Next Steps

Explore Advanced and Specialized Topics to Continue Growing

Minseok Jeon
DGIST

November 2, 2025

Outline

1. Introduction
2. Parallel & Distributed Data Structures
   - 2.1 Concurrent Data Structures
   - 2.2 Distributed Data Structures
   - 2.3 Parallel Algorithms
3. Data Structures in Machine Learning
   - 3.1 Tensors
   - 3.2 Graph Neural Networks
   - 3.3 Embeddings and Search
4. Persistent & Functional Data Structures
5. External Memory & Cache-Oblivious Structures
   - 5.1 B-Trees for External Memory
   - 5.2 LSM Trees
   - 5.3 Cache-Oblivious Algorithms
6. Reading Research Papers
7. Summary

Introduction

Course Journey Recap

What You've Learned:

- **Foundations**: Arrays, linked lists, stacks, queues, hash tables
- **Trees & Graphs**: Binary trees, BSTs, heaps, graph algorithms
- **Advanced Structures**: Tries, B-trees, union-find, segment trees
- **Algorithms**: Sorting, searching, graph traversal, dynamic programming
- **Applications**: Real-world projects and interview preparation

You're Ready!

You now have a solid foundation in data structures and algorithms. This lecture explores where to go next.

What's Next?

Five Advanced Directions:

1. **Parallel & Distributed Data Structures**
   - Multi-threaded and distributed computing
   - Concurrent data structures, consistent hashing, CRDTs
2. **Data Structures in Machine Learning**
   - Tensors, graph neural networks, embeddings
   - Specialized structures for ML workflows
3. **Persistent & Functional Data Structures**
   - Immutable structures, version control
   - Used in functional programming languages
4. **External Memory & Cache-Oblivious Structures**
   - Optimizing for disk I/O and memory hierarchy
   - B-trees, LSM trees, cache-aware algorithms
5. **Reading Research Papers & Implementations**
   - Learning cutting-edge techniques
   - Implementing from academic papers

Parallel & Distributed Data Structures

Why Parallel & Distributed?

Motivation:

- Modern CPUs have multiple cores
- Big data requires distributed systems
- Single-threaded = leaving performance on table
- Cloud computing is inherently distributed

Challenges:

- Race conditions
- Deadlocks
- Data consistency
- Network failures

Examples:

- Multi-threaded web servers
- Distributed databases (Cassandra, MongoDB)
- MapReduce frameworks (Hadoop, Spark)
- P2P networks (BitTorrent, blockchain)

Goal:

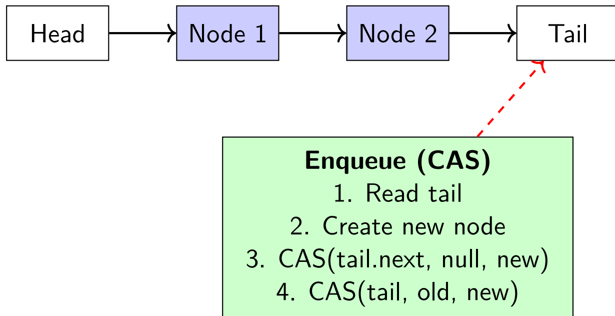
- Safe concurrency
- Scalability across machines
- Fault tolerance

Concurrent Data Structures

Thread-Safe Implementations:

- **Lock-Free Structures**
  - Use atomic operations (CAS - Compare-And-Swap)
  - No threads block waiting for locks
  - Example: Lock-free queue, lock-free stack
  - Complexity: Higher implementation complexity, better throughput
- **Concurrent Hash Maps**
  - Java's `ConcurrentHashMap`: Lock striping (segment-level locks)
  - C++ concurrent containers: `std::concurrent_*`
  - Fine-grained locking for better concurrency
- **Read-Write Locks**
  - Multiple readers, single writer
  - Reader-writer problem solution
  - Use case: Read-heavy workloads (caches, databases)
- **Producer-Consumer Queues**

Lock-Free Queue Example



Key Idea:

- Atomic CAS ensures no two threads modify same location simultaneously
- Retry on CAS failure (optimistic concurrency)
- No locks → no deadlocks, better scalability

Distributed Hash Tables (DHT)

Concept:

- Hash table across multiple machines
- Decentralized, no single point of failure
- Each node responsible for key range

Algorithms:

- **Chord**: Ring topology, $O(\log n)$ lookup
- **Kademlia**: XOR metric, used in BitTorrent
- **Consistent Hashing**: Load balancing

