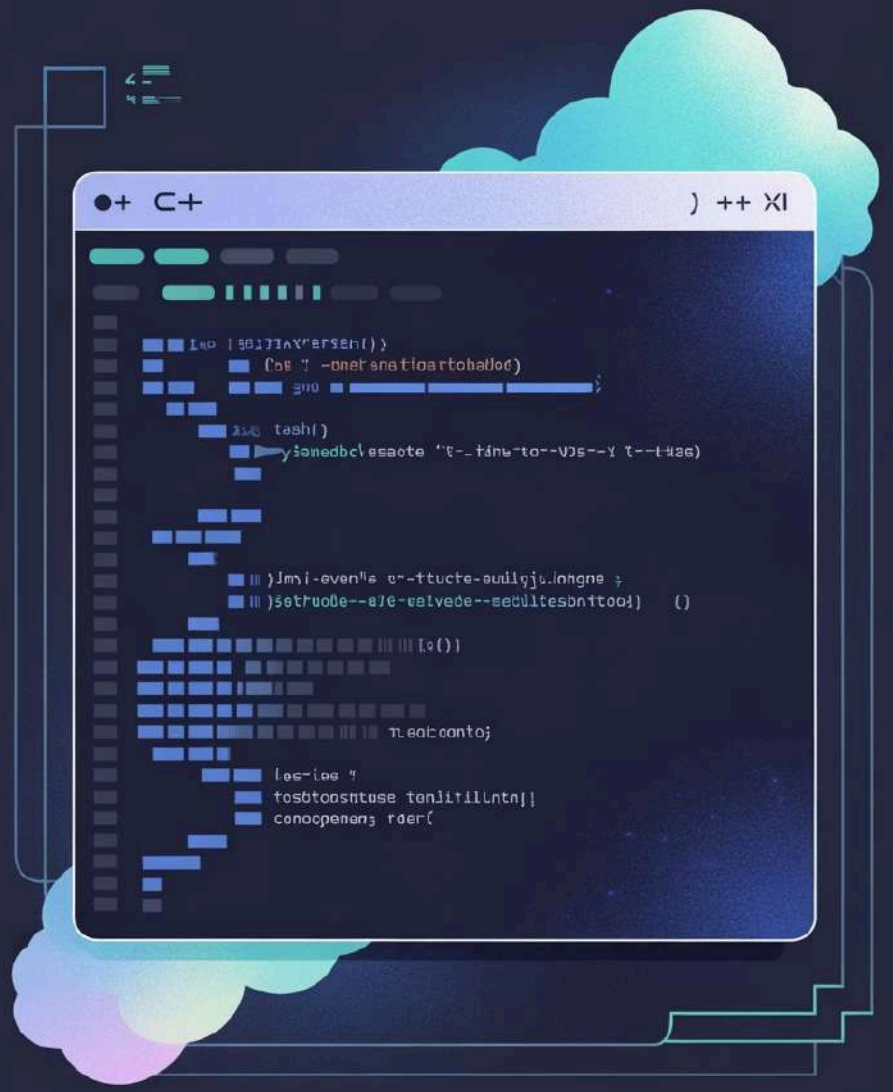
Hash Table Usage Example

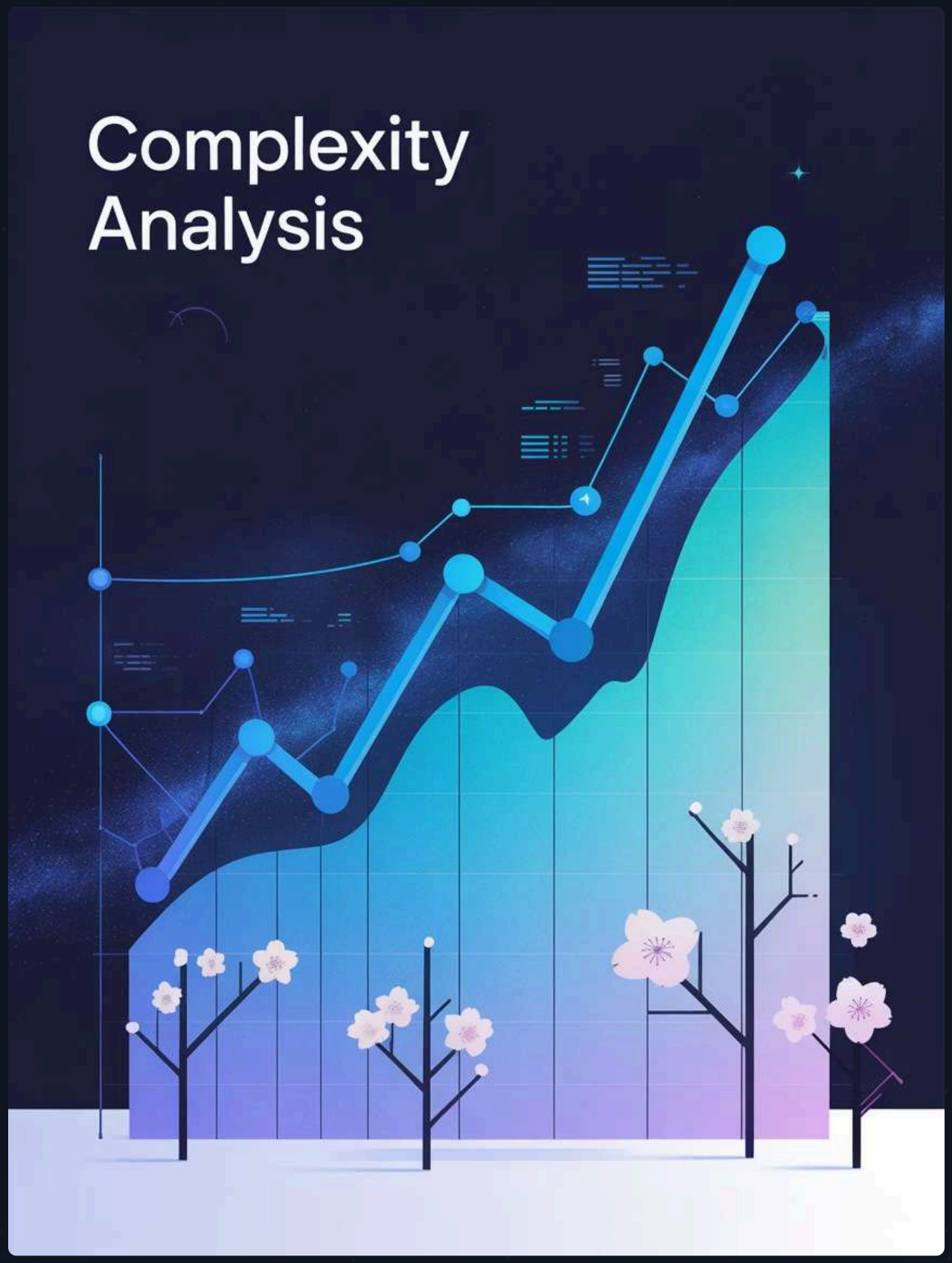
Program Code:

```
int main() {
    HashTable ht(5);

    ht.insert(15);
    ht.insert(20);
    ht.insert(9);
    ht.insert(14);

    ht.display();

    cout << (ht.search(20) ?
        "Found" : "Not found")
        << endl;

    ht.remove(20);
    ht.display();

    return 0;
}
```

