

CS240: Data Structures and Data Management

Therese Biedl

with the help of many fellow cs240 instructors

This collection of notes has been slowly growing from a “let’s fill a few gaps where the textbooks don’t cover the material well” to a full-fledged textbook on data structures and data management. It is more-or-less in lockstep with the CS240 slides, though occasionally the order may be a bit different. It includes lots of material that has been covered in some offering of an enriched section, though not all this material could be covered in one offering.

Material for these notes was taken from many sources, but I would like to especially acknowledge the textbooks by Knuth (The Art of Computer Programming vol. 3, 1973), Lewis and Denenberg (Data Structures and Their Algorithms, 1997), Sedgewick (Algorithms in C++, 1998), Cormen, Leiserson, Rivest and Stein (Introduction to Algorithms, 2009), de Berg, Cheong, van Kreveld and Overmars (Computational Geometry: Algorithms and Applications, 2010), Morin (Open Data Structures: An Introduction, 2013) and Goodrich and Tamassia (Algorithm Design and Applications, 2014). Many of my fellow instructors at UWaterloo have contributed to the course over the years, via the slides, careful proofreading of the book, or generally inspiring discussions. Many thanks to all, but especially to (in alphabetical order) Karen Anderson, Jérémy Barbay, Sajed Haque, Kevin Lanctot, Alejandro López-Ortiz, Ian Munro, Matthew Nichols, Mark Petrick, Éric Schost, Arne Storjohann, Olga Veksler, Tomáš Vinař, and Sebastian Wild.

Undoubtedly there are many errors in these notes. Please email me at biedl@uwaterloo.ca if you spot such errors; indicating the page number, the line number and the date of the version. Or send me an annotated PDF if you find many errors at once.

Enjoy!

Therese Biedl
Waterloo, Ontario, Canada
2022-03-16

Warning

Students should be aware that this lecture notes contains the intellectual property of the author, other cs240 instructors, and/or the University of Waterloo.

Sharing intellectual property without the intellectual property owner’s permission is a violation of intellectual property rights. For this reason, it is necessary to ask the author and/or the University of Waterloo for permission before uploading and sharing these notes online (e.g., to an online repository). Doing so without expressed permission is considered a violation of intellectual property rights.

Please alert biedl@uwaterloo.ca if you become aware of the notes circulating, either through the student body or online. The intellectual property rights owner deserves to know (and may have already given their consent).

Contents

1	Introduction	3
1.1	Motivation	4
1.2	Definitions and assumptions	6
1.2.1	Problems	6
1.2.2	Algorithms	7
1.2.3	Computer model	9
1.2.4	Algorithm analysis	10
1.3	Asymptotic notation	13
1.3.1	Tight asymptotic bounds	16
1.3.2	Growth rates	18
1.3.3	Friends of O and Ω	19
1.3.4	Rules for asymptotic notation	21
1.4	Algorithm analysis revisited	28
1.4.1	Step 1. Describe the algorithm idea	28
1.4.2	Step 2. Give a detailed description	29
1.4.3	Step 3. Argue correctness	29
1.4.4	Step 4. Analyze the efficiency	30
1.4.5	Comparing algorithms	32
1.4.6	Analysis of recursive algorithms	32
1.5	Run-time analysis in special situations	38
1.5.1	Average-case analysis	38
1.5.2	Randomized algorithms	45
1.5.3	Amortized analysis and potential functions (cs240e)	48
1.6	Take-home messages	52
1.7	Historical remarks	54
2	Priority Queues	55
2.1	ADT Priority Queue	55
2.1.1	Applications of priority queues	58
2.1.2	Elementary implementations	58

2.2	Non-trivial PQ realization: Binary heaps	60
2.2.1	Binary Tree Review	61
2.2.2	Heap definition	63
2.2.3	Storing heaps	63
2.2.4	Heap operations	64
2.3	Heapsort	67
2.3.1	Making the algorithm in-place	67
2.3.2	Heapify	68
2.4	More PQ operations (cs240e)	71
2.4.1	Meldable heap	72
2.4.2	Binomial heaps	76
2.5	Take-home messages	80
2.6	Historical remarks	81
3	Sorting	83
3.1	The Selection problem	83
3.1.1	Partition	85
3.1.2	Algorithm <i>quick-select</i>	88
3.1.3	Average-case analysis and randomization	91
3.2	Algorithm <i>quick-sort</i>	97
3.2.1	Average-case analysis	99
3.2.2	Tips & Tricks for <i>quick-sort</i>	101
3.3	Lower Bound for Comparison-based Sorting	106
3.4	Sorting integers	109
3.4.1	Sorting by one digit	109
3.4.2	Sorting multi-digit numbers	111
3.5	Take-home messages	113
3.6	Historical notes	114
4	ADT Dictionary	115
4.1	ADT Dictionary	115
4.1.1	Implementations you have seen	116
4.1.2	Outlook	119
4.1.3	Lazy deletion (cs240e)	119
4.2	AVL trees	120
4.2.1	AVL-tree definition	120
4.2.2	Height of an AVL-tree	122
4.2.3	AVL tree operations	123
4.3	Other balanced binary search trees	131
4.3.1	Scapegoat trees (cs240e)	133
4.4	Take-home messages	138

4.5	Historical remarks	139
5	More on ADT Dictionary	141
5.1	Randomized implementations of ADT Dictionary	141
5.1.1	Treaps (cs240e)	144
5.1.2	Skip lists	147
5.2	Biased search requests	156
5.2.1	Optimal static order in an unsorted list/array	157
5.2.2	Optimal static binary search trees (cs240e)	158
5.2.3	Self-adjusting lists/arrays	158
5.2.4	Self-adjusting binary search trees (cs240e)	162
5.3	Take-home messages	168
5.4	Historical notes	168
6	Special-key Dictionaries	171
6.1	Searching among sorted numbers	171
6.1.1	Lower Bound for search	172
6.1.2	Making binary search optimal (cs240e)	173
6.1.3	Interpolation search	176
6.1.4	Improving and analyzing <i>interpolation-search</i> (cs240e)	179
6.2	Dictionary for Words	185
6.2.1	Preliminaries	185
6.2.2	Binary tries	187
6.2.3	Variations of binary tries	191
6.2.4	Compressed tries	195
6.2.5	Multi-way tries	197
6.3	Take-home messages	199
6.4	Historical notes	199
7	Hashing	201
7.1	Direct addressing	201
7.2	Hashing overview	202
7.3	Collisions	203
7.3.1	Hashing with chaining	204
7.3.2	Hashing with probe sequences	208
7.3.3	Cuckoo hashing	215
7.3.4	Hashing run-time summary	217
7.4	Choosing hash functions	218
7.4.1	Universal hashing	220
7.4.2	Hash functions for other data	225
7.5	Take-home messages	226

7.6	Historical remarks	227
8	Range searches	229
8.1	Introduction	230
8.1.1	Range searches	230
8.1.2	Higher-dimensional data	231
8.1.3	Two simple (but deficient) ideas	232
8.2	Quad trees	233
8.2.1	Dictionary operations	235
8.2.2	Quad-tree height	236
8.2.3	Range search in a quad-tree	238
8.2.4	Quad-tree extensions and discussion	239
8.3	kd-trees	241
8.3.1	Dictionary operations	242
8.3.2	Range search in a kd-trees	243
8.3.3	kd-trees in higher dimensions	247
8.4	Range trees	247
8.4.1	Dictionary operations	247
8.4.2	Range search in a binary search tree	249
8.4.3	Range search in a range-tree	252
8.4.4	Range trees in a higher dimension	254
8.5	3-sided range searches (cs240e)	254
8.5.1	Idea 1: Range trees, except use heaps.	254
8.5.2	Idea 2: Cartesian trees	256
8.5.3	Idea 3: Priority search tree	257
8.6	Take-home messages	258
8.7	Historical remarks	259
9	Pattern Matching	261
9.1	Preliminaries	262
9.1.1	Brute-force algorithm	263
9.1.2	Pre-processing	264
9.2	Karp-Rabin fingerprinting	265
9.3	Knuth-Morris-Pratt algorithm	271
9.3.1	Pattern matching with an NFA	271
9.3.2	Pattern matching with a deterministic finite automaton	272
9.3.3	The KMP-automaton	273
9.3.4	Knuth-Morris-Pratt pattern matching	274
9.3.5	Computing the failure array	277
9.3.6	Related finite automata (cs240e)	280
9.4	Boyer-Moore algorithm	282

9.4.1	The last-occurrence heuristic	282
9.4.2	Simplified Boyer-Moore	285
9.4.3	The good-suffix heuristic (cs240e)	286
9.5	Suffix trees and suffix arrays	290
9.5.1	Trie of suffixes	291
9.5.2	Suffix trees	291
9.5.3	Suffix arrays	293
9.6	Take-home messages	295
9.7	Historical remarks	296
10	Text Compression	299
10.0.1	Detour: Streams	301
10.1	Character-by-character encoding	301
10.1.1	Encoding and decoding	303
10.1.2	Huffman's algorithm to build a prefix-free trie	306
10.2	Multi-character encodings	313
10.2.1	Run-Length Encoding	314
10.2.2	Lempel-Ziv-Welch	318
10.3	bzip2	325
10.3.1	Huffman encoding	325
10.3.2	Modified run-length encoding	325
10.3.3	Move-to-front transform	327
10.3.4	Burrows-Wheeler Transform	328
10.3.5	bzip2 discussion	334
10.4	Take-home messages	334
10.5	Historical remarks	335
11	External Memory	337
11.1	Introduction	337
11.1.1	The external memory model	338
11.1.2	Stream-based algorithms	340
11.2	External sorting (cs240e)	341
11.2.1	d -way merge	342
11.2.2	d -way merge-sort	343
11.3	External dictionaries	345
11.3.1	The overall idea	346
11.3.2	2-4-trees	347
11.3.3	Red-black trees (cs240e)	355
11.3.4	a - b -trees	356
11.3.5	B -trees	361
11.3.6	B -tree variations (cs240e)	363

11.4	Extendible hashing	366
11.4.1	Tries of blocks	367
11.4.2	A few discussion items	369
11.4.3	Avoiding the trie	370
11.5	Take-home messages	371
11.6	Historical remarks	371
A	Probability theory	377
A.1	Some rules	377
B	Modular arithmetic	381
B.1	Modular congruence vs. modulo operator	381

Chapter 1

Introduction

Contents

1.1	Motivation	4
1.2	Definitions and assumptions	6
1.2.1	Problems	6
1.2.2	Algorithms	7
1.2.3	Computer model	9
1.2.4	Algorithm analysis	10
1.3	Asymptotic notation	13
1.3.1	Tight asymptotic bounds	16
1.3.2	Growth rates	18
1.3.3	Friends of O and Ω	19
1.3.4	Rules for asymptotic notation	21
1.4	Algorithm analysis revisited	28
1.4.1	Step 1. Describe the algorithm idea	28
1.4.2	Step 2. Give a detailed description	29
1.4.3	Step 3. Argue correctness	29
1.4.4	Step 4. Analyze the efficiency	30
1.4.5	Comparing algorithms	32
1.4.6	Analysis of recursive algorithms	32
1.5	Run-time analysis in special situations	38
1.5.1	Average-case analysis	38
1.5.2	Randomized algorithms	45
1.5.3	Amortized analysis and potential functions (cs240e)	48
1.6	Take-home messages	52
1.7	Historical remarks	54

1.1 Motivation

Welcome to cs240! This course is about designing data structures and algorithms so that we can manage data efficiently. You should already have some experience programming with arrays and lists, and most problems could somehow be solved using only arrays and lists. But in this course we don't just want to solve problems "somehow", we are about *efficiency*: we want to find the solution as fast as possible. This predominantly means trying keep the run-time small, but space is also important because the computer may run out of memory and may start swapping, which slows down computation.

We will see many examples where the naive implementation is vastly slower than what we could do if we store the data appropriately (hence "data structures") or if we are careful in how we process it (hence "data management"). Thus we will study how to build structures where we can efficiently insert and search for data (and perhaps also delete it), and study how to sort data. The emphasis is on how to analyze in a *theoretical* way how well such data structures and algorithms can perform. In particular, it is not good enough to say "it works well when I implement it"; we strive to find theoretical justification for why an algorithm ought to perform well.

Why should you care? Most of the data structures and algorithms that we will see in this course are considered "standard" and often they are available in libraries such as STL. Nevertheless, it is useful to study them for two reasons. One reason is that you may at some point in your programming or research career be asked to solve something that is a lot like the problems studied here, but slightly different so that the standard algorithms cannot be applied directly. So you may need to modify the code from the standard library, for which it really helps to understand why it works. The second, and perhaps even more important reason for studying this course is that the goal is for you to understand not only *how* to solve the problem, but to understand *why* one should solve the problem this way. This trains logical thinking and problem solving skills, which should come in handy no matter what your profession in the future will be.

