

contents

<i>foreword</i>	<i>xv</i>
<i>preface</i>	<i>xvii</i>
<i>acknowledgments</i>	<i>xix</i>
<i>about this book</i>	<i>xxi</i>
<i>about the author</i>	<i>xxvi</i>
<i>about the cover illustration</i>	<i>xxvii</i>

1 *Introducing data structures* **1**

1.1 *Data structures* **2**

<i>Defining a data structure</i>	<i>2</i>	▪	<i>Describing a data structure</i>	<i>5</i>
<i>Algorithms and data structures: Is there a difference?</i> <i>5</i>				

1.2 *Setting goals: Your expectations after reading this book* **6**

1.3 *Packing your knapsack: Data structures meet the real world* **7**

<i>Abstracting the problem away</i>	<i>7</i>	▪	<i>Looking for solutions</i>	<i>8</i>			
<i>Algorithms to the rescue</i>	<i>9</i>	▪	<i>Thinking (literally) outside of the box</i>	<i>10</i>	▪	<i>Happy ending</i>	<i>11</i>

PART 1 *IMPROVING OVER BASIC DATA STRUCTURES***13**

2 *Improving priority queues: d-way heaps* **15**

2.1 *Structure of this chapter* **16**

2.2	The problem: Handling priority	17
	<i>Priority in practice: Bug tracking</i>	17
2.3	Solutions at hand: Keeping a sorted list	19
	<i>From sorted lists to priority queues</i>	19
2.4	Describing the data structure API: Priority queues	19
	<i>Priority queue at work</i>	21
	<i>Priority matters: Generalize FIFO</i>	22
2.5	Concrete data structures	22
	<i>Comparing performance</i>	23
	<i>What's the right concrete data structure?</i>	23
	<i>Heap</i>	24
	<i>Priority, min-heap, and max-heap</i>	26
	<i>Advanced variant: d-ary heap</i>	27
2.6	How to implement a heap	28
	<i>BubbleUp</i>	29
	<i>PushDown</i>	33
	<i>Insert</i>	35
	<i>Top</i>	37
	<i>Update</i>	40
	<i>Dealing with duplicates</i>	41
	<i>Heapify</i>	42
	<i>Beyond API methods: Contains</i>	44
	<i>Performance recap</i>	45
	<i>From pseudo-code to implementation</i>	46
2.7	Use case: Find the k largest elements	47
	<i>The right data structure . . .</i>	47
	<i>. . . and the right use</i>	48
	<i>Coding it up</i>	48
2.8	More use cases	49
	<i>Minimum distance in graphs: Dijkstra</i>	49
	<i>More graphs: Prim's algorithm</i>	49
	<i>Data compression: Huffman codes</i>	50
2.9	Analysis of branching factor	55
	<i>Do we need d-ary heaps?</i>	55
	<i>Running time</i>	56
	<i>Finding the optimal branching factor</i>	56
	<i>Branching factor vs memory</i>	57
2.10	Performance analysis: Finding the best branching factor	58
	<i>Please welcome profiling</i>	59
	<i>Interpreting results</i>	61
	<i>The mystery with heapify</i>	66
	<i>Choosing the best branching factor</i>	67
3	Treaps: Using randomization to balance binary search trees	69
3.1	Problem: Multi-indexing	70
	<i>The gist of the solution</i>	71
3.2	Solution: Description and API	71
3.3	Treap	72
	<i>Rotations</i>	75
	<i>A few design questions</i>	80
	<i>Implementing search</i>	81
	<i>Insert</i>	82
	<i>Delete</i>	87
	<i>Top, peek, and update</i>	89
	<i>Min, max</i>	90
	<i>Performance recap</i>	91

- 3.4 Applications: Randomized treaps 91
 - Balanced trees* 92 ▪ *Introducing randomization* 94
 - Applications of Randomized Treaps* 96
- 3.5 Performance analysis and profiling 97
 - Theory: Expected height* 97 ▪ *Profiling height* 100 ▪ *Profiling running time* 104 ▪ *Profiling memory usage* 106
 - Conclusions* 107