Program Output:

```
0: 15 -> 20 -> NULL
1: NULL
2: NULL
3: NULL
4: 9 -> 14 -> NULL
Found
0: 15 -> NULL
1: NULL
2: NULL
3: NULL
4: 9 -> 14 -> NULL
```





Chaining Method Complexity Analysis

O(1+α)	O(n)	O(n+m)
Average Complexity	**Worst Case**	**Memory**
where α = load factor	when all elements fall into one chain	n elements + m table cells

The load factor α = n/m shows the average number of elements in one chain. With α ≤ 1, performance remains excellent.

Advantages and Disadvantages of the Chaining Method

✅ Advantages

- Simplicity of implementation and understanding
- Easily scalable with data growth
- Efficient memory utilization
- Supports any number of elements
- Deletion of elements does not require restructuring

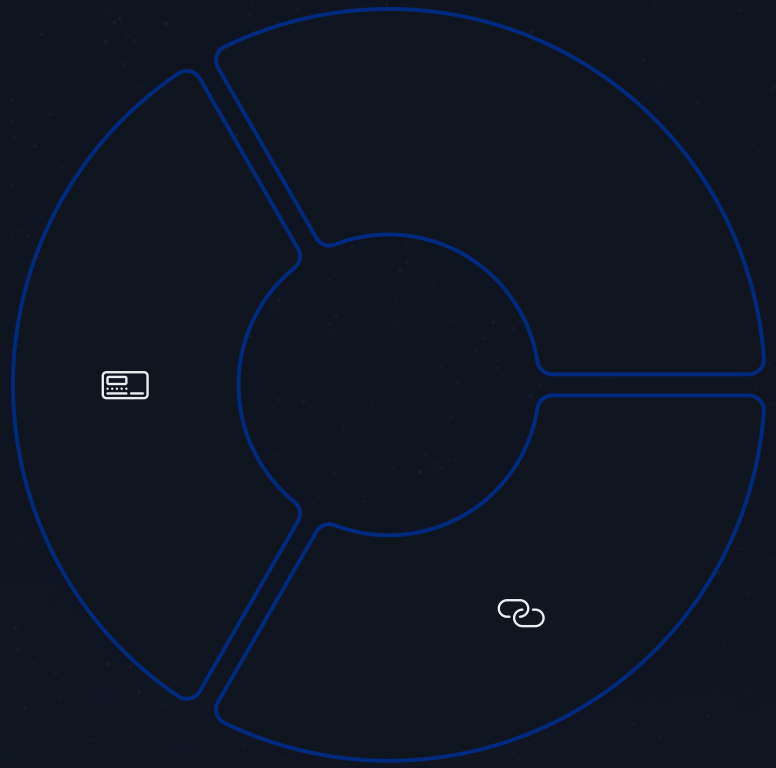
❌ Disadvantages

- Additional memory for list pointers
- Performance degradation with a high load factor
- Poor data locality in memory
- Possible degradation to O(n) in the worst case

Summary

Fast Search

Hash functions provide data retrieval in constant time



Collisions are Inevitable

But there are effective methods to resolve them

Chaining Method

One of the simplest and most popular approaches to collision resolution

Review Questions for Self-Assessment

1 Why are collisions inevitable?

Consider the "birthday problem" principle and the limited table size with an unlimited set of possible keys.

2 Differences from open addressing

Compare chaining with open addressing. What is the fundamental difference in approaches?

3 Load factor and performance

How does the load factor affect operation execution time? What is the optimal range?

Further Study

Now that you have mastered the basics of hash functions and the separate chaining method, are you ready for practical tasks? In the workshop, we will implement a full-fledged hash table and test it on various datasets.

</>

Practical Implementation

Creating your own hash table from scratch

Testing

Checking performance on different data

Optimization

Exploring other collision resolution methods

See you at the workshop! Prepare your questions and be ready to code. 🚀