Expected background Course cs240 builds on top of previous computer science courses, and in particular expands on the material that you should have seen in cs136 or cs145. In particular you are expected to know

- arrays and linked lists,
- strings,
- stacks and queues,
- abstract data types,
- recursive algorithms,
- binary trees,
- sorting,
- binary search, and
- binary search trees.

We will (very briefly) review most of these, but read your notes from the predecessor course and/or open any standard textbook for a detailed review.

Much of the analysis in this course will be done theoretically, and involve working with summations and recursions. You should be familiar with these from your math predecessor courses, and we will also occasionally need limits, derivatives and integrals. The following lists some specific concepts that we need:

- **Logarithms:**

- $c = \log_b(a)$ means $b^c = a$. E.g. $n = 2^{\log n}$.
- $\log(a)$ (in this course) always means $\log_2(a)$
- $\log(a \cdot c) = \log(a) + \log(c)$, $\log(a^c) = c \log(a)$
- $\log_b(a) = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a(b)}$, $a^{\log_b c} = c^{\log_b a}$
- $\log x < x$.
- $\ln(x) = \text{natural log} = \log_e(x)$, $\frac{d}{dx} \ln x = \frac{1}{x}$
- $\log^i(x)$ is a shortcut for $(\log(x))^i = \log(x) \cdot \dots \cdot \log(x)$ (with i factors). $\log \log x$ stands for $\log(\log(x))$.
- concavity: $\lambda \log x + (1-\lambda) \log y \leq \log(\lambda x + (1-\lambda)y)$ for $0 \leq \lambda \leq 1$

- **Factorial:**

- $n! := n(n-1)(n-2) \cdots 2 \cdot 1 = \text{number of ways to permute } n \text{ elements}$
- $\log(n!) = \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in \Theta(n \log n)$

To evaluate summations and recursions, we will typically use induction, which you are expected to be familiar with. The following states some known evaluations; you may use these without proof. For some of these facts we also state the result in asymptotic notation (defined formally in Section 1.3) because this may be easier to remember and is usually sufficient for our run-time and space analysis.

- **Arithmetic sequence:** $\sum_{i=0}^{n-1} (a + di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2)$ if $d \neq 0$.

We will especially use that $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$.

- **Geometric sequence:**

$$\sum_{i=0}^{n-1} a r^i = \begin{cases} a \frac{r^n - 1}{r - 1} & \in \Theta(r^{n-1}) \quad \text{if } r > 1 \\ na & \in \Theta(n) \quad \text{if } r = 1 \\ a \frac{1 - r^n}{1 - r} & \in \Theta(1) \quad \text{if } 0 < r < 1. \end{cases}$$

We will especially use that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ and that $\sum_{i=0}^{n-1} \frac{1}{2^i} = 2 - (\frac{1}{2})^{n-1}$; therefore $\lim_{n \rightarrow \infty} \sum_{i=0}^{n-1} \frac{1}{2^i} = 2$.

- **Harmonic sequence:** $H_n := \sum_{i=1}^n \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$ where $\gamma \approx 0.57721$ is the Euler-Mascheroni constant.

- A few more inequalities: $\sum_{i=1}^n \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1)$, $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$ and $\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$. More generally, $\sum_{i=1}^n i^k \in \Theta(n^{k+1})$ for $k \geq 0$.

We will also need a few ideas from a probability course (such as STAT230), in particular random variables (typically denoted X), the expected value $E[X]$, and (for the enriched section) variance $V[X]$. Some basic rules that we will need are listed below, and some further material (mostly needed for the enriched section) is in Appendix A.

- **Linearity of expectation:** $E[aX] = aE[X]$, $E[X + Y] = E[X] + E[Y]$, where X, Y are random variables and a is a constant.
- If X is a random variable that takes on only non-negative integer values, then $E[X] = \sum_{i=0}^{\infty} i \cdot P(X = i) = \sum_{i=1}^{\infty} P(X \geq i)$.

We also briefly need the notation of a finite automaton that you should have encountered in cs241.

Last but not least, it cannot be emphasized enough that cs240 does not just tell you what data structure or algorithm to use, instead the emphasis is on arguing *why* this is a good data structure or algorithm to use. As such, there will be many proofs and arguments of correctness. You should be very familiar with how to do such arguments (and in particular, proofs by induction and arguing program correctness via loop invariants) from courses such as math239/249 and cs245.

1.2 Definitions and assumptions

In this section, we briefly review some standard terms in algorithm design and some assumptions that we make. In particular, we will explain all the terms of the following statement (which is a typical statement to make when analyzing algorithms):

merge-sort is a recursive algorithm that solves the Sorting Problem in $O(n \log n)$ run-time and $O(n)$ auxiliary space.

Most of these terms should be familiar from your introductory computer science courses (such as cs136), so the following is for the most part only a rather brief refresher.

1.2.1 Problems

Contrary to how it is used in common language, in computer science a *problem* has a precise definition: It is a *language*, i.e., a (possibly infinite) subset of possible words of an alphabet, and we want to know whether a given word belongs to the language. For this course, we do not need to be quite so precise, and instead let a problem consists of a description of all possible inputs (“*instances*”) and a requirement on the desired output. In this data structures course, we will primarily study two problems:

- In the *Sorting Problem*, the input consists of a set of n items that can be compared. Usually we assume that the items are integers stored in an array $A[0..n-1]$, but sometimes we will instead assume that the items in $A[0..n-1]$ are classes that contain a field *key* (and we want to sort by *key*), and sometimes we will sort a list, rather than an array.

The desired output is a *sorting permutation*, i.e., a permutation π such that permuting the input according to π puts it in sorted order. (We will discuss this in more detail in Section 3.) Most of the algorithms that we see will not actually return the sorting permutation, instead they simply *put* array A into this sorted order. The two versions of sorting are equivalent: Any algorithm that can put an array into sorted order can be used to obtain the sorting permutation. (Proving this is left as an exercise.)

The input items for the Sorting Problem do not necessarily have to be distinct, though for ease of description we often assume that they are. It is not hard to see that this makes no difference to the difficulty of the problem because we can modify the input-values slightly or apply a tie-breaker when comparing them (details for this are again left as an exercise).

- In the *Search Problem*, the input consists of a set of n items, as well as some restrictions on the item that we are searching for. The desired output is a reference to the item that we are searching for, or some indication (such as an output of **FAIL**) that there is no such item. There is no fixed rule what should happen if there is more than one item; we will sometimes be content with returning one of them and sometimes ask for returning all of them. We distinguish two subtypes of searches:

- In a *structured search*, only a very restricted kind of search can be done. We model this by assuming that we *search by keys*, i.e., each given item has a unique identifier (the *key*) and we are looking for all items whose key is either equal to a given k or falls into a given range $[k', k'']$. Typically items have much more information than just the key associated with them, so they are *key-value-pairs* (KVPs), containing a key and some other values that are irrelevant for the search.
- In an *unstructured search*, we do not know beforehand what type of search will be happening. Needless to say that this makes it much harder to store the items to make search efficient. The example that we will study here is *string search*, where we are given a text T and are looking for a substring P within this text. We do not know either P or T beforehand.

Searching is *the* main problem when faced with a large amount of data to store (why would you store it if you never looked at items in it later?) and will occupy the bulk of this course.

There are of course many other problems that one could study; for example we will not even touch upon the vast world of graph algorithms. These are left to the successor course cs341.

1.2.2 Algorithms

In its most precise definition, an algorithm is a Turing machine (a concept that you will see in greater detail in a theory of computation course such as cs360 or cs365, and cs341 may also briefly touch on it). Again this is more precise than really needed for our course here, and we will instead stick with the following, less precise, definition. An *algorithm* is a step-by-step process that carries out a series of computations on the given input. It is *correct* if for every possible instance the algorithm obtains the desired output; we say that such an algorithm *solves*

176 the problem.

177 Most of our algorithms will be *deterministic*, i.e., at any given time there is only one possible
178 next step to be applied. However, we will sometimes see *randomized algorithms* where the next
179 step may depend on the outcome of a random experiment. We will also (very briefly) see *non-*
180 *deterministic* algorithms where there may be multiple allowed next steps, and we are content as
181 long as for some possible choice the output is as desired.

182 To make algorithms easier to understand, we usually break them up into smaller chunks
183 (depending on your favorite programming language these might be called *functions*, *methods* or
184 *sub-routines*). The entire algorithm itself is also viewed as one function, which means it can use
185 itself as a sub-routine. If this happens then the algorithm is called *recursive*. While recursive
186 algorithms are often very short and easy to understand, in practice a *non-recursive* algorithm
187 that does the same would be preferred. (The recursive algorithm uses much time and space to
188 store temporary information so that it can return from a sub-routine correctly, and therefore
189 tends to be slower.)

190 Notice that the definition of algorithm is simply a “list of steps”; we do not put any particular
191 restriction on how these steps are described. This is in contrast to a *program*, which is a
192 specific *implementation* of an algorithm in a programming language of your choice. A program
193 means that the algorithm has been translated into a language that a computer can understand.
194 Unfortunately, most (if not all) of these languages are not easy for humans to understand,
195 and much brain-space is wasted on getting the syntax just right and including programming-
196 language-related details. To communicate an algorithm to a human, we therefore use *pseudocode*,
197 which is a program with all details omitted that are obvious to humans. Since “obvious” is in the
198 eye of the beholder, there are no fixed rules of what exactly should/shouldn’t be in pseudo-code,
199 but here are some suggestions for how to shorten things:

- 200 • omit syntax such as semi-colons and parentheses,
- 201 • omit all checking for incorrect input,
- 202 • do not show full code for symmetric cases,
- 203 • use English descriptions for obvious operations such as “swap”

204 However, the pseudo-code *must* be understandable to humans. In particular, do keep paren-
205 theses and variable-declarations where they are needed to make the meaning clear. And, even
206 more so than in programs, you *must* use clear variable-names, indentation and comments. Also,
207 while the way to describe a loop is quite flexible (any of “for i = 1...n”, “for i∈{1,...,n}”
208 “for i←1 to n”, and “(for i=1; i≤n; i++)” are perfectly fine), being sloppy about the
209 range of the loop is not acceptable—it must be clear for exactly which values the code is exe-
210 cuted, and mathematical notation such as \leq or $<$ is advised. The example below shows a small
211 piece of pseudo-code. (You may have seen this code before; it is a particularly slow solution to
212 the Sorting Problem called *bubble-sort*.)

213 This course is quite independent of the programming language that is used for the program
214 (the algorithms should be feasible in all of them), but the pseudo-code that is given here “smells”
215 a bit of C++ and Java (see e.g. Algorithm 1.4 on Page 34 for a longer example).

Algorithm 1.1: *bubble-sort*($A, n \leftarrow A.size$)

```

1 for  $i \leftarrow 0$  to  $n-2$  do
2   for  $j \leftarrow 0$  to  $n-i-2$  do
3     if  $(A[j] > A[j+1])$  then swap  $A[j]$  and  $A[j+1]$ 

```

1.2.3 Computer model

We are only studying algorithms that are correct, i.e., convert the given instance into the desired output. Implicit in the definition of correctness is the assumption that the algorithm should do so in a finite amount of time. The main objective in this course is to find algorithms for which this “run-time” is small. But how do we measure the run-time, or put differently, what exactly can an algorithm do in one step? This would be easy to define if the algorithm were modeled as a Turing machine (which comes with a well-defined notion of one step), but since we gave a more general definition of algorithm, we must clarify the computer model. We are using here the *Random Access Machine (RAM) model*¹ which assumes the following:

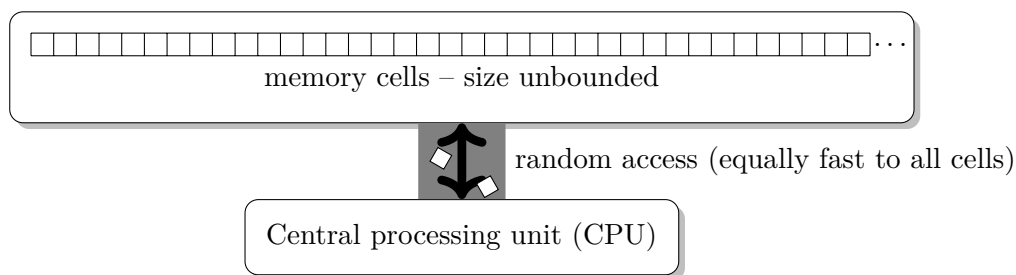


Figure 1.1: The random access machine model.