4 **Bloom filters: Reducing the memory for tracking content** 110

- 4.1 The dictionary problem: Keeping track of things 111
- 4.2 Alternatives to implementing a dictionary 113
- 4.3 Describing the data structure API: Associative array 114
- 4.4 Concrete data structures 115
 - Unsorted array: Fast insertion, slow search* 115 ▪ *Sorted arrays and binary search: Slow insertion, fast(-ish) search* 116 ▪ *Hash table: Constant-time on average, unless you need ordering* 117
 - Binary search tree: Every operation is logarithmic* 118 ▪ *Bloom filter: As fast as hash tables, but saves memory (with a catch)* 119
- 4.5 Under the hood: How do Bloom filters work? 120
- 4.6 Implementation 122
 - Using a Bloom filter* 122 ▪ *Reading and writing bits* 124
 - Find where a key is stored* 126 ▪ *Generating hash functions* 126
 - Constructor* 127 ▪ *Checking a key* 128 ▪ *Storing a key* 130
 - Estimating accuracy* 132
- 4.7 Applications 132
 - Cache* 132 ▪ *Routers* 133 ▪ *Crawler* 134 ▪ *IO fetcher* 134
 - Spell checker* 134 ▪ *Distributed databases and file systems* 135
- 4.8 Why Bloom filters work 135
 - Why there are no false negatives . . .* 136 ▪ *. . . But there are false positives* 137 ▪ *Bloom filters as randomized algorithms* 137
- 4.9 Performance analysis 138
 - Running time* 138 ▪ *Constructor* 138 ▪ *Storing an element* 138 ▪ *Looking up an element* 139
- 4.10 Estimating Bloom filter precision 139
 - Explanation of the false-positive ratio formula* 141
- 4.11 Improved variants 144
 - Bloomier filter* 144 ▪ *Combining Bloom filters* 144 ▪ *Layered Bloom filter* 144 ▪ *Compressed Bloom filter* 145 ▪ *Scalable Bloom filter* 146

5 *Disjoint sets: Sub-linear time processing* 147

- 5.1 The distinct subsets problem 148
- 5.2 Reasoning on solutions 151
- 5.3 Describing the data structure API: Disjoint set 153
- 5.4 Naïve solution 154
 - Implementing naïve solution* 155
- 5.5 Using a tree-like structure 159
 - From list to trees* 159 ▪ *Implementing the tree version* 160
- 5.6 Heuristics to improve the running time 162
 - Path compression* 164 ▪ *Implementing balancing and path compression* 166
- 5.7 Applications 168
 - Graphs: Connected components* 169 ▪ *Graphs: Kruskal's algorithm for minimum spanning tree* 169 ▪ *Clustering* 170
 - Unification* 171

6 *Trie, radix trie: Efficient string search* 173

- 6.1 Spell-check 174
 - A princess, a Damon, and an elf walkz into a bar* 175
 - Compression is the key* 176 ▪ *Description and API* 177
- 6.2 Trie 177
 - Why is it better again?* 180 ▪ *Search* 183 ▪ *Insert* 187
 - Remove* 189 ▪ *Longest prefix* 192 ▪ *Keys matching a prefix* 193 ▪ *When should we use tries?* 195
- 6.3 Radix tries 197
 - Nodes and edges* 199 ▪ *Search* 202 ▪ *Insert* 204
 - Remove* 207 ▪ *Longest common prefix* 209 ▪ *Keys starting with a prefix* 209
- 6.4 Applications 211
 - Spell-checker* 211 ▪ *String similarity* 213 ▪ *String sorting* 214
 - T9* 215 ▪ *Autocomplete* 215

7 *Use case: LRU cache* 218

- 7.1 Don't compute things twice 219
- 7.2 First attempt: Remembering values 222
 - Description and API* 224 ▪ *Fresh data, please* 224 ▪ *Handling asynchronous calls* 226 ▪ *Marking cache values as "Loading"* 227

- 7.3 Memory is not enough (literally) 228
- 7.4 Getting rid of stale data: LRU cache 230
 - Sometimes you have to double down on problems* 231 ▪ *Temporal ordering* 232 ▪ *Performance* 238
- 7.5 When fresher data is more valuable: LFU 238
 - So how do we choose?* 240 ▪ *What makes LFU different* 240
 - Performance* 243 ▪ *Problems with LFU* 243
- 7.6 How to use cache is just as important 244
- 7.7 Introducing synchronization 245
 - Solving concurrency (in Java)* 248 ▪ *Introducing locks* 249
 - Acquiring a lock* 250 ▪ *Reentrant locks* 252 ▪ *Read locks* 252
 - Other approaches to concurrency* 253
- 7.8 Cache applications 254