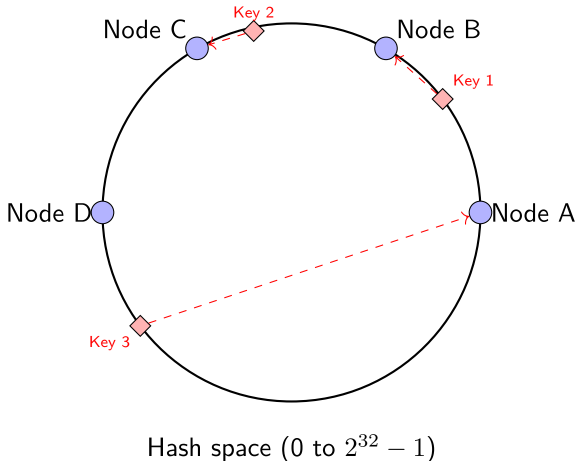
Consistent Hashing:

- Map keys and nodes to ring
- Key stored on next node clockwise
- Adding/removing nodes affects only neighbors
- Used in: Memcached, Redis Cluster, Cassandra

Benefits:

- Minimal key redistribution on node changes
- Fault tolerance
- Horizontal scalability

Consistent Hashing Visualization



Key Assignment: Each key assigned to first node clockwise on ring

CRDTs: Conflict-Free Replicated Data Types

Concept:

- Designed for eventual consistency
- No coordination needed
- Merging replicas is commutative & associative
- Guaranteed convergence

Types:

- **G-Counter**: Grow-only counter
- **PN-Counter**: Increment/decrement counter
- **G-Set**: Grow-only set
- **OR-Set**: Observed-Remove set
- **LWW-Register**: Last-Write-Wins register

Example: G-Counter

- Each replica has array of counts (one per node)
- Increment updates local count
- Merge takes element-wise max
- Value = sum of all counts

Applications:

- Collaborative editing (Google Docs)
- Distributed databases (Riak, Redis)
- Real-time synchronization
- Mobile apps with offline support

Parallel Algorithms Overview

Parallel Sorting:

- **Parallel Merge Sort**: Divide array, sort in parallel, merge
- **Parallel Quick Sort**: Partition in parallel, recurse
- Fork-Join framework (Java, C++)
- Speedup: Near-linear with number of cores

Parallel Prefix Sum:

- Tree-based reduction
- Applications: Stream compaction, radix sort
- GPU-friendly (CUDA, OpenCL)

MapReduce Paradigm:

- **Map**: Process data in parallel
- **Reduce**: Aggregate results
- Frameworks: Hadoop, Spark

Example: Word Count

1. Map: (doc $\rightarrow$ (word, 1) pairs)
2. Shuffle: Group by word
3. Reduce: Sum counts per word

Synchronization:

- Barriers for phase sync
- Atomic operations
- Memory fences

Data Structures in Machine Learning

ML Data Structures Landscape

Why Specialized Structures?

- Machine learning operates on high-dimensional data
- Computational efficiency is critical (training time, inference latency)
- Memory optimization for large models and datasets
- Specialized hardware (GPUs, TPUs) requires specific layouts

Key Structures:

1. **Tensors**: Multi-dimensional arrays for neural networks
2. **Graph Structures**: For graph neural networks
3. **Embeddings**: Nearest neighbor search in high dimensions
4. **Batch Processing**: Efficient data loading and preprocessing

Tensors: Multi-Dimensional Arrays

Definition:

- Generalization of vectors and matrices
- 0D: Scalar
- 1D: Vector
- 2D: Matrix
- 3D+: Tensor

Example: Image Tensor

- Shape: (batch, channels, height, width)
- (32, 3, 224, 224) = 32 RGB images of 224x224
- Memory: $32 \times 3 \times 224 \times 224 \times 4$ bytes ≈ 19 MB

Storage Formats:

- **Row-major (C-style)**: Last index varies fastest
- **Column-major (Fortran)**: First index varies fastest
- Affects cache performance

Operations:

- Broadcasting: Automatic dimension matching
- Reshaping: Change dimensions without copying
- Slicing: Extract subtensors
- Element-wise ops: Add, multiply, etc.