- The computer consists of a set of memory cells, each of which stores one integer (where the integer might be encoding a character, a reference, or even a command). In much of the CS literature these cells are called *computer words*, but since we will use “word” for something very different later we will stick with the term “memory cell” for cs240.
- There is no a-priori limit on the number of memory cells that the algorithm can use. (This is not realistic, though in the age of very cheap memory available in the cloud it is also not too unrealistic.)
- All memory cells are equal, and in particular, accessing any given memory cell takes the same (constant) amount of time. (Again this is not realistic; memory cells that are near the CPU are *much* faster to access than memory cells in the cloud that are accessed via

¹‘Random’ here has *nothing* to do with random variables or the randomized algorithms that we will see later; perhaps a better name would have been ‘everywhere-equally-fast access machine’, but the term RAM has been established as standard.

the internet. We will return to this in Chapter 11.)

- A memory cell is big enough to store any integer that we deal with.

This is not realistic: The memory cells in a real computer are typically restricted to 32 or 64 bits, and storing a number that is bigger than 2^{64} hence poses a problem. The RAM model is a *model*, not a computer. If the numbers get so big that they do not fit into a cell, then one should treat them as words, rather than as a number, and break them into parts that do fit into a cell.

- Any *primitive operation* takes the same (constant) amount of time.

What counts as “primitive operation” depends on which version of the RAM model one considers (there are several). For purposes of cs240, we will permit the following: addition, subtraction, multiplication, division, taking mod, rounding, comparison, and the control-operations of accessing a memory cell, calling a sub-routine, entering a loop, breaking or returning.

Assuming that all these take the same amount of time is unrealistic (surely dividing is not as fast as adding?). Also, sometimes we might want other operations (e.g. \sqrt{n} , $\log(n)$) and it is not clear how long these take. There are other models of computers (e.g. the *real RAM* which allows real numbers and trigonometric functions but then forbids mod and rounding), but the above list of primitive operations will be sufficient for cs240. (We will occasionally need $\lfloor \log n \rfloor$, but will then compute it from scratch with a simple for-loop.)

1.2.4 Algorithm analysis

With the RAM model defined, we can define the *run-time* of an algorithm to be the number of memory accesses plus the number of primitive operations. The *space* used by an algorithm is the total number of memory cells that are ever accessed (for reading or writing or both) by the algorithm. We assume here that the algorithm is smartly designed to re-use space that has been freed, so we measure the maximum amount of space that is used at some point during the algorithm. On the other hand, we do not allow to use space unless it has been *initialized*, i.e., explicitly set to some value (possibly NIL to indicate ‘no valid item stored’). We do not always include initialization-steps in our pseudo-codes, but it must always be done.

Recall that an *instance* is one possible input for a problem. With our computer model clarified, we can now define the *size* of an instance: it is the number of memory cells needed to store it. We nearly always use n for the size of an instance, and write \mathcal{I}_n for the set of instances of size n . When analyzing the space of an algorithm, we are most interested in how much space is required other than the (obviously needed) space for the input; this is called the *auxiliary space*. The *complexity* of an algorithm refers to how much effort this algorithm takes; in this course this usually refers to the run-time (our main criterion that we aim to keep small). But in some other contexts it may refer to space or other criteria, so it is safer not to use the term “complexity” on its own unless it is clear from the context that you mean the run-time. Typically larger instances will take more time and space, so what we measure is the complexity *relative to n* , i.e., over all instances in \mathcal{I}_n .

How do we determine the run-time and/or auxiliary space? One method to do so would be to do *experiments*, i.e., to implement the algorithm, let it run on some instances, and measure the time/space that has been used. However, there are some major drawbacks to this approach: The outcome depends on the quality of the implementation, the hardware used, and the instances that were used for the experiment. Last but not least, it requires an implementation, which may be costly to create and test. Therefore in this course we focus on *theoretical* analysis of algorithms, which is done on the pseudo-code, and hence independent of the hardware/software environment. Experiments are sometimes used to decide between algorithms that are equally good in theory (or where the theoretical analysis is simply too complicated to do), but whenever possible, one should do a theoretical analysis first to “weed out” some algorithm ideas that are unlikely to do well in practice.

To judge the run-time an algorithm adequately, we should take into account how big its input was. (Sorting 1000 items ought to take longer than sorting 10 items.) Formally, define the *size* of an instance I to be the number of memory cells that are occupied by I . For example, when sorting integers the size of the instances is the number of integers, because we assumed that memory cells are big enough to store the integers. The size of an instance is denoted by $|I|$, and typically n is used as a convenient shortcut for $|I|$.

The run-time of an algorithm is hence measured relative to n . To see how this is done, consider the pseudo-code of *bubble-sort* that we saw earlier.

Algorithm 1.1: *bubble-sort*($A, n \leftarrow A.size$) // repeated

```

1 for  $i \leftarrow 0$  to  $n-2$  do
2   for  $j \leftarrow 0$  to  $n-i-2$  do
3     if ( $A[j] > A[j+1]$ ) then swap  $A[j]$  and  $A[j+1]$ 

```

We proceed as follows. First, count a constant amount of time for each primitive operation. (Typically, each line of code means a constant number of operations, though sometimes such line of codes hide function-calls and hence larger amounts of work.) Then, sum up over each loop to obtain the run-time. In this example, the run-time hence is at most

$$c_0 + \sum_{i=0}^{n-2} \left(c_1 + \sum_{j=0}^{n-i-2} (c_2 + c_3) \right),$$

where c_0 is the time for initiating the entire procedure, c_1 is the time for advancing the outer loop in line 1, c_2 is the amount of time that for advancing the inner loop in line 2, and c_3 is the maximum amount of time taken in line 3. (This depends on whether we swap or not; by default we take an upper bound.)

There are a few issues with this analysis. First, we have no idea what c_0, c_1, c_2 and c_3 are—this depends on the computer hardware, and on exactly how many primitive operations one counts (e.g. how long does it take to advance a loop? Do we only count increasing the counter, or also

checking the boundary-condition, or also loading the counter from the memory, or....)? Second, evaluating this expression becomes cumbersome. For this (comparatively simple) example one could still evaluate the sum precisely. But since we do not know what the constants are, this is quite meaningless, because all that we can say here is that it is “some quadratic term plus some lower-order terms”. So rather than having to evaluate the sums exactly, we will soon introduce *asymptotic notation* which makes this idea of “some quadratic term” more precise. You should have seen big- O before, but we will in the next section review it in detail and introduce its “friends”.

A problem with this analysis is that the run-time sometimes depends on the instance. We already got a glimpse of this when defining c_3 , because the run-time for line 3 depends on whether we need to do a swap or not. The difference can be even more dramatic. Consider the example in Algorithm 1.2. (You may have seen it before; it is *insertion-sort* which sorts an array by repeatedly moving $A[i]$ into the correct place in $A[0..i-1]$.)

Algorithm 1.2: *insertion-sort*($A, n \leftarrow A.size$)

Input : Array A of size at least n
1 **for** $i \leftarrow 1$ **to** $n - 1$ **do**
2 $j \leftarrow i$
3 **while** $j > 0$ **and** $A[j] < A[j - 1]$ **do**
4 swap $A[j]$ and $A[j - 1]$
5 $j \leftarrow j - 1$

If we try the same run-time analysis, then we would have to sum up over the executions of the while-loop. However, it is not clear how often the while-loop operates, since this depends on the entries in A (i.e., the input instance). So we must clarify what to do if we have different run-times for different instances of the same size.

Let us use $T_{\mathcal{A}}(I)$ to denote the run-time of algorithm \mathcal{A} on an instance I . Then the *worst-case run-time* of \mathcal{A} is

$$T_{\mathcal{A}}^{\text{worst}}(n) := \max_{I \in \mathcal{I}_n} T_{\mathcal{A}}(I)^2$$

where \mathcal{I}_n denotes all instances of size n . Put differently, within all instances of size n , we consider the worst case, i.e., the instance (of this given fixed size) for which the run-time is as bad as it could be. The worst-case run-time gives us a guaranteed upper bound on the run-time for an instance of size n . By default, when asked to analyze the run-time of algorithm, we analyze the worst case. For example in *insertion-sort*, in the worst-possible case, the while-loop executes i times. (An analysis then shows that the worst-case run-time is quadratic, but we will come back to this later.)

²In later discussion, we will usually omit \mathcal{A} and often even omit the quantifier *worst* or *avg*, since it will be clear from the context which algorithm we are analyzing under which scenario.

Analyzing the worst case makes sense in time-critical situations, where you need a good guarantee on the run-time, no matter what happens. But we will soon (when studying *quick-sort*) see an example of an algorithm where the worst-case run-time is quite bad, but the algorithm works well in practice. One method to provide theoretical evidence for this is to consider the *best-case run-time*, which measures the smallest possible run-time (among all instances of fixed size n). However, just like the worst-case time is sometimes too pessimistic, the best-case time is sometimes too optimistic. A more realistic measure of actual performance is the *average-case run-time*, which averages over all possible inputs. We will return to this (and some other methods of measuring run-time in special situations) in Section 1.5.

1.3 Asymptotic notation

Let us now review the O -notation that you should have seen in previous courses. Recall that the main idea is to find a simple expression that captures how a function (such as the run-time of an algorithm) is growing. Roughly speaking, we ignore constant factors and lower-order terms. Formally, we have:

Definition 1.1. Let $f(x), g(x)$ be two functions on the positive reals. We say that $f(x) \in O(g(x))$ if there exist $c > 0$ and $n_0 \geq 0$ such that $|f(x)| \leq c|g(x)|$ for all $x \geq n_0$.

Put differently, $O(g(x))$ is the set of all those functions f on the positive reals that satisfy $f(x) \in O(g(x))$ for all $x \geq n_0$, where $c > 0$ and n_0 are constants.

There are numerous synonyms for “ $f(x) \in O(g(x))$ ”: We also say that “ f is in big- O of g ”, “ f is asymptotically upper-bounded by g ” or “ f asymptotically grows no faster than g ”. Also, while technically one should always write “asymptotically” in these expressions, in reality it often gets dropped because *all* our analysis of run-time and space will use asymptotic notation and so one tends to forget. Finally, sometimes writing “for all $x \geq n_0$ ” will be a bit awkward; we use the phrase “for sufficiently large x ” to express “there exists a constant n_0 such that it holds for all $x \geq n_0$ ”.

Figure 1.2 illustrates two examples, and in particular serves to explain the use of the two constants n_0 and c . Consider the left figure where $f(x) = \sqrt{x}$ while $g(x) = x^2$. Clearly \sqrt{x} is smaller than x^2 , right? Actually, wrong. If $0 < x < 1$ then $\sqrt{x} > x^2$. But this is only in this small interval, for all “big enough” x indeed $\sqrt{x} \leq x^2$. This is exactly what the constant n_0 is used for: We do not have to care about small values, we only care about what happens in the long term.

To explain why we need constant c , consider the example on the right where $f(x) = 2x^2 + 10x$ while $g(x) = x^2$. Clearly $f(x) > g(x)$. But they increase at the same rate, and so (at least for big x) behave in a similar fashion. This is why we permitted to multiply $g(\cdot)$ by a constant c : It allows to forget lower-order terms (because $2x^2 + 10x \leq 3x^2$ for sufficiently large x) and multiplicative factors.³

³Allowing an arbitrary factor c is perhaps *too* permissive: Clearly an algorithm with n^2 primitive operations

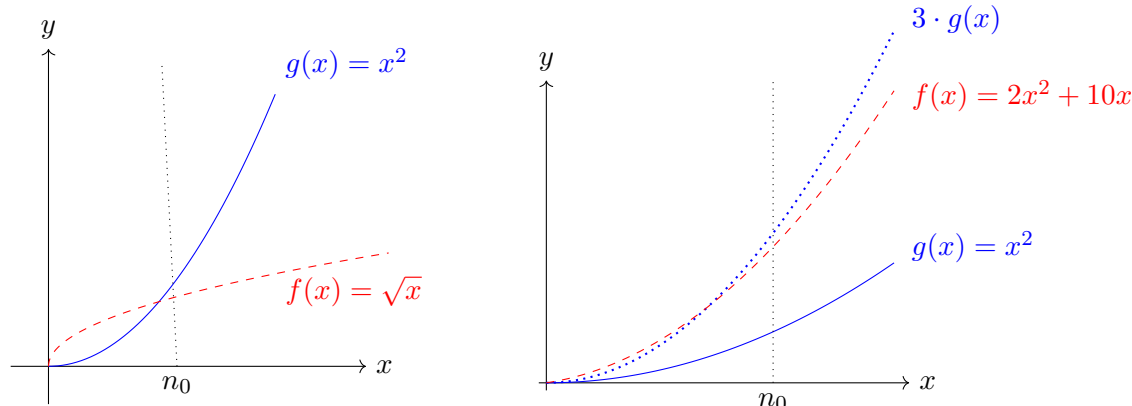


Figure 1.2: Comparing function-growth. We have $\sqrt{n} \in O(n^2)$ and $n^2 + 10 \in O(n^2)$.

