Data Structures: Advanced Data Structures

Minseok Jeon
DGIST

November 2, 2025

Contents

Introduction

Advanced Data Structures

Definition

Advanced data structures are specialized structures optimized for performance and specific problem domains.

Knowledge Points

1. Hash tables and hashing
2. Collision handling: chaining/open addressing
3. Tries (prefix trees)
4. Disjoint Set / Union-Find
5. Segment Trees and Fenwick Trees
6. B-Trees and B+ Trees
7. Real-world applications

Hash Tables and Hashing

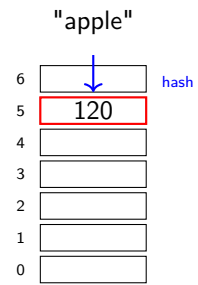
What is a Hash Table?

Definition

A **hash table** (hash map) provides O(1) average-case insertion, deletion, and lookup by mapping keys to values using a **hash function**.

Core Concept:

Key $\rightarrow$ Hash Function $\rightarrow$ Index $\rightarrow$ Value



Example

"apple" $\rightarrow$ hash("apple") $= 5 \rightarrow$ array[5] $=$ 120

Good Hash Function Properties

Essential Properties

1. **Deterministic**
   - Same key always produces same hash
2. **Uniform distribution**
   - Spreads keys evenly across table
3. **Fast to compute**
   - O(1) time
4. **Minimize collisions**
   - Different keys should have different hashes

Common Hash Functions

1. **Division Method:**

$$\text{h(key)} = \text{key \% m}$$

2. **Multiplication Method:**

$$\text{h(key)} = \lfloor m \times (key \times A \bmod 1) \rfloor$$

3. **String Hashing:**

Polynomial hash with prime base

Collisions and Load Factor

Problem: Collisions

When two different keys hash to the same index:

$$\text{hash("apple")} = 5$$
$$\text{hash("orange")} = 5 \quad \leftarrow \textbf{Collision!}$$

Load Factor

Ratio of filled slots to total slots:

$$\alpha = \frac{n}{m}$$

where n = number of elements
m = table size

Example: 7 elements in table of size 10
$\rightarrow$ Load factor = 0.7

When to Resize

- Typically resize when $\alpha > 0.7$
- Double the table size
- Rehash all existing elements
- Maintains O(1) operations

Hash Table Complexity

Operation	Average Case	Worst Case
Insert	O(1)	O(n)
Delete	O(1)	O(n)
Search	O(1)	O(n)
Space	O(n)	O(n)

Note

Worst case occurs when all keys collide into the same slot. With a good hash function and proper load factor management, average case O(1) is achieved.

Applications

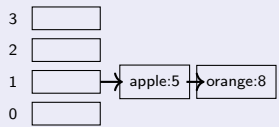
Databases (indexing), caching, symbol tables (compilers), sets, counting frequencies, detecting duplicates

Collision Handling

Two Main Collision Resolution Strategies

1. Chaining (Separate Chaining)

- Each slot contains a linked list
- Colliding elements form a chain
- Never fills up completely



2. Open Addressing

- All elements in table itself
- Probe for next empty slot
- Must maintain load factor < 1.0

Probing Methods:

- Linear: h(k) + i
- Quadratic: h(k) + i^2
- Double hashing: h_1(k) + i*h_2(k)

Chaining Implementation

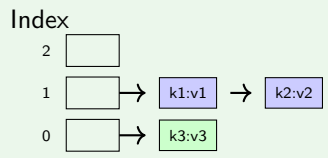
Advantages

- ✓ Simple to implement
- ✓ Never fills up
- ✓ Good for unknown size
- ✓ Deletion is easy

Disadvantages

- × Extra memory for pointers
- × Poor cache performance
- × Chains can become long

Visual Example



Open Addressing Strategies

Linear Probing

Try consecutive slots: h(k), h(k)+1, h(k)+2, ...
Problem: Primary clustering (keys cluster together)