PART 2 MULTIDEMENSIONAL QUERIES257

8 *Nearest neighbors search* 259

- 8.1 The nearest neighbors search problem 260
- 8.2 Solutions 261
 - First attempts* 261 ▪ *Sometimes caching is not the answer* 262
 - Simplifying things to get a hint* 262 ▪ *Carefully choose a data structure* 264
- 8.3 Description and API 266
- 8.4 Moving to k-dimensional spaces 268
 - Unidimensional binary search* 268 ▪ *Moving to higher dimensions* 269 ▪ *Modeling 2-D partitions with a data structure* 270

9 *K-d trees: Multidimensional data indexing* 273

- 9.1 Right where we left off 274
- 9.2 Moving to k-D spaces: Cycle through dimensions 275
 - Constructing the BST* 276 ▪ *Invariants* 281 ▪ *The importance of being balanced* 282
- 9.3 Methods 282
 - Search* 284 ▪ *Insert* 287 ▪ *Balanced tree* 289
 - Remove* 293 ▪ *Nearest neighbor* 301 ▪ *Region search* 310
 - A recap of all methods* 316
- 9.4 Limits and possible improvements 316

10 **Similarity Search Trees: Approximate nearest neighbors search for image retrieval** 319

- 10.1 Right where we left off 320
 - A new (more complex) example* 321 ▪ *Overcoming k-d trees' flaws* 322
- 10.2 R-tree 322
 - A step back: Introducing B-trees* 323 ▪ *From B-Tree to R-tree* 324
 - Inserting points in an R-tree* 326 ▪ *Search* 328
- 10.3 Similarity search tree 330
 - SS-tree search* 333 ▪ *Insert* 337 ▪ *Insertion: Variance, means, and projections* 345 ▪ *Insertion: Split nodes* 348 ▪ *Delete* 352
- 10.4 Similarity Search 359
 - Nearest neighbor search* 359 ▪ *Region search* 363
 - Approximated similarity search* 364
- 10.5 SS+tree 367
 - Are SS-trees better?* 367 ▪ *Mitigating hyper-sphere limitations* 369
 - Improved split heuristic* 370 ▪ *Reducing overlap* 371

11 **Applications of nearest neighbor search** 375

- 11.1 An application: Find nearest hub 376
 - Sketching a solution* 377 ▪ *Trouble in paradise* 379
- 11.2 Centralized application 380
 - Filtering points* 381 ▪ *Complex decisions* 383
- 11.3 Moving to a distributed application 386
 - Issues handling HTTP communication* 387 ▪ *Keeping the inventory in sync* 389 ▪ *Lessons learned* 390
- 11.4 Other applications 391
 - Color reduction* 391 ▪ *Particle interaction* 393 ▪ *Multidimensional DB queries optimization* 395 ▪ *Clustering* 398

12 **Clustering** 400

- 12.1 Intro to clustering 401
 - Types of learning* 402 ▪ *Types of clustering* 403
- 12.2 K-means 405
 - Issues with k-means* 410 ▪ *The curse of dimensionality strikes again* 412 ▪ *K-means performance analysis* 413 ▪ *Boosting k-means with k-d trees* 414 ▪ *Final remarks on k-means* 418

12.3 DBSCAN 418

Directly vs density-reachable 419 ▪ *From definitions to an algorithm* 420 ▪ *And finally, an implementation* 422
Pros and cons of DBSCAN 424

12.4 OPTICS 426

Definitions 427 ▪ *OPTICS algorithm* 428 ▪ *From reachability distance to clustering* 433 ▪ *Hierarchical clustering* 436
Performance analysis and final considerations 441

12.5 Evaluating clustering results: Evaluation metrics 442

Interpreting the results 446

13 **Parallel clustering: MapReduce and canopy clustering** 448

13.1 Parallelization 449

Parallel vs distributed 450 ▪ *Parallelizing k-means* 451
Canopy clustering 452 ▪ *Applying canopy clustering* 454

13.2 MapReduce 455

Imagine you are Donald Duck . . . 455 ▪ *First map, then reduce* 459 ▪ *There is more under the hood* 462

13.3 MapReduce k-means 463

Parallelizing canopy clustering 467 ▪ *Centroid initialization with canopy clustering* 469 ▪ *MapReduce canopy clustering* 471