Example: Consider the example $f(n) = 2n^2 + 3n + 11$ and $g(n) = n^2$. From what was said above, it should be quite obvious that $f(n) \in O(g(n))$. But let us see how one would prove this formally *from first principle*, i.e., from the original definition using c and n_0 . For such a proof, one needs to state what constants n_0 and c are used, and then show that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$. Define the constants to be $c = 16$ and $n_0 = 1$. Then for all $n \geq n_0 = 1$ we have

$$2n^2 + 3n + 11 \underset{n \geq 1}{\leq} 2n^2 + 3n^2 + 11n^2 = 16n^2 = cn^2$$

and so $f(n) \in O(g(n))$.

Note that c and n_0 are not unique, for example $n_0 = 4$ and $c = 11$ would have worked as well. As a general rule, using a bigger n_0 can never hurt (because you are cutting off more values that don't need to be checked), but it does need to be finite. With a bigger n_0 , it is (for O -notation) often feasible to find a smaller c , but do not work too hard on finding the smallest possible c , because an O -bound holds as long as you find one finite one.

How does one come up with the constants? Typically write yourself down what you want to hold, in our example that would be

$$2n^2 + 3n + 11 \stackrel{!}{\leq} cn^2.$$

(The exclamation-mark is used to indicate that we do not know this yet, but want to achieve it.) This should immediately tell you that you need $c \geq 2$. In fact, you will need a bigger c to

would be faster than an algorithm with $100n^2$ primitive operations. But there is no good way to express in a precise way which multiplicative factor is significant, especially since the factors are very rough estimates. So we stick with asymptotic notation that allows *any* factor since we have no better tools available.

make up for the lower-order terms. How big should you make it? *Any* $c > 2$ can actually be shown to work for this example, but the bound is easier to prove if you make c bigger. Once you have picked your c , re-write what you need to show, in this case (using, say, $c = 3$) we need

$$(3 - 2)n^2 - 3n - 11 \stackrel{!}{\geq} 0.$$

If you are lucky then you can argue this directly. If you don't see a direct argument, then study the function on the left-hand-side. Where are its roots? (In other words, make the inequality an equality and solve for n .) For $c = 3$, the roots are $\frac{3}{2} \pm \sqrt{\frac{9}{4} + 11}$.

Typically, one could use as n_0 the largest root. But recall: using a bigger n_0 does not hurt, so rather than handling awkward fractions and square-roots, round up generously. So for example we could see that $\frac{3}{2} + \sqrt{\frac{9}{4} + 11} \leq 2 + \sqrt{5 + 11} = 6$, so let's use $n_0 = 6$. Now you have a good guess for a pair of (c, n_0) . Verify that it actually works, i.e., prove that $f(n) \leq c g(n)$ for $n \geq n_0$ for your choice of c and n_0 . In the example

$$2n^2 + 3n + 11 \underset{n \geq 6}{\leq} 2n^2 + 3n \cdot \frac{n}{6} + 11 \cdot \frac{n}{6} \cdot \frac{n}{6} = 2n^2 + \frac{1}{2}n^2 + \frac{11}{36}n^2 < 3n^2 = cn^2.$$

If you can't prove the inequality, then the most likely culprit is that your choice of c was too small. Try a bigger c and repeat until you succeed.

All the above should happen as scratch-work (and, with more experience, simply in your head). Once you found an appropriate (c, n_0) pair, the proof is usually a two-liner: state the constants c and n_0 , and then write the above equation that shows that $f(n) \leq c g(n)$ for $n \geq n_0$.

A few comments: You may have noticed that in the above proof from first principle, we did not include the absolute-value-signs that were part of the inequality " $|f(x)| \leq c|g(x)|$ for all $x \geq n_0$ ". Why did we do not need to do this? A quick inspection of the function f shows that it is *always* positive. As such, there is no need for the clutter of absolute-value signs. More generally, the functions that we analyze in cs240 these typically measure run-time and/or space-usage. As such, they ought to be positive. When doing some arithmetic to obtain bounds, we may sometimes end up with functions (such as $n^2 - 3n$) that are negative for some small values, but they will always be positive for sufficiently large n . Therefore, in cs240 we typically ignore the absolute-value signs.⁴

The other thing that you may have noticed is that in the definition of big- O we used x for the parameter of function f , but in the example we suddenly used n . Principally it does not matter what we call the parameter, but typically x represents a real number while n suggests that the function is defined for positive integers. Again, in cs240 we use asymptotic analysis mostly for functions that measure run-time and/or space-usage relative to the size of the input,

⁴We added them in our definition because the concept of asymptotic notation can be used in many other contexts (e.g. to measure how well a numerical approximation performs), and here the functions might well be negative and so the absolute-value-signs are necessary.

hence their domain are non-negative integers. So most of the time we will use n for the name of the parameter. (This also explains the use of ‘ n_0 ’ for the cutoff point.) Some exceptions occur when we are using calculus-tools such as limits and derivatives, because those look more familiar when calling the parameter x .

We should also briefly discuss how to analyze a function f that is not defined everywhere. The inequality “ $|f(x)| \leq c|g(x)|$ for all $x \geq n_0$ ” is then required to hold only in the *domain* of f , i.e., for all those x where $f(x)$ is defined. We usually choose $g(x)$ to be a very simple function (such as $g(x) = x^2$) that is defined everywhere, but if for some reason we restrict the domain of g as well, then the inequality only has to hold for values where both f and g are defined.

1.3.1 Tight asymptotic bounds

The O -notation is useful for getting upper bounds on the run-time without having to worry about the exact constants for each line of code. However, the definition allows a bit too much freedom. For example, we know that $2n^2 + 3n + 11 \in O(n^2)$, but we could also show that $2n^2 + 3n + 11 \in O(n^{100})$. (Use the same c and n_0 and the same proof, except in the final step add $cn^2 \leq cn^{100}$.) So if you have an algorithm \mathcal{A} where the run-time is $2n^2 + 3n + 11$, then the statement “algorithm \mathcal{A} has run-time $O(n^{100})$ ” is perfectly correct. But that’s probably not what you meant to express; you meant to say something like “the run-time is quadratic and we can’t get a better bound than that”.

The latter part of the statement is expressed via “big-Omega”, denoted Ω and also called *asymptotic lower bound*.

Definition 1.2. Let $f(x), g(x)$ be two functions on the positive reals. We say $f(x) \in \Omega(g(x))$ if there exist $c > 0$ and $n_0 \geq 0$ such that $|f(x)| \geq c|g(x)|$ for all $x \geq n_0$.

Note that this definition is almost exactly the same as big- O (Definition 1.1), except that the inequality has been reversed. See Figure 1.3 for an example. The same motivation applies for using n_0 and c : We only care what happens for sufficiently large n , and we do not want to care about constant factors.

Example: Consider the function $\frac{1}{2}n^2 - 5n$, which we claim to be in $\Omega(n^2)$. To prove this from first principle, we need to choose n_0 and c such that $f(n) \geq c \cdot n^2$ for all $n \geq n_0$. In this particular example, choose $n_0 = 20$ and $c = \frac{1}{4}$. Then for $n \geq n_0 = 20$, we have

$$\frac{1}{2}n^2 - 5n = \frac{1}{4}n^2 + \underbrace{\frac{1}{4}n^2 - 5n}_{n \geq 20} \geq \frac{1}{4}n^2 + \frac{1}{4}20n - 5n = \frac{1}{4}n^2 = c \cdot n^2.$$

Much the same rule applies for choosing c and n_0 , except that this time, if inequalities don’t work out, then you should make c *smaller*. (That’s only natural, seeing that this will help with proving that something is $\geq cg(n)$.)

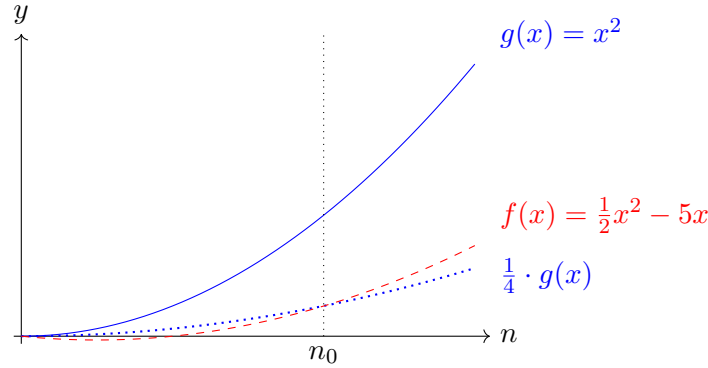


Figure 1.3: Comparing function-growth. We have $\frac{1}{2}x^2 - 5x \in \Omega(x^2)$.

417 **Definition 1.3.** Let $f(x), g(x)$ be two functions on the positive reals. We say that $f(x) \in \Theta(g(x))$
 418 if $f(x) \in O(g(x))$ and $f(x) \in \Omega(g(x))$. So there exist constants $n_0 \geq 0$ and $c_1, c_2 > 0$ such that
 419 $c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|$ for all $x \geq n_0$.

420 Alternative names for this are “ $f(x)$ is in Theta of $g(x)$ ”, or “ $f(x)$ is asymptotically the
 421 same as $g(x)$ ”. Saying “the run-time is $\Theta(n^2)$ ” expresses what we wanted above: “the run-time
 422 is $O(n^2)$ and this is tight”. So for meaningful bounds, we should always use Θ .

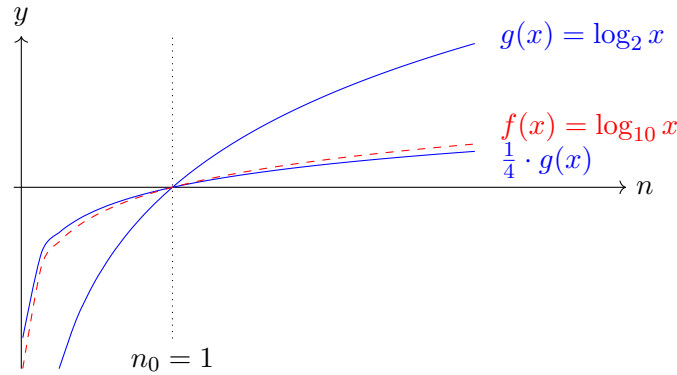


Figure 1.4: Comparing function-growth. We have $\log_{10} x \in \Theta(\log x)$.

Example: We claim that $\log_{10}(x) \in \Theta(\log x)$. (Recall that \log without any base indicated means that the base is 2.) To see this, first prove $\log_{10}(x) \in O(\log x)$. This holds because with increasing base the log gets smaller, hence $\log_{10} x \leq \log_2 x = 1 \cdot \log x$ for $x \geq 1$. (We have used $c_2 = 1$ and $n_0 = 1$.) Now show $\log_{10}(x) \in \Omega(\log x)$. To do so, choose $n_0 = 1$ and $c_1 = \frac{1}{4}$. We know $\log_{10}(x) \geq 0$ for $x \geq n_0$ (we chose $n_0 = 1$ to avoid having to work with absolute values)

and using log-rules we have

$$\log_{10}(x) = \frac{\log_2(x)}{\log_2(10)} \approx \frac{\log x}{3.32} \geq \frac{1}{4} \log x = c_1 \log x.$$

The astute reader will notice that we actually could have proved both O and Ω in one shot, using $n_0 = 1$ and $c_1 = c_2 = 1/\log 10$. But that's fairly rare; usually when proving a Θ -bound we have to prove O -bound and Ω -bound separately.

Second example: Consider the function $f(n)$ whose domain are the integers and which satisfies $f(n) = n$ if n is odd and $f(n) = 3n$ if n is even. We claim that $f(n) \in \Theta(n)$. To see this, observe first that $f(n) \leq 3n$, and so $f(n) \in O(n)$ (use $c = 3$). Also, $f(n) \geq n$, and so $f(n) \in \Omega(n)$ (use $c = 1$).

A useful (but sloppy) notation: If we know that $T(n) \in \Theta(n)$, then we know that $cn \leq T(n) \leq c'n$ for some constants c, c' and all sufficiently large n . In some situations (such as when doing arithmetic with asymptotic notation—more on this below), it will get quite annoying to have to deal with two constants. For this reason, we will sometimes write $T(n) \approx c \cdot n$ and do arithmetic with $c \cdot n$ instead. This is technically incorrect (for example, what c would you use for our second example above?). Ideally it would only be used to give a rough idea of what bound we expect to achieve, and the bound would then formally be proved by proving O -bound and Ω -bound separately.

1.3.2 Growth rates

We will use asymptotic notations mainly for analyzing the run-time $T(n)$ of some algorithm. Some names have been developed for commonly occurring asymptotic run times. We list these *growth rates* below, and also study how the growth rate affects the performance at the size gets big. If we compare an instance of size n with one that is twice as big, how much bigger will the run-time be? We say that $T(n)$ is

- *constant* if $T(n) \in \Theta(1)$. Here '1' stands for the function that is identical to 1, i.e., $f(n) = 1$ for all n . For example, looking up element $A[i]$ in a given array takes constant time. If the instance is twice as big, the time is still exactly the same; it is independent of the problem size.
- *logarithmic* if $T(n) \in \Theta(\log n)$, i.e., $T(n) \approx c \log n$ for some constant c . For example, binary search in an array has logarithmic run-time. If the instance is twice as big, then the run-time increases by just a constant, because $T(2n) \approx c \log(2n) = c(\log n + 1) = c \log n + c \approx T(n) + c$. Occasionally we will speak of *poly-logarithmic* run-time if $T(n) \in \Theta(\log^d n)$ for some constant d .

- *linear* if $T(n) \in \Theta(n)$, i.e., $T(n) \approx cn$ for some constant c . For example, searching whether an array contains a particular value takes linear time in the worst case. If the instance is twice as big, then the run-time doubles, because $T(2n) \approx c(2n) = 2cn \approx 2T(n)$. Occasionally we will speak of *super-linear* run-time if $T(n)$ is more than linear. (In the notation that we will define below, $T(n) \in \omega(n)$.)
- *linearithmic* if $T(n) \in \Theta(n \log n)$, i.e., $T(n) \approx cn \log n$ for some constant c . This name is rarely used, but the run-time is very common, for example for *merge-sort* that we will see below.
- *quadratic* if $T(n) \in \Theta(n^2)$, i.e., $T(n) \approx cn^2$ for some constant c . For example, *insertion-sort* has quadratic run-time in the worst case. If the size of the input doubles, then the run-time quadruples.
- *polynomial* if $T(n) \in \Theta(n^d)$ for some constant $d \geq 0$. This encompasses most of the above growth-rates, but is usually used only when d is big (but constant). Examples of this are rare in cs240, but in later courses you will see problems such as matrix-multiplication or all-pair-shortest-path where we have polynomial algorithms, but no quadratic ones are known. If the size of the instance doubles, then the run-time increases by a factor, but at least it is a constant factor (namely, 2^d , which is constant since d is constant).
- *exponential* if $T(n) \in \Theta(b^n)$ for some constant $b > 1$. For the following illustration, let us assume that $T(n) \approx 2^n$. If the size of the instance doubles, then $T(n)$ increases hugely! Namely, $T(2n) \approx 2^{2n} = (2^n)^2 \approx (T(n))^2$, so the run-time is roughly squared. To convince yourself that this is a really awful run-time, perhaps a reverse argument is even more impressive. Imagine you get a new computer that is twice as fast as the one you had before. How much bigger an instance can you solve in the same time t ? Well, if the old computer could solve an instance of size n , so $t \approx 2^n$, then the new computer can do twice as many steps, so it can handle size n' where $2^{n'} = 2t = 2 \cdot 2^n = 2^{n+1}$. So the so-much-faster computer can handle an instance that is just *one* unit bigger. This should explain why exponential run-time is usually considered unacceptable for a solution for a problem. You will study the gap between polynomial and exponential run-time in much more detail in cs341.

1.3.3 Friends of O and Ω

Big- O and big- Ω express upper and lower bounds, but permit the possibility of equality (at least asymptotically). In some situations we want to express that the run-time of one algorithm is strictly smaller than the run-time of another. We could do this by giving tight asymptotic bounds for both and then compare, but how to do this if we are speaking about a hypothetical algorithm? For example, how would you express the concept “there exists no sorting algorithm whose run-time is strictly smaller than $O(n \log n)$?”⁵ For this reason, we introduce more asymptotic notation.

⁵This statement is *almost* true; it needs a small modifier added. We’ll see more on this in Chapter 3.

491 **Definition 1.4.** Let $f(x), g(x)$ be two functions on the positive reals. We say that $f(x) \in o(g(x))$
 492 if for any constant $c > 0$ there exists an $n_0 \geq 0$ s.t. $|f(x)| \leq c|g(x)|$ for all $x \geq n_0$.

493 We say that $f(x) \in \omega(g(x))$ if for any constant $c > 0$ there exists an $n_0 \geq 0$ s.t. $|f(x)| \geq$
 494 $c|g(x)|$ for all $x \geq n_0$.

495 Alternative names for o are “little- o ” or “asymptotically strictly smaller”, while ω is pro-
 496 nounced “little- ω ” or “asymptotically strictly larger”. Using ω will be *very* rare in this course
 497 (we introduce it mostly for completeness’ sake), but little- o will be useful in two ways. We can
 498 use it to state impossibilities (“there exists no sorting algorithm with run-time $o(n \log n)$ ”⁵) or to
 499 express lower-order terms if we care about constants (“the number of comparisons is $2n + o(n)$ ”).

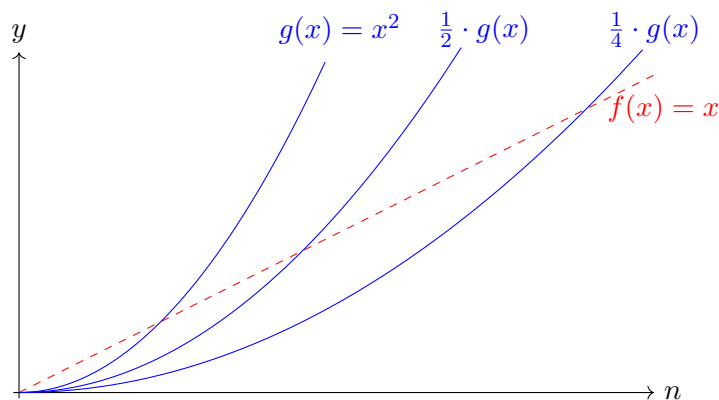


Figure 1.5: Comparing function-growth. We have $x \in o(x^2)$.

500 To understand little- o better, consider the example in Figure 1.5. It would be very easy to
 501 show that $x \leq x^2$ for sufficiently large x . What little- o expresses is that the same holds even if
 502 we scale down the function on the right-hand side. So we’d have to show that $x \leq \frac{1}{2}x^2$, which is
 503 still true for sufficiently large x , but the cut-off point is bigger. And we’d also have to show that
 504 $x \leq \frac{1}{4}x^2$, where the cut-off point is even bigger. And we’d have to show it for increasing smaller
 505 scales of $g(x)$, and the cut-off point gets bigger and bigger. As a rule of thumb for little- o proofs,
 506 the difficult case should be if c is very small, and your cut-off point n_0 should depend on the c
 507 that you are given.

508 The main difference between little- o and big- O is the change of quantifiers. For big- O , the
 509 inequality must hold for *one* constant c . *You* get to choose c , and making it bigger will be to
 510 your advantage. In contrast, for little- o the inequality must hold for *all* constants c . You have
 511 no control over c , it will be given to you (and in particular, it could be arbitrarily small). Your
 512 only control is over n_0 .

Example: Let us prove from first principle that $f(n) \in o(g(n))$ for $f(n) = n$ and $g(n) = n^2$.
 Thus, for any $c > 0$ we have to find a suitable n_0 such that $f(n) \leq cg(n)$. As should have been

clear from Figure 1.5, the cut-off point n_0 must depend on c —the smaller c , the bigger n_0 needs to be. In this example, for a given $c > 0$, define $n_0 = \frac{1}{c}$. Then for $n \geq n_0$ we have

$$f(n) = n = n \cdot c \cdot \underbrace{\frac{1}{c}}_{n \geq n_0 = \frac{1}{c}} \leq cn^2 = cg(n).$$

Coming up with the right formula for n_0 is not as easy for little- o . You are *not* allowed to make any assumptions about c . Similar to what we did for big- O , the approach is to write down what you want to hold, but this time you need to re-formulate it without substituting a value for c until you get a lower bound for n_0 (and it should depend on c).

Bounds involving o and ω are proved from first principle only very rarely. Instead, one nearly always uses the limit-rule that we will see in the next section.

1.3.4 Rules for asymptotic notation

The following introduces some rules for asymptotic notations, which will almost always be enough to obtain asymptotic bounds for us, because run-times and auxiliary space are usually a mix of polynomials and log-factors.

Some basic rules: Asymptotic notation uses words that are very similar to math-(in)equalities: $O, \Omega, \Theta, o, \omega$ “feel” a lot like $\leq, \geq, =, <, >$. However, properties that you know about inequalities do *not* automatically transfer to asymptotic notation; they must be proved (and sometimes they do not hold). Here are some rules for non-negative functions⁶, together with some ideas of how one could prove them. (You should be able to fill in the details of those proofs yourself.)

- **Identity:** $f(n) \in \Theta(f(n))$. This is proved using $c = 1$ and any n_0 .
- **Constant multiplication:** $K \cdot f(n) \in \Theta(f(n))$, for any positive constant K . The proof is again easy, use $c = K$ and any n_0 .
- **Transitivity:** If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$. For if $f(n) \leq c_1 g(n)$ and $g(n) \leq c_2 h(n)$ for sufficiently large n , then $f(n) \leq c_1 \cdot c_2 h(n)$ for sufficiently large n .
Likewise, if $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$.
This has some useful implications such as
- **Dominance:** If $f(n) \in O(g(n))$, and $g(n) \leq h(n)$ for all sufficiently large n , then $f(n) \in O(h(n))$. This holds because $g(n) \in O(h(n))$. Likewise $f(n) \in \Omega(g(n))$, and $g(n) \geq h(n)$ for all sufficiently large n , then $f(n) \in \Omega(h(n))$.
- **Addition:** If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$. For if c, c' and n_0, n'_0 are the constants for the O -bounds for f_1 and f_2 , then using $\max\{c, c'\}$ as multiplier and $\max\{n_0, n'_0\}$ as cut-off point one easily shows the bound for $f_1(n) + f_2(n)$. Likewise if $f_1(n) \in \Omega(g_1(n))$ and $f_2(n) \in \Omega(g_2(n))$, then $f_1(n) + f_2(n) \in \Omega(g_1(n) + g_2(n))$.

⁶For negative functions, sometimes absolute value signs need to be inserted; we leave this as exercise.

That's perhaps not quite what you wanted—the term inside the O or Ω should be as simple as possible, so you'd like to simplify this further. For that, we need other rules.

- **Maximum-rule for O :** If a function $h(n)$ belongs to $O(f(n) + g(n))$ then it belongs to $O(\max\{f(n), g(n)\})$. Here $\max\{f(n), g(n)\}$ denotes the function that takes the pointwise maximum, i.e., for every n it takes on whichever value of $f(n)$ and $g(n)$ is the bigger one. In particular we have $f(n) \leq \max\{f(n), g(n)\}$ and $g(n) \leq \max\{f(n), g(n)\}$ for all n , and therefore $f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$. This implies the maximum-rule by applying “dominance” and “constant multiplication”.

Maximum? You might have expected something like $O(f(n)) + O(g(n))$, but this notation makes no sense ($O(f(n))$ is a *set*, not a number, so we can't add it—see the warning below). But we want some way to combine the two functions into one, and \max seems the right way since we measure how the function grows.

- **Maximum-rule for Ω :** If a function $h(n)$ belongs to $\Omega(f(n) + g(n))$ then it belongs to $\Omega(\max\{f(n), g(n)\})$. This holds because $f(n) + g(n) \geq \max\{f(n), g(n)\}$.

You may have expected ‘minimum’ here, not ‘maximum’, but again the maximum is the right way to combine since we measure how much the functions grow, and this is dominated by the one that is larger.

How do we use this? Consider for example $f(n) = 2n^2 + 32$. One can easily prove that $f(n) \in \Theta(n^2)$ from first principles, but let us do it with the above rules instead. We know that $n^2 \in \Theta(n^2)$ by the identity-rule, and likewise $1 \in \Theta(1)$. Applying the rule about constant multiplication, therefore $2n^2 \in \Theta(n^2)$ and $32 \in \Theta(1)$. By the addition-rule therefore $f(n) = 2n^2 + 32 \in \Theta(n^2 + 1)$. Since $\max\{n^2, 1\} = n^2$ for sufficiently large n , therefore $f(n) \in \Theta(n^2)$ by the maximum-rule.

A warning word about arithmetic: It is very tempting to treat asymptotic notation as if $O(f(n))$ were a number, and to write things like “ $n^2 + n = O(n^2) + O(n) = O(n^2)$ ”. Don't. $O(n^2)$ is a *set*, not a number. There is no such thing as arithmetic with sets. We did write “ $O(n^2 + n) = O(n^2)$ ”, and this is correct because the ‘=’ refers to set-equality. But you cannot say “ $O(n) < O(n^2)$ ”, you can only say “ $O(n) \subset O(n^2)$ ”.

Let's demonstrate why you shouldn't with an example. You should know that $f(n) := \sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$. But here is a “proof” (incorrect) that $f(n) \in O(n)$.

Show $f(n) \in O(n)$ by induction on n . In the base case, $n = 1$, we have $f(1) = 1 \in O(1)$. Assume we know that $f(n) \in O(n - 1)$. Then $f(n) = n + f(n - 1) = n + O(n - 1) = O(n) + O(n) = O(2n) = O(n)$.

This result is nonsense, because $f(n) \in \Omega(n^2)$, and this implies $f(n) \notin O(n)$ (see later). Every step in the proof seems perfectly correct—except that there was arithmetic intermixed with asymptotic notation, leading to a wrong result.

Having said that, there is one exception that we occasionally do. Recall the earlier statement that an algorithm “uses $2n + o(n)$ comparisons”. Here we did intermix arithmetic and o -notation,

and the concept should really be expressed as “uses $2n + f(n)$ comparisons, where $f(n) \in o(n)$ ”. But this is cumbersome, so mixed expression has become the norm for final statements of a fact. But avoid using asymptotic anywhere within proofs that manipulate equations (especially if you are doing recursions or proofs by induction).

At the times when we really need to do arithmetic when asymptotic expressions are involved, we will use one of the following options:

- One can replace asymptotic notation by an explicit constant factor c , as we did in Section 1.3.2. For example, for proving an O -bound, use a sentence such as “Let c be a constant that is bigger than all the constants hidden in the O -notation”. Then you know that any expression “ $f(n) + O(n)$ ” can be upper-bounded by “ $f(n) + cn$ ”, and you can do arithmetic with the latter. For Θ -notation, one should technically do this twice (with two constants c, c' , one for the O -bound and the other for the Ω -bound), though it is common to use the sloppy notation $\approx cn$ instead.
- One can define “one time unit” to be a suitably big constant amount of time, and do arithmetic with time units. Thus, use a sentence such as “Assume a time unit is defined such that any operation which we counted as $O(1)$ takes at most one time unit.” Then “ $f(n) + O(n)$ ” can be upper-bounded by “ $f(n) + n$ time units”, and again you can do arithmetic with this. (The enriched section will see a few examples of this in Section 1.5.3.)
- Often the run-time of an algorithm is dominated by one particularly difficult operation. For example for sorting we will consider *key-comparisons*, i.e., comparisons between two input-items. We only do this for algorithms where the run-time is known to be in $O(\#key-comparisons)$. Therefore we will compute the number of key-comparisons, rather than the run-time, and hence avoid using asymptotic notation. (We will see a few examples of this in Chapter 3.)

Limit-rule: Useful for all asymptotic-notation is the following limit-rule.

Lemma 1.1. Let $f(x), g(x)$ be two functions that satisfy $f(x), g(x) > 0$ for all sufficiently large x . Assume that $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)}$ exists (call it L). Then

$$f(x) \in \begin{cases} o(g(x)) & \text{if } L = 0 \\ O(g(x)) & \text{if } L < \infty \\ \Omega(g(x)) & \text{if } L > 0 \\ \omega(g(x)) & \text{if } L = \infty \end{cases}$$

Let us first see how to apply this, by revisiting the example we had earlier, with $f(n) = 2n^2 + 3n + 11$, $g(n) = n^2$. With the limit-rule, it would be even easier to show that $f(n) \in \Theta(g(n))$, because

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{2n^2 + 3n + 11}{n^2} = \underbrace{\lim_{n \rightarrow \infty} \frac{2n^2}{n^2}}_{=2} + \underbrace{\lim_{n \rightarrow \infty} \frac{3n}{n^2}}_{=0} + \underbrace{\lim_{n \rightarrow \infty} \frac{11}{n^2}}_{=0} = 2.$$

In fact we can analyze all polynomials that way; we will return to this below.

Let us get some intuition why the limit-rule should hold. The ratio of f and g gets arbitrarily close to L as x gets bigger. Put differently, $f(x) \approx L \cdot g(x)$ for all sufficiently large x . If $0 < L < \infty$, then this is what we wrote for $f(n) \in \Theta(g(n))$, using constant L . (However, recall that this is sloppy notation, and we should be a bit more precise—see below.) On the other hand, if $\lim_{x \rightarrow \infty} f(x)/g(x) = 0$, then this not only says that $f(x)$ is smaller than $g(x)$, but grows much much slower than $g(x)$. This feels the same as the condition $f(x) \in o(g(x))$. (This one is actually exactly the same, as we will see in Lemma 1.2.) Now let us make this intuition more precise.

Proof. (cs240e)⁷ We only show that $L < \infty$ implies $f(x) \in O(g(x))$; the other cases are similar and left as exercises. We need to show that $f(x) \in O(g(x))$, i.e., there exist constants $c > 0$ and n_0 such that $|f(x)| \leq c \cdot |g(x)|$ for all sufficiently large x . Let n_0 be so large that $f(x), g(x) > 0$ for all $x \geq n_0$, it will then be enough to show that $f(x) \leq cg(x)$ (i.e., we can forget about the absolute value signs) because we will restrict x to be at least n_0 . Now that we know what we want to show, let us recall what limit means:

For all $\varepsilon > 0$ there exists a constant n_ε s.t. $|\frac{f(x)}{g(x)} - L| \leq \varepsilon$ for all $x \geq n_\varepsilon$.

Here we cannot just drop the absolute value signs, because we have no idea whether we approach the limit from below or above, but this will not be a problem since we only need an upper bound. Rephrasing what we know for $\varepsilon = 1$, we know that there exists a constant n_1 such that

$$\frac{f(x)}{g(x)} - L \leq \left| \frac{f(x)}{g(x)} - L \right| \leq 1 \quad \text{for all } x \geq n_1.$$

Since $g(x) > 0$ for $x \geq n_0$, we can multiply with $g(x)$ without affecting the inequality, so

$$f(x) \leq (L + 1)g(x) \quad \text{for all } x \geq \max\{n_1, n_0\}.$$

So if we set $c = L + 1$, then c is a constant (by $L < \infty$) and $c > 0$ since L is positive (because $f(x)$ and $g(n)$ are for sufficiently large n). So we have $f(x) \leq cg(x)$ for all $x \geq \max\{n_0, n_1\}$ as desired. \square

The limit-rule is an equivalence *if* we know that the limit exists. For example, if $f(x) \in O(g(x))$ and *if* the limit $L = \lim_{n \rightarrow \infty} \frac{f(x)}{g(x)}$ exists, then $L < \infty$. (A proof of this is left as an exercise.)

⁷A “(cs240e)” at a proof or a statement or even a whole section means that this is usually not covered in the regular section, and usually is covered in the enriched section. However, the determining factor for whether you need to read it is whether your instructor covered it or not.

No limit? What do we know if the limit does not exist? Nothing! Consider the two examples illustrated in Figure 1.6:

- Set $f(n) = n(2 + \sin(n\pi/2))$ and $g(n) = n$. The limit $\lim_{n \rightarrow \infty} f(n)/n$ does not exist, and yet $f(n)$ and $g(n)$ grow the same asymptotically. (Observe that $n \leq f(n) \leq 3n$.)
- Set $f(n) = 1 + n^2(1 + \sin(n\pi/2))$ and $g(n) = n$, where again the limit does not exist. We have $f(n) = 1 + 2n^2$ for infinitely many values of n , and therefore for any constant c and n_0 there exists some $n \geq n_0$ with $f(n) > cn$. This means $f(n) \notin O(n)$. But also $f(n) = 1$ for infinitely many values of n , which implies $f(n) \notin \Omega(n)$. So $f(n)$ and $g(n)$ cannot be compared asymptotically.

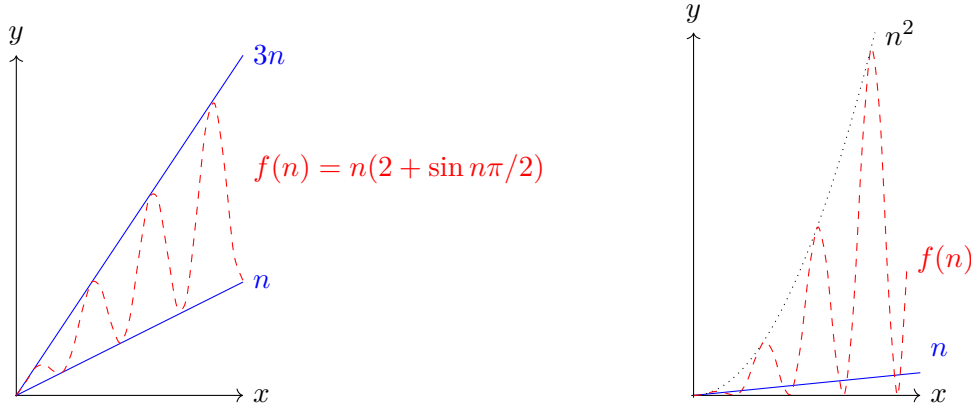


Figure 1.6: Two oscillating functions $f(n)$ where $\lim_{n \rightarrow \infty} \frac{f(n)}{n}$ does not exist. This tells us nothing about whether $f(n) \in \Theta(n)$.

Fortunately such examples are rare (and involve some form of oscillating functions, which would be very unusual for run-times and space). Still, do not conclude anything about big- O or big- Ω if the limit does not exist. Surprisingly enough, the situation is different for little- o (and symmetrically little- ω): If the limit does not exist, then $f(n) \notin o(g(n))$. We show here the contrapositive.

Lemma 1.2. (*cs240e*) Let $f(n), g(n)$ be two functions that satisfy $f(n), g(n) > 0$ for sufficiently large n . Then $f(n) \in o(g(n))$ if and only if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ exists and equals 0.

Proof. Necessity holds by the limit-rule. To show sufficiency, assume that $f(n) \in o(g(n))$. Fix an arbitrary $\varepsilon > 0$; we must show that for some sufficiently large n_ε we have $|\frac{f(n)}{g(n)}| < \varepsilon$ for all $n \geq n_\varepsilon$. By the definition of little- o , we know that for all $c > 0$ there exists an n_0 such that $f(n) < c g(n)$ for all $n \geq n_0$. Fix the specific n_0 that applies for $c = \varepsilon$. We also know that for some n'_0 we have $f(n) > 0$ and $g(n) > 0$ for all $n \geq n'_0$. Setting $n_\varepsilon = \max\{n_0, n'_0\}$, we therefore have for all $n \geq n_\varepsilon$

$$|\frac{f(n)}{g(n)}| = \frac{f(n)}{g(n)} < \frac{c g(n)}{g(n)} = c = \varepsilon$$

638 as desired. □

639 **Polynomial rule:** You will need to use the limit-rule directly only rarely, but it is very handy
 640 for proving other extremely useful rules. Here is the first one. Recall that a function f is a
 641 *polynomial* if we can write $f(n) = c_0 + c_1n + c_2n^2 + \cdots + c_dn^d$ with $c_d \neq 0$. The term c_dn^d is
 642 called the *leading term*. The polynomial-rule states

643 if $f(n)$ is a polynomial for which the constant c_d at the leading term is positive,
 644 then $f(n) \in \Theta(n^d)$.

Do not sweat too much about the condition “ c_d is positive”; this will always be true for run-times. The proof is extremely simple if one knows rules for limits:

$$\lim_{n \rightarrow \infty} \frac{c_0 + c_1n + c_2n^2 + \cdots + c_dn^d}{n^d} = \underbrace{\lim_{n \rightarrow \infty} \frac{c_0}{n^d}}_0 + \underbrace{\lim_{n \rightarrow \infty} \frac{c_1n}{n^d}}_0 + \cdots + \underbrace{\lim_{n \rightarrow \infty} \frac{c_dn^d}{n^d}}_{c_d} = c_d$$

645 and the result holds by the limit-rule. Where did we use $c_d > 0$? Recall that the limit-rule
 646 requires $f(n) > 0$ for sufficiently large n , and this holds if and only if $c_d > 0$.

Logarithm-rules: We know multiple results concerning logarithms and asymptotic notation. First observe that

$$\log_b(n) \in \Theta(\log n) \quad \text{for any base } b > 1,$$

or put differently, all logarithms grow the same way asymptotically. We already proved this for $b = 10$ earlier, but it is proved more easily with the limit-rule since

$$\lim_{n \rightarrow \infty} \frac{\log_b n}{\log_2 n} = \lim_{n \rightarrow \infty} \frac{\frac{\log_2 n}{\log_2 b}}{\log_2 n} = \lim_{n \rightarrow \infty} \frac{1}{\log_2 b} = \frac{1}{\log_2 b}.$$

Next observe that

$$\log n \in o(n),$$

or put different, logarithms grow much much slower than linear functions. This holds since

$$\lim_{n \rightarrow \infty} \frac{\log n}{n} = \lim_{n \rightarrow \infty} \frac{\frac{\ln n}{\ln 2}}{n} = \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{\ln n}{n} \stackrel{\text{l'Hôpital}}{=} \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{1} = \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{1}{n} = 0.$$

647 (Recall that $\ln n$ denotes the natural log, whose derivative is $\frac{1}{n}$.) We have used here l'Hôpital's
 648 rule: If the numerator and the denominator in a limit either both go to 0, or both go to
 649 infinity, then we can replace the terms by their derivative. This is a very common tool to use
 650 when applying the limit-rule to find asymptotic behavior. This rule can be made even stronger.

651 **Lemma 1.3.** For any positive constants $c > 0$ and $d > 0$, we have $(\log n)^c \in o(n^d)$.

Before proving this, we'd like to point out that the difficult case occurs if c is huge and d is tiny, e.g. $(\log n)^{10} \in o(n^{1/1000})$. Also recall the shortcut $\log^c n$ for $(\log n)^c$.

Proof. (cs240e) The idea is a repeated application of l'Hôpital's rule, driven by the size of c and d . Formally, we first prove by induction on k that $\lim_{n \rightarrow \infty} \frac{\ln^k n}{n} = 0$. This was proved for $k = 1$ above, and for $k > 1$ we have

$$\lim_{n \rightarrow \infty} \frac{\ln^k n}{n} \stackrel{\text{l'Hôpital}}{=} \lim_{n \rightarrow \infty} \frac{k(\ln^{k-1}(n)) \frac{1}{n}}{1} = k \cdot \lim_{n \rightarrow \infty} \frac{\ln^{k-1}(n)}{n} \stackrel{\text{Hypothesis}}{=} k \cdot 0 = 0.$$

Now fix arbitrary positive constants c and d . Then

$$\lim_{n \rightarrow \infty} \frac{\ln^c n}{n^d} = \left(\lim_{n \rightarrow \infty} \frac{\ln^{c/d} n}{n} \right)^d \leq \left(\lim_{n \rightarrow \infty} \frac{\ln^{\lceil c/d \rceil} n}{n} \right)^d = 0^d = 0$$

Since $\log^c n = \left(\frac{1}{\ln 2}\right)^c \ln^c n$, we have $\lim_{n \rightarrow \infty} \frac{\log^c n}{n^d} = \left(\frac{1}{\ln 2}\right)^c \cdot 0 = 0$, which proves the claim by the limit-rule. \square

Relationships between asymptotic notations. A favorite exam questions is to ask students to fill the following table with TRUE or FALSE (for a variety of choices of $f(\cdot)$ and $g(\cdot)$):

		Is $f(n) \in \dots (g(n))$?			
$f(n)$	$g(n)$	o	O	Ω	ω
$\log n$	\sqrt{n}				

Some of those entries you will have to prove using the limit-rule or polynomial-rule or from first principles, but many of the entries can simply be deduced because there are relationships between classes of asymptotic notation. Let us fill the example to demonstrate this. We know that $\log n \in o(\sqrt{n})$ (apply the generalized log-rule with $c = 1$ and $d = \frac{1}{2}$). So the leftmost entry is TRUE. This implies that the next entry is also TRUE (i.e., $\log n \in O(\sqrt{n})$) due to the following:

$$f(n) \in o(g(n)) \text{ implies } f(n) \in O(g(n))$$

This is quite obvious, since big- O demands an inequality for *one* constant c while little- o demands it for *all* constants c , so little- o is more restrictive. Symmetrically, we also know

$$f(n) \in \omega(g(n)) \text{ implies } f(n) \in \Omega(g(n)).$$

The next entry in the table is FALSE (i.e., $\log(n) \notin \Omega(g(n))$) due to the following:

$$\text{If } f(n) \in o(g(n)), \text{ then } f(n) \notin \Omega(g(n)).$$

Intuitively, this should be clear: the left-hand-side says “strictly smaller” while the right-hand-side says “greater-or-equal”. But intuition is not always correct for asymptotics notation, and the proof is not trivial. (See below.) Note that the contra-positive of this statement could also be useful:

$$\text{If } f(n) \in \Omega(g(n)), \text{ then } f(n) \notin o(g(n)).$$

Coming back to the table, we can deduce that the rightmost entry is also FALSE, i.e., $\log n \notin \omega(n)$. For if we had $\log n \in \omega(n)$ then $\log(n) \in \Omega(n)$, which we just argued to be false. So the table can be filled by doing only one proof; the other three entries can simply be deduced. Generally it holds that in this table, at most two of the entries can be TRUE. (For some bizarre functions, such as the right one in Figure 1.6, fewer than two entries are TRUE.)

As a second example, we claimed earlier that $f(n) \in \Omega(n^2)$ implies $f(n) \notin O(n)$. This can be argued as follows. We know that $n \in o(n^2)$ by the limit-rule. If we had $f(n) \in O(n)$, then hence $f(n) \in o(n^2)$, hence $f(n) \notin \Omega(n^2)$. Contradiction.

Now we provide the missing proof:

Lemma 1.4. *Let $f(n), g(n)$ be two functions that satisfy $f(n), g(n) > 0$ for sufficiently large n . If $f(n) \in \Omega(g(n))$, then $f(n) \notin o(g(n))$.*

Proof. (cs240e) Consider $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$. In the first case, this limit does not exist. Then by Lemma 1.2 we know that $f(n) \notin o(n)$.

So consider the case where the limit L exists. Since $f(n) \in \Omega(g(n))$, we know that there exists some constant $c > 0$ such that $f(n) \geq c g(n)$ for all sufficiently large n . Since $f(n), g(n)$ are positive, therefore $\frac{f(n)}{g(n)} \geq c > 0$ for all sufficiently large n . This implies $L > 0$, and therefore again $f(n) \notin o(n)$ by Lemma 1.2. \square

1.4 Algorithm analysis revisited

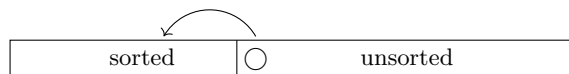
Now that we have all the tools in place, let us revisit the main goal of this course: How to solve a problem efficiently? There are two parts to this question: What needs to be done to describe one particular solution (i.e., algorithm), and how do we decide between multiple algorithms?

To describe one algorithm, there are typically four steps. We will illustrate them here by arguing that *insertion-sort* (which we saw in Algorithm 1.2) solves the Sorting Problem.

1.4.1 Step 1. Describe the algorithm idea

This should typically in English prose, with 1-2 sentence and/or a picture that describe the crucial insights. Omit all details.

For example for insertion-sort, this could be “Keep a sorted part, and repeatedly insert one more item into its correct place in it”. We are very sloppy here, we have given no details of how to store the sorted part, but it gets the idea across.



1.4.2 Step 2. Give a detailed description

This typically means pseudo-code, but a description in English can sometimes be good enough. Either way, the description should be detailed enough that an experience programmer can convert this to code without thinking much.

The pseudo-code for *insertion-sort* was given earlier (Algorithm 1.2), but we repeat it here for ease of reading.

Algorithm 1.2: *insertion-sort*($A, n \leftarrow A.size$)

Input : Array A array of size at least n

```

1 for  $i \leftarrow 1$  to  $n - 1$  do
2    $j \leftarrow i$ 
3   while  $j > 0$  and  $A[j] < A[j - 1]$  do
4     swap  $A[j]$  and  $A[j - 1]$ 
5      $j \leftarrow j - 1$ 
```

1.4.3 Step 3. Argue correctness

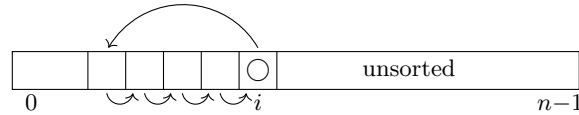
You should have seen in cs245 how to do this formally, with loop-invariants and proofs by induction. In cs240 we don't need to be quite so detailed, but you should give enough explanations and/or invariants in the pseudo-code that any cs245-student would be able to complete a formal proof without much thinking. Often the correctness argument can be integrated with the description of the idea (Step 1) and/or by breaking the pseudo-code into small parts and explaining what each part achieves.

There are some algorithms that are so extremely simple that one would be really hard-pressed to write more than "it's obvious" about correctness. Then the correctness-argument can be omitted. But if you are ever in doubt whether it is obvious or not, then it is not obvious and needs an argument.

One part of arguing correctness is to show that the algorithm actually terminates, even if there are loops. Most of the time this is obvious, but it is worth arguing for while-loops and *especially* for recursions why we make progress. Something (typically some parameter and/or the size of the problem) has to get smaller every time and have a lower bound (typically 0) at which the loop/recursion ends.

For *insertion-sort*, the while-loop terminates because j gets smaller every time and the loop stops when $j = 0$. As for why it works correctly, add "*// A[0..i-1] is sorted*" as a loop invariant for the for-loop. Then argue briefly that this holds initially (because $i = 1$ and so $A[0]$

715 is sorted) and that it holds whenever the while-loop terminates (because it terminates exactly
 716 when $A[i]$ has been moved to the index j^* where it belongs in the order, while the items in
 717 $A[j^*..i-1]$ have been moved over).



719 1.4.4 Step 4. Analyze the efficiency

720 You should analyze the run-time of your algorithm, typically in the worst-case scenario. You
 721 should also analyze the auxiliary space that is used. Both should be done using Θ -notation, i.e.,
 722 give tight bounds.

723 For *insertion-sort*, let us first do the auxiliary space, because this is easy. Beyond the space
 724 for the input, we only use two variables i and j , so the auxiliary space is $\Theta(1)$.

725 Now let us consider the run-time. The exact run-time would be determined by counting one
 726 unit per primitive operation. Using asymptotic notation, we can simplify this to “count $\Theta(1)$
 727 per line of code”. (This assume that the line of code does not hide any major work. If there are
 728 calls to other methods within it, then you need to analyze those methods separately and then
 729 count this much.) Then sum up the terms over the loops.

730 Before doing this for *insertion-sort*, let us do a simpler example from Algorithm 1.1 (we
 731 repeat the code here for ease of reading).

Algorithm 1.1: *bubble-sort*($A, n \leftarrow A.size$) // repeated

```

1 for  $i \leftarrow 0$  to  $n-2$  do
2   for  $j \leftarrow 0$  to  $n-i-2$  do
3     if ( $A[j] > A[j+1]$ ) then swap  $A[j]$  and  $A[j+1]$ 

```

732 Line 3 takes $\Theta(1)$ time. Advancing the for-loops also takes $\Theta(1)$ time. (This will be the last
 733 time we ever talk about the time to advance the loops. It is always $\Theta(1)$ per execution, and the
 734 inside of the loop also takes $\Theta(1)$, so this term can be ignored.)

Let c be a constant such that all these operations take $\approx c$ units of time. We do here a transition from asymptotic notation to an actual constant so that we can do arithmetic. We are cheating a bit, because technically we should use two constants c and c' , chosen such that all operations take between c and c' units of time. But again we use \approx as a convenient shortcut for this. The total run-time of this code is hence

$$\approx c + \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c.$$

These sums are simple enough that we could determine the closed form of this, but in general that won't be possible (and isn't needed). Instead, to obtain an O -bound, generously increase boundaries or even the terms until the result is easy to analyze. For example, we can say that

$$c + \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c \leq c + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c = c + \sum_{i=0}^{n-1} cn = c + cn^2 \in O(n^2).$$

In fact, often it is even possible to avoid fiddling with sums and constants altogether. We could also obtain a bound of $O(n^2)$ with the following argumentation. “Line 3 takes $O(1)$ time. The loop in line 2 executes at most n times, so lines 2-3 take $O(n)$ time. The loop in line 1 executes at most n times, so lines 1-3 take $O(n^2)$ time.”

However, we are not done. Recall that we want a *tight* bound, i.e., a Θ -bound. To prove this, we need to find a *lower* bound on the run-time. This is not quite so obvious here, because the inner loop might execute very little (e.g. when $i = n$). To show that $c + \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c$ is in $\Omega(n^2)$, it comes in really handy that we can throw away some terms to simplify the sums. In particular, the run-time is at most

$$c + \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c \geq \sum_{i=0}^{n/2-1} \sum_{j=0}^{n-i-2} c \geq \sum_{i=0}^{n/2-1} \sum_{j=0}^{n/2-1} c \geq c \frac{n}{2} \frac{n}{2} = c \frac{n^2}{4} \in \Omega(n^2).$$

Note that we silently assumed that n is divisible by 2. This is usually an acceptable assumption—if we can show an asymptotic bound for all even values of n , then it is fair to assume that it also holds for odd values since typically run-times change monotonically.

Now we return to *insertion-sort*. Recall that here the run-time depends on the instance; by default we analyze the worst-case run-time. Proving an O -bound on the worst-case is easy: The inner loop executes at most n times, so lines 3-5 take $O(n)$ time, and the outer loop executes at most n times and takes $O(n)$ time, so the worst-case run-time is $O(n^2)$.

We need to argue that this bound is tight (i.e., give an Ω -bound). First, let us point out that an Ω -bound for the worst-case run-time is something *different* than the best-case run-time. For the best-case run-time, we took the best possible input. For the lower bound on the worst-case run-time, we still take the worst possible instance I , then look at its run-time $T(I)$, and lower-bound this run-time. Actually, we do not necessarily have to take the instance that is the worst overall, but we do have to find some instance for which the run-time is large.

Specifically, pick as instance a reversely sorted array. At each execution of the for-loop, the array has been rearranged so that the first i items are sorted and the rest is unchanged. Hence $A[i]$ (the item that gets swapped forward) is smaller than all items in $A[0..i-1]$. In consequence, we must do i swaps for $A[i]$, and the number of swaps is at least

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Omega(n^2).$$

Clearly the run-time is at least as big as the number of swaps, so the worst-case run-time is in $\Omega(n^2)$.

Summarizing, the worst-case run-time for *insertion-sort* is $\Theta(n^2)$. A similar (and simpler) argument shows that the best-case run-time is $\Theta(n)$ (we need $\Omega(n)$ time for the outer loop alone, and $O(n)$ run-time is achieved if the array is sorted).

1.4.5 Comparing algorithms

Now we answer the second question: How do we *compare* algorithms? Our primary concern is run-time. As stated earlier, experimental comparison has numerous drawbacks, so mostly we will compare theoretical run-time. Typically, we will consider algorithm \mathcal{A}_1 to be better than \mathcal{A}_2 if the worst-case run-time of \mathcal{A}_1 is asymptotically strictly smaller than the one of \mathcal{A}_2 . Or to put it in formulas:

$$T_{\mathcal{A}_1}^{\text{worst}}(n) \in o\left(T_{\mathcal{A}_2}^{\text{worst}}(n)\right).$$

For example, *merge-sort* (which will be reviewed in the next section and has run-time $\Theta(n \log n)$) is considered to be better than *insertion-sort*, which has run-time $\Theta(n^2)$.

If the worst-case run-times are asymptotically the same, then typically we decide by bounds for the average-case run-time and/or the auxiliary space. Sometimes we will have a *space-time tradeoff*: One algorithm may be faster but use more space while another algorithm may be slower but use less space. So there isn't always one clear answer as to which algorithm is best; often the answer is "it depends".

One should point out that there are some pitfalls in using the worst-case asymptotic run-time bound for comparing two algorithms \mathcal{A}_1 and \mathcal{A}_2 . Let's say that \mathcal{A}_1 has worst-case asymptotic run-time $O(n^3)$ while \mathcal{A}_2 has worst-case asymptotic run-time $O(n^2)$. Does that mean that \mathcal{A}_2 is always better? Not necessarily. First, the worst-case considers only the worst-possible instance; there may well be many instances where the run-time of \mathcal{A}_2 is worse than the run-time of \mathcal{A}_1 . (For example, *insertion-sort* is much faster than *merge-sort* if the input is sorted.) Secondly, we forgot that we should use a *tight* bound, i.e., a Θ -bound. Using an O -notation is basically meaningless. If we knew that \mathcal{A}_1 has worst-case run-time $\Theta(n^3)$ and \mathcal{A}_2 has worst-case run-time $\Theta(n^2)$, then this could be taken as an indication that \mathcal{A}_2 is probably better. But even here there are problems. We could have $T_{\mathcal{A}_1}(n) \leq n^3$ while $T_{\mathcal{A}_2}(n) \leq 10^{20} n^2$. So it is true that for big enough n algorithm \mathcal{A}_2 is faster, but it takes staggeringly large values of n to achieve. So perhaps \mathcal{A}_1 is still faster in practice. Summarizing, using the worst-case run-time to compare two algorithms gives us some idea of who might be better in practice, but it is no guarantee. (But we also have no better tools for comparing available, short of implementing both and running experiments, with all the associated issues discussed earlier.)

1.4.6 Analysis of recursive algorithms

Let us do another big example of how to design an algorithm, to reinforce the concept and because it will bring up the topic of recursion. We return to the Sorting problem, and describe

788 and analyze an algorithm you might have seen before: *merge-sort*.

789 **Step 1:** Describe the idea. The idea of *merge-sort* is to split the input into two roughly
 790 equal-sized pieces, sort them recursively, and then merge the two sorted parts together.

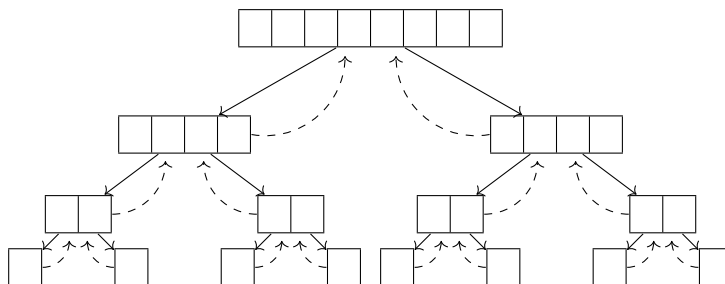


Figure 1.7: The idea of merge-sort.

791 **Step 2:** Give the pseudo-code. For this we need to give two code-fragments; one explains the
 792 merge-routine and the other does the actual merge-sort. See Algorithms 1.3 and 1.4.

Algorithm 1.3: *merge-sort*($A, n \leftarrow A.size, \ell \leftarrow 0, r \leftarrow n - 1, S \leftarrow \text{NIL}$)

Input : Array A of size at least n , $0 \leq \ell \leq r \leq n - 1$

(possibly uninitialized) array S of size n

1 **if** S is NIL **then** initialize it as array $S[0..n-1]$

2 **if** $(r \leq \ell)$ **then**

3 | **return**

4 **else**

5 | $m = \lfloor (r + \ell) / 2 \rfloor$

6 | *merge-sort*(A, n, ℓ, m, S)

7 | *merge-sort*($A, n, m + 1, r, S$)

8 | *merge*(A, ℓ, m, r, S)

793 The pseudo-code for *merge-sort* may not have been quite what you expected. A much easier
 794 way to describe the same idea would be to copy each half of A into a new array and recurse.
 795 E.g.

```

796 merge-sort(A,n)
797 if (n > 1) {
798     m = (n-1)/2
799     merge-sort(A[0..m])
800     merge-sort(A[m+1..n-1])
801     merge(A,0,n-1)
802 }
```


Algorithm 1.4: *merge*(A, ℓ, m, r, S)

Input : $A[0..n-1]$ is an array, $A[\ell..m]$ is sorted, $A[m+1..r]$ is sorted.
 $S[0..n-1]$ is an array

```

1 copy  $A[\ell..r]$  into  $S[\ell..r]$ 
2 int  $i_L \leftarrow \ell$ 
3 int  $i_R \leftarrow m+1$ 
4 for ( $k \leftarrow \ell; k \leq r; k++$ ) do
5   if  $i_L > m$  then  $A[k] \leftarrow S[i_R++]$ 
6   else if  $i_R > r$  then  $A[k] \leftarrow S[i_L++]$ 
7   else if  $S[i_L] \leq S[i_R]$  then  $A[k] \leftarrow S[i_L++]$ 
8   else  $A[k] \leftarrow S[i_R++]$ 

```

803 where the *merge*-routine is modified to temporarily assign array S . Conceptually, this is the
804 same algorithm. The reason why we chose our pseudocode is that it would be much more
805 efficient in practice. Passing an array as a parameter to a sub-routine means that a full copy of
806 the array is created. This is slow, and if we don't truly need a copy, then it is much better to
807 pass along the boundaries of the sub-array, as we did by passing ℓ and r . Secondly, allocating a
808 new array every time we call *merge* is slow. It is better to allocate an array only once and then
809 pass it along, as we did with array S .

810 So there is no one unique correct pseudocode; the level of detail depends on how much we
811 want to emphasize programming tricks that would make it better in practice (but usually do not
812 change the worst-case bound) versus the simplicity of the code. Many of our later pseudo-codes
813 will lean towards simplicity.

814 **Step 3:** Correctness. First argue that *merge* achieves that $A[\ell..r]$ is sorted afterwards. This
815 holds because the two parts of the array were sorted before, and we go through both parts
816 simultaneously and always copy over the smaller item. The correctness of *merge-sort* is then
817 quite obvious: We split the array in two parts and sort each recursively, and since *merge* then
818 merges them correctly, the end-result is sorted.

819 We should also briefly discuss that the recursion terminates. Recall that for recursions, the
820 typical way to do this is to argue that the size of the sub-problem gets smaller. For *merge-sort*,
821 we would measure the size of the sub-problem as $r - \ell$, because we only need to sort $A[\ell..r]$.
822 (Technically perhaps $r - \ell + 1$ would be more accurate, since that's the number of elements in
823 $A[\ell..r]$, but the '+1' makes no difference to the argument below.)

824 To see that the sub-problems are smaller, recall that we choose $m = \lfloor \frac{\ell+r}{2} \rfloor$ and let us re-write
825 this as $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lceil \frac{r-\ell}{2} \rceil$. We also know that $r > \ell$ (else we would have returned) and therefore
826 $0 \leq \lceil \frac{r-\ell}{2} \rceil < r - \ell$. In the first sub-problem, the size is $m - \ell$, and we have $m - \ell = \lceil \frac{r-\ell}{2} \rceil < r - \ell$.
827 In the second sub-problem, the size is $r - (m+1)$ and we have $r - (m+1) = r - \ell - \lceil \frac{r-\ell}{2} \rceil - 1 < r - \ell$.
828 So for both sub-problems the size is smaller and we terminate when the size reaches 0.

Step 4: Analysis. Whenever an algorithm uses sub-routines, we should analyze those first and merge their run-times in the formula for the main routine. So let us first analyze *merge*. This is straight-forward: The for-loop executes $r - \ell + 1$ times, and each execution takes constant time, so the run-time is $\Theta(r - \ell + 1)$, i.e., the size of the array that we were given to merge.

Since *merge-sort* is recursive, the worst-case run-time of *merge-sort* must be described as a *recursive function*, i.e., a function that depends on itself. Thus let $T(n)$ be the worst-case run-time of *merge-sort* when we have to sort n items (i.e., $r - \ell + 1 = n$.) Then $T(0) = T(1) \in \Theta(1)$, because for such small array-ranges we return immediately. If $n > 1$, then we split in half, recurse and merge. The merging takes $\Theta(n)$ time, so the run-time satisfies

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + f(n)$$

for some function $f(n) \in \Theta(n)$.

When analyzing run-times (especially for recursive algorithms) we very frequently make assumptions such as “ n is divisible as needed”. More generally we ignore small constant additive terms (e.g. resulting from rounding). We also ignore lower-order term in the asymptotically bounded function $f(n)$. (We will discuss below on an example why this generally does not distort the asymptotic bound on the run-time.) Thus we write the following sloppy recursion for $T(n)$:

$$T(n) \leq \begin{cases} c & \text{if } n \leq 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

for some constant c .

Resolving recursive functions Analyzing the growth rate of recursive functions is very easy with a proof by induction, presuming you can guess of how the function grows. Coming up with the right guess to make this induction work out is much harder. (You will see a useful method for this in cs341.) Here we will simply give you all the other recursions needed in the course (and also some hint of where they will be used). We state here only the recursive part of the formula (the base case is always $T(1) \in \Theta(1)$), and we use asymptotic notation rather than a constant (which is, just barely, acceptable since we do not do arithmetic with it).

Recursion	resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	<i>binary-search</i>
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	<i>merge-sort</i>
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	<i>heapify</i>
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	<i>quick-select</i>
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	<i>interpolation-search</i>
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	<i>range-search</i>

So now we have analyzed the worst-case run-time of *merge-sort* to be $\Theta(n \log n)$. Note that the run-time for *merge-sort* is actually independent of the instance—we always take $\Theta(n)$ time

for merging, and we always recurse the same way. So the worst case and best case always the same: $\Theta(n \log n)$.

We should also briefly analyze the space. Algorithm *merge-sort* allocates a full second array S in the outer-most recursion; this takes $