Quadratic Probing

Try quadratic intervals: h(k), h(k)+1^2, h(k)+2^2, h(k)+3^2, ...
Reduces primary clustering but can cause secondary clustering

Double Hashing

Use second hash function: h(k, i) = (h_1(k) + i * h_2(k)) % m
Best collision resolution for open addressing

Important

Must use tombstone markers for deletions to maintain probe sequences

Chaining vs Open Addressing

Feature	Chaining	Open Addressing
Memory	Extra (pointers)	More compact
Cache performance	Poor	Better
Load factor	Can exceed 1.0	Must stay < 1.0
Deletion	Easy	Complex (tombstones)
Implementation	Simple	More complex
Best for	Unknown size	Known size, high performance

Real-World

Python's dict uses open addressing with pseudo-random probing (optimized version of double hashing)

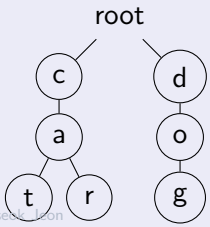
Tries (Prefix Trees)

What is a Trie?

Definition

A **trie** (pronounced "try") is a tree-like structure for storing strings where each node represents a character. Excellent for prefix-based operations.

Storing: "cat", "car", "dog"



Key Properties

- Each path = a word
- Common prefixes share paths
- Fast prefix lookups
- Space-efficient for common prefixes

Operations

- Insert: O(m)
- Search: O(m)

Trie Operations

Insert Word

1. Start at root
2. For each character:
   - Create node if needed
   - Move to child node
3. Mark end of word

Search Word

1. Start at root
2. Follow characters
3. Check end-of-word marker

Autocomplete

1. Navigate to prefix end
2. DFS from that node
3. Collect all words in subtree

Example

Prefix: "app"
Results:
- app
- apple
- application
- apply

Trie Applications

1. Autocomplete Systems

- Google search suggestions
- IDE code completion
- Command-line completion

2. Spell Checking

- Microsoft Word
- Browser spell check
- Find words within edit distance

3. IP Routing

- Longest prefix matching
- Network routing tables

4. Phone Contacts

- T9 predictive text
- Contact search

Trie vs Hash Table

Trie: O(m) exact search, O(m) prefix search, sorted order
Hash Table: O(1) exact search, O(n) prefix search, no order

Disjoint Set / Union-Find

Disjoint Set (Union-Find)

Definition

A data structure that efficiently handles partitioning of elements into disjoint (non-overlapping) sets.

Key Operations

- **Find**: Which set does element belong to?
- **Union**: Merge two sets into one
- **Connected**: Are two elements in same set?

Use Cases

- Connected components in graphs
- Kruskal's MST algorithm
- Network connectivity
- Image processing (connected regions)
- Equivalence relations

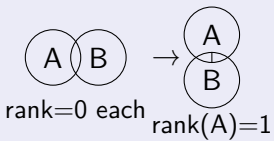
Example

Initial: {0}, {1}, {2}, {3}, {4}
union(0, 1): {0, 1}, {2}, {3}, {4}

Optimizations

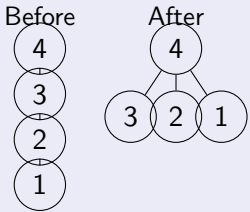
1. Union by Rank

- Track height of each tree
- Attach smaller tree under larger tree
- Keeps trees balanced
- Improves to O(log n)



2. Path Compression

- During find, point all nodes to root
- Flattens tree structure
- Future finds become faster
- Achieves $O(\alpha(n)) \approx O(1)$



Union-Find Complexity

Implementation	Find	Union
Naive	O(n)	O(n)
Union by rank	O(log n)	O(log n)
Path compression	O(log n)	O(log n)
Both optimizations	**$O(\alpha(n)) \approx O(1)$**	**$O(\alpha(n)) \approx O(1)$**