13.4 MapReduce DBSCAN 475

PART 3 **PLANAR GRAPHS AND MINIMUM CROSSING NUMBER....**483

14 **An introduction to graphs: Finding paths of minimum distance** 485

14.1 Definitions 486

Implementing graphs 487 ▪ *Graphs as algebraic types* 489
Pseudo-code 490

14.2 Graph properties 491

Undirected 492 ▪ *Connected* 492 ▪ *Acyclic* 494

14.3 Graph traversal: BFS and DFS 495

Optimizing delivery routes 495 ▪ *Breadth first search* 497
Reconstructing the path to target 500 ▪ *Depth first search* 502
It's queue vs stack again 505 ▪ *Best route to deliver a parcel* 506

14.4 Shortest path in weighted graphs: Dijkstra 507

Differences with BFS 508 ▪ *Implementation* 509
Analysis 510 ▪ *Shortest route for deliveries* 511

14.5 Beyond Dijkstra's algorithm: A* 513

How good is A search?* 517 ▪ *Heuristics as a way to balance real-time data* 520

15 **Graph embeddings and planarity: Drawing graphs with minimal edge intersections** 522

15.1 Graph embeddings 523

Some basic definitions 525 ▪ *Complete and bipartite graphs* 526

15.2 Planar graphs 528

Using Kuratowski's theorem in practice 529 ▪ *Planarity testing* 530 ▪ *A naïve algorithm for planarity testing* 531
Improving performance 536 ▪ *Efficient algorithms* 538

15.3 Non-planar graphs 539

Finding the crossing number 541 ▪ *Rectilinear crossing number* 543

15.4 Edge intersections 545

Straight-line segments 545 ▪ *Polylines* 549 ▪ *Bézier curves* 550 ▪ *Intersections between quadratic Bézier curves* 552
Vertex-vertex and edge-vertex intersections 555

16 **Gradient descent: Optimization problems (not just) on graphs** 558

16.1 Heuristics for the crossing number 560

Did you just say heuristics? 560 ▪ *Extending to curve-line edges* 566

16.2 How optimization works 568

Cost functions 568 ▪ *Step functions and local minima* 571
Optimizing random sampling 571

16.3 Gradient descent 574

The math of gradient descent 575 ▪ *Geometrical interpretation* 576 ▪ *When is gradient descent applicable?* 579
Problems with gradient descent 579

16.4 Applications of gradient descent 581

An example: Linear regression 583

16.5 Gradient descent for graph embedding 585

A different criterion 586 ▪ *Implementation* 588

17 *Simulated annealing: Optimization beyond local minima* 591

17.1 Simulated annealing 593

Sometimes you need to climb up to get to the bottom 595

Implementation 597 ▪ *Why simulated annealing works* 598

Short-range vs long-range transitions 601 ▪ *Variants* 602

Simulated annealing vs gradient descent: Which one should I use? 603

17.2 Simulated annealing + traveling salesman 604

Exact vs approximated solutions 606 ▪ *Visualizing cost* 607

Pruning the domain 608 ▪ *State transitions* 609 ▪ *Adjacent vs*

random swaps 613 ▪ *Applications of TSP* 614

17.3 Simulated annealing and graph embedding 615

Minimum edge crossing 615 ▪ *Force-directed drawing* 618

18 *Genetic algorithms: Biologically inspired, fast-converging optimization* 624

18.1 Genetic algorithms 625

Inspired by nature 627 ▪ *Chromosomes* 631 ▪ *Population* 633

Fitness 634 ▪ *Natural selection* 635 ▪ *Selecting individuals for*

mating 639 ▪ *Crossover* 645 ▪ *Mutations* 648 ▪ *The genetic*

algorithm template 650 ▪ *When does the genetic algorithm work best?* 651

18.2 TSP 652

Fitness, chromosomes, and initialization 653 ▪ *Mutations* 653

Crossover 654 ▪ *Results and parameters tuning* 656 ▪ *Beyond*

TSP: Optimizing the routes of the whole fleet 660

18.3 Minimum vertex cover 661

Applications of vertex cover 662 ▪ *Implementing a genetic algorithm* 663

18.4 Other applications of the genetic algorithm 665

Maximum flow 665 ▪ *Protein folding* 667 ▪ *Beyond genetic*

algorithms 668 ▪ *Algorithms, beyond this book* 669

appendix A *A quick guide to pseudo-code* 671

appendix B *Big-O notation* 682

appendix C *Core data structures* 690

appendix D *Containers as priority queues* 704

appendix E *Recursion* 708

appendix F *Classification problems and randomized algorithm metrics* 716

index 723