Sparse Tensors

Motivation:

- Many tensors are sparse (mostly zeros)
- Example: Embeddings, text data, graphs
- Dense storage wasteful

COO Format (Coordinate):

- Store: (indices, values)
- Example: (0,2): 5, (1,1): 3, (2,0): 7
- Good for: Construction, random access

CSR Format (Compressed Sparse Row):

- Store: row_ptr, col_indices, values
- More memory efficient than COO
- Fast row access
- Used in: Matrix multiplication, graph algorithms

Example: 3x3 Sparse Matrix

$$\begin{bmatrix} 0 & 0 & 5 \\ 0 & 3 & 0 \\ 7 & 0 & 0 \end{bmatrix}$$

CSR:
row_ptr = [0, 1, 2, 3]
col_indices = [2, 1, 0]

GNN Graph Representations

Adjacency Matrix:

- Dense $O(V^2)$ representation
- Fast edge queries: $O(1)$
- Good for: Dense graphs, small graphs
- Matrix multiplication for message passing

Adjacency List:

- Sparse representation
- Space: $O(V + E)$
- Good for: Sparse graphs (most real-world)
- Iteration over neighbors: $O(\text{degree})$

Edge List:

- Simple (source, target, weight) tuples
- Easy to store and load
- Requires sorting for efficient queries

Libraries:

- **NetworkX**: Python graph library
- **PyTorch Geometric**: GNN framework
- **DGL**: Deep Graph Library
- **GraphSAGE, GCN**: Message passing

Applications:

- Social networks
- Molecular structures
- Recommendation systems

Embedding Structures

Embedding Tables:

- Lookup table for categorical features
- Maps discrete IDs to dense vectors
- Example: Word embeddings (word $\rightarrow$ 300D vector)
- Learned during training

Nearest Neighbor Search:

- Find similar embeddings
- Exact: Brute force $O(n)$
- Approximate: Much faster, slight accuracy loss

Search Structures:

- **KD-Trees**: $O(\log n)$ for low dimensions $(d < 20)$
- **Ball Trees**: Better for higher dimensions
- **HNSW**: Hierarchical Navigable Small World
  - Graph-based approximate search
  - Very fast: $O(\log n)$ queries
- **FAISS**: Facebook's similarity search library
  - GPU acceleration
  - Billions of vectors

Use Cases:

- Recommendation engines

Persistent & Functional Data Structures

Persistence: Preserving History

Definition

Persistent data structures preserve previous versions after modifications

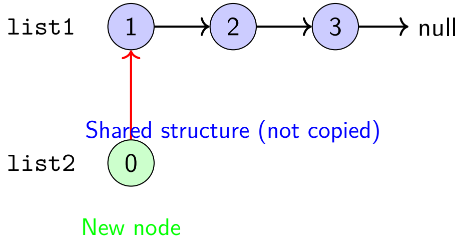
Types of Persistence:

- **Ephemeral**: Standard mutable structures (old version destroyed)
- **Partially Persistent**: Access all versions, modify only latest
- **Fully Persistent**: Access and modify any version
- **Confluently Persistent**: Merge different versions

Benefits:

- Undo/redo functionality
- Version control systems
- Concurrent programming without locks
- Functional programming paradigm

Persistent Lists: Structural Sharing

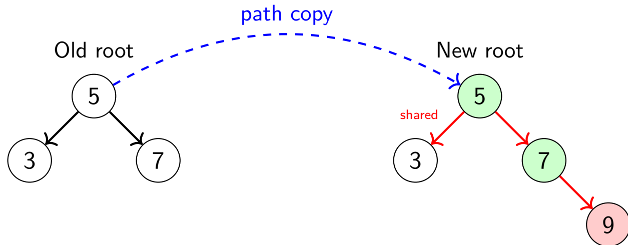


Key Idea:

- `list1 = [1, 2, 3]`
- `list2 = cons(0, list1) = [0, 1, 2, 3]`
- Both lists coexist, sharing nodes [1, 2, 3]
- Time: $O(1)$ for cons, Space: $O(1)$ per operation

Persistent Trees: Path Copying

Updating a Red-Black Tree:



Path Copying:

- Copy only nodes on path from root to modified node
- Other subtrees shared (not copied)
- Time: $O(\log n)$ per operation, Space: $O(\log n)$ per version
- Old version still accessible through old root

Persistent Hash Maps: HAMT

Hash Array Mapped Trie (HAMT):

- Used in Clojure, Scala
- 32-way branching tree
- Hash bits determine path
- Efficient updates with sharing

Structure:

- Root has 32 children (5 bits of hash)
- Each level consumes 5 bits
- Depth: $\lceil \log_{32} n \rceil$

Persistent Vectors:

- Clojure's persistent vector
- 32-way branching tree
- $O(\log_{32} n) \approx O(1)$ for practical sizes
- Efficient append, update, lookup

Complexity:

- Lookup: $O(\log_{32} n)$
- Update: $O(\log_{32} n)$
- Append: $O(\log_{32} n)$
- For $n < 32^6 \approx 10^9$: ≤ 6 operations

Applications

Functional programming languages, concurrent systems, version control

External Memory & Cache-Oblivious Structures

Memory Hierarchy Reality

Memory Hierarchy:

- L1 Cache: 64 KB, 1 ns
- L2 Cache: 256 KB, 4 ns
- L3 Cache: 8 MB, 10 ns
- RAM: 16 GB, 100 ns
- SSD: 512 GB, 100 μs
- HDD: 2 TB, 10 ms

Gap:

- RAM is 100x slower than L1
- Disk is 100,000x slower than RAM
- I/O is the bottleneck

External Memory Model:

- Data stored on disk
- Limited RAM (cache)
- Transfer data in blocks
- Goal: Minimize I/O operations

Cost Model:

- Count block reads/writes
- Ignore in-memory computation
- Block size: B elements
- Memory size: M elements

Why Care?

- Big data doesn't fit in RAM

B-Trees: Disk-Optimized Trees

Design for Disk:

- High branching factor (100-1000)
- Each node = one disk block
- Shallow tree (minimize I/Os)
- All leaves at same level

Example:

- Branching factor $B = 100$
- Height: $\log_{100} n$
- For $n = 1$ billion: height ≈ 5
- 5 disk reads for any query!

Operations:

- Search: $O(\log_B n)$ I/Os
- Insert: $O(\log_B n)$ I/Os
- Delete: $O(\log_B n)$ I/Os
- Range query: $O(\log_B n + k/B)$ I/Os

B+ Trees:

- All data in leaves
- Internal nodes only keys
- Leaves linked (range queries)
- Used in: Databases (MySQL, PostgreSQL)

File Systems:

- ext4, NTFS, Btrfs use B-trees

LSM Trees: Write-Optimized Structures

Log-Structured Merge Trees:

- Optimized for write-heavy workloads
- Used in: LevelDB, RocksDB, Cassandra, HBase
- Trade read performance for write throughput

Structure:

- **Memtable**: In-memory sorted map
- **SSTables**: Immutable sorted files on disk
- Levels of increasing size (10x each level)

Operations:

- **Write**: Insert into memtable ($O(\log n)$)
- When full: Flush memtable to SSTable
- **Read**: Check memtable, then SSTables (bloom filter helps)
- **Compaction**: Merge SSTables periodically

Benefits:

- Fast writes (sequential I/O)
- Good compression
- No fragmentation

Drawbacks:

- Slower reads

Cache-Oblivious Algorithms

Definition

Algorithms optimal for all levels of memory hierarchy without knowing cache parameters (B, M)

Key Idea:

- Recursive divide-and-conquer
- Eventually fits in cache
- Automatically adapts to any cache size

Examples:

- **Funnelsort**: Cache-oblivious sorting
  - $O(N/B \log_{M/B} N/B)$ I/Os (optimal)
  - Recursive merging with "funnels"
- **Van Emde Boas Layout**: Tree layout for cache efficiency
  - Recursive decomposition
  - Better cache performance than level-order

Reading Research Papers

Why Read Papers?

Benefits:

- Learn cutting-edge techniques before they're in textbooks
- Understand the "why" behind algorithms, not just "how"
- Develop critical thinking and research skills
- Stay current with field developments
- Prepare for graduate studies or research careers

Where to Find Papers:

- **Conferences**: STOC, FOCS, SODA (theory), SIGMOD, VLDB (databases)
- **Archives**: arXiv.org (cs.DS category), Google Scholar
- **Digital Libraries**: ACM Digital Library, IEEE Xplore
- **Surveys**: Start with survey papers for broad overviews

Three-Pass Reading Strategy

Pass 1: Quick Scan (5-10 min)

- Read: Title, abstract, introduction, conclusion
- Goal: Understand problem, why important, main contribution
- Decide: Is this paper relevant to me?

Pass 2: Careful Read (1 hour)

- Read entire paper, skip proofs
- Focus on: Algorithm design, key insights, complexity analysis
- Look at figures and examples carefully
- Note: Questions, unclear parts, novel ideas

Pass 3: Deep Dive (4-5 hours)

- Re-implement algorithm from scratch
- Verify proofs, work through examples
- Challenge assumptions, think of improvements
- Compare with related work

Key Sections to Focus On

Abstract:

- Problem statement
- Proposed solution
- Main results (complexity, bounds)

Introduction:

- Motivation (why this problem matters)
- Related work (what's been done)
- Contributions (what's new)

Preliminaries:

- Definitions and notation
- Problem model
- Background concepts

Main Algorithm:

- Pseudocode
- Invariants and correctness
- Key insights

Analysis:

- Time/space complexity
- Lower bounds
- Optimality arguments

Experiments:

- Performance comparisons
- Real-world validation
- Limitations

Implementation Resources

Finding Implementations:

- **GitHub**: Search for paper title or algorithm name
- **Papers with Code**: Links papers to code
- **Author websites**: Often provide reference implementations

Libraries:

- **cp-algorithms.com**: Competitive programming algorithms
- **KACTL**: KTH's algorithm template library
- **Boost**: C++ algorithm and data structure library

Visualization:

- **VisuAlgo**: Interactive algorithm visualizations
- **Algorithm Visualizer**: Step-by-step animations
- **Draw diagrams**: Understanding improves with visualization

Practice Projects:

- Implement a classic paper (Skip Lists, Bloom Filters)
- Reproduce experimental results
- Compare with standard library
- Write explanatory blog post

Staying Current

Following the Field:

- **arXiv RSS**: Subscribe to cs.DS (Data Structures) category
- **Social Media**: Follow researchers on Twitter/Mastodon
- **Blogs**: Read technical blogs (Terry Tao, Shtetl-Optimized, etc.)
- **Conferences**: Attend talks (many recorded and posted online)

Community Engagement:

- **Reading groups**: Join or start a paper reading group
- **Study circles**: Discuss papers with peers
- **Online forums**: Stack Overflow, CS Theory StackExchange
- **Reddit**: r/compsci, r/algorithms

Building Intuition:

- Ask: "Why does this work? What's the key insight?"
- Draw diagrams and work through examples
- Identify the invariant or potential function
- Connect new structures to ones you already know

Summary

Summary: Next Steps

Five Directions to Explore:

1. **Parallel & Distributed Data Structures**
   - Concurrent structures, lock-free algorithms, CRDTs
   - Distributed hash tables, consistent hashing
2. **Machine Learning Data Structures**
   - Tensors, sparse formats, graph neural networks
   - Embeddings, nearest neighbor search (HNSW, FAISS)
3. **Persistent & Functional Structures**
   - Structural sharing, path copying, HAMT
   - Applications in functional programming and version control
4. **External Memory & Cache-Oblivious**
   - B-trees, LSM trees for disk optimization
   - Cache-oblivious algorithms for memory hierarchy
5. **Research Papers & Implementations**
   - Three-pass reading strategy

Recommended Learning Path

Immediate Next Steps (1-3 months):

- Pick ONE direction that interests you most
- Read 2-3 introductory papers or textbook chapters
- Implement 1-2 basic structures from that area
- Build a small project demonstrating the concept

Medium Term (3-6 months):

- Dive deeper: Read 5-10 papers in chosen area
- Implement more complex structures
- Contribute to open-source projects
- Write blog posts or tutorials explaining what you learned

Long Term (6-12 months):

- Explore second or third direction
- Work on research project or thesis
- Attend conferences (virtual or in-person)
- Consider graduate studies if interested in research

Recommended Resources

Textbooks:

- "Purely Functional Data Structures" (Okasaki)
- "The Art of Multiprocessor Programming" (Herlihy & Shavit)
- "Algorithms and Data Structures for External Memory" (Vitter)

Online Courses:

- MIT 6.851: Advanced Data Structures
- Stanford CS166: Data Structures
- Coursera: Machine Learning specializations

Websites:

- arXiv.org (cs.DS)
- Papers with Code
- cp-algorithms.com
- VisuAlgo.net

Communities:

- CS Theory StackExchange
- r/algorithms, r/compsci
- Local university reading groups

Final Thoughts

You've Come Far!

From basic arrays to advanced algorithms, you've built a strong foundation in data structures and algorithms.

Key Takeaways:

- Data structures are everywhere in modern computing
- Specialization matters: Different domains need different structures
- Learning is continuous: New structures and algorithms constantly developed
- Implementation deepens understanding: Don't just read, code!
- Community helps: Learn from others, share your knowledge

Parting Advice:

- Follow your curiosity
- Build projects you care about
- Don't be intimidated by research papers
- Ask "why" constantly

Where to Apply Your Knowledge

Career Paths:

- **Software Engineering**: Backend systems, databases, search engines
- **Research**: Academic or industrial research labs
- **Data Science/ML**: Building scalable ML systems
- **Systems Programming**: Operating systems, compilers, databases
- **Competitive Programming**: Contests, algorithm challenges

Real-World Impact:

- Design Google's search infrastructure
- Build Facebook's graph database
- Optimize Netflix's recommendation engine
- Develop high-frequency trading systems
- Create next-generation databases

The Future is Yours

Data structures and algorithms are the foundation. What you build on top is up to you!

Thank You!

Best of luck in your journey!

Questions?

"The best way to predict the future is to invent it." – *Alan Kay*

Now go forth and build amazing things with data structures!

Stay curious, keep learning, and never stop exploring!