$\alpha(n)$ - Inverse Ackermann Function

- Grows **extremely slowly**
- $\alpha(n) < 5$ for any practical n
- Essentially constant time in practice

Application: Kruskal's MST

Sort edges: O(E log E), Process edges with Union-Find: O(E * α(V)) $\approx$ O(E)

Segment Trees and Fenwick Trees

Range Query Problem

Problem

Given an array, efficiently answer queries like:

- Sum of elements in range [L, R]
- Minimum/maximum in range [L, R]
- Update single element

Naive Approach

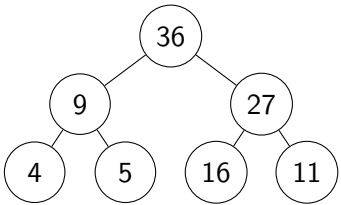
- Range query: O(n) - iterate through range
- Update: O(1)
- Too slow for multiple queries!

Goal

Both operations in O(log n) time

Segment Tree Structure

Array: [1, 3, 5, 7, 9, 11]



Each node = interval sum

Key Ideas

- Binary tree structure
- Each node represents an interval
- Leaf = single array element
- Internal node = sum/min/max of children
- Height = O(log n)

Operations

- Build: O(n)
- Query range: O(log n)
- Update element: O(log n)

Fenwick Tree (Binary Indexed Tree)

Advantages

- More space-efficient than segment tree
- O(n) space vs O(4n) for segment tree
- Simpler implementation
- Efficient for prefix sums

Limitations

- Only works for prefix sums
- Cannot do min/max queries
- No range updates
- Less versatile than segment tree

Key Operations

- Update: Add delta to element
- Prefix sum: Sum from 0 to index
- Range sum: prefix(R) - prefix(L-1)

Bit Trick

Uses bit manipulation for tree navigation:

- Next: index + (index & -index)
- Parent: index - (index & -index)

Segment Tree vs Fenwick Tree

Feature	Segment Tree	Fenwick Tree
Space	O(4n)	O(n)
Build	O(n)	O(n log n)
Query	O(log n)	O(log n)
Update	O(log n)	O(log n)
Range update	Yes (lazy prop)	No
Versatility	High	Limited
Implementation	Complex	Simple

When to Use

Segment Tree: Need min/max/GCD queries, range updates, multiple query types
Fenwick Tree: Only prefix sums, simpler implementation, memory constrained

B-Trees and B+ Trees

Why B-Trees?

Problem with BSTs for Disk Storage

- Disk access is **very slow** (milliseconds vs nanoseconds for RAM)
- Want to minimize disk reads
- BSTs read one node at a time
- Need to read large blocks of data at once

B-Tree Solution

- Each node can have **multiple keys** (not just one)
- Each node has **multiple children**
- All leaves at same level (perfectly balanced)
- Optimized for disk I/O
- One node = one disk block

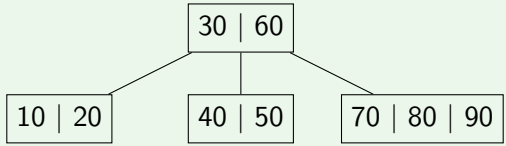
B-Tree Structure

Properties

Given minimum degree $t \geq 2$:

- Each node has at least t-1 keys (except root)
- Each node has at most 2t-1 keys
- Each internal node has at least t children
- Each internal node has at most 2t children
- All leaves at same depth

Example B-Tree (t=3)



Each node = 1 disk block

B+ Trees

Key Differences

1. **All data in leaves**
   - Internal nodes only store keys
2. **Leaves are linked**
   - Sequential access
3. **Better for range queries**
   - Traverse linked leaves

Advantages

- All leaves at same level
- Efficient range queries
- Internal nodes have more keys
- Less tree height
- Used in most databases

Used In

- MySQL (InnoDB)
- PostgreSQL
- MongoDB
- File systems (HFS+, NTFS)

B-Tree Operations and Complexity

Operations

Search:

- Find position in node (binary search within node)
- Follow child pointer if needed
- O(log n) disk reads

Insert:

- If node is full, split it
- Promote middle key to parent
- Recursively split if needed
- O(log n) disk reads/writes

Database Index Example

CREATE INDEX idx_name ON users(name);
Internally creates B+ tree where:

Real-World Applications

Hash Tables in Practice

Database Systems

- Hash indexes for fast lookups
- Join operations
- Duplicate detection

Caching Systems

- Redis
- Memcached
- Browser caches
- DNS caches

Programming Languages

- Python: dict
- Java: HashMap
- JavaScript: Object, Map
- C++: unordered_map

Blockchain

- Bitcoin transaction lookup
- Merkle tree verification

Tries in Practice

Search Engines

- Google search suggestions
- Autocomplete
- Query correction

Networking

- IP routing tables
- Longest prefix matching
- DNS resolution

Text Editors

- Microsoft Word spell check
- VSCode IntelliSense
- Vim command completion

Mobile Devices

- T9 predictive text
- Contact search
- App name search

Union-Find in Practice

Network Analysis

- Social network components
- Computer network connectivity
- Circuit connectivity

Graph Algorithms

- Kruskal's MST
- Cycle detection
- Dynamic connectivity

Image Processing

- Connected region detection
- Image segmentation
- Blob detection

Other Applications

- Compiler variable equivalence
- Percolation (physics)
- Maze generation

B-Trees in Practice

Database Systems

- MySQL InnoDB storage engine
- PostgreSQL primary indexes
- MongoDB index structure
- SQLite database indexing
- Oracle Database

File Systems

- HFS+ (macOS)
- NTFS (Windows)
- ext4 directory indexing
- ReiserFS

Why B-Trees Win

1. Minimize disk I/O
2. Read large blocks at once
3. Excellent for range queries
4. Maintain sorted order
5. Automatic balancing

Performance

For 1 million records with B-tree of degree 100:

- Height ≈ 3
- Search $= 3$ disk reads
- BST might need 20 reads!

Summary

Key Takeaways - Part 1

Hash Tables

- O(1) average-case operations
- Chaining vs Open Addressing
- Load factor management crucial
- Powers dictionaries, caches, databases

Tries

- Efficient prefix operations: O(m)
- Autocomplete, spell check, IP routing
- Space-efficient for common prefixes
- Trade memory for prefix speed

Key Takeaways - Part 2

Union-Find

- Manages disjoint sets efficiently
- With optimizations: $O(\alpha(n)) \approx O(1)$
- Kruskal's MST, connectivity, image processing
- Union by rank + path compression essential

Segment Trees & Fenwick Trees

- Efficient range queries: O(log n)
- Segment tree: Versatile, complex
- Fenwick tree: Simple, limited to sums
- Applications: Range queries, inversions

Key Takeaways - Part 3

B-Trees & B+ Trees

- Optimized for disk storage
- Minimize disk I/O: O(log n) reads
- B+ Trees: Better for range queries
- Foundation of modern databases

Choosing the Right Structure

- Consider access patterns
- Memory vs speed trade-offs
- Exact lookups $\rightarrow$ Hash table
- Prefix operations $\rightarrow$ Trie
- Range queries $\rightarrow$ Segment/Fenwick tree
- Disk storage $\rightarrow$ B-tree

Complexity Comparison

Data Structure	Key Operation	Time	Best For
Hash Table	Insert/Search/Delete	O(1) avg	Exact lookups
Trie	Prefix search	O(m)	Prefix operations
Union-Find	Union/Find	$O(\alpha(n))$	Connectivity
Segment Tree	Range query	O(log n)	Range queries
Fenwick Tree	Prefix sum	O(log n)	Sum queries
B-Tree	Disk search	O(log n)	Disk storage

Remember

No single data structure is best for everything. Choose based on your specific requirements and constraints.

Thank You!

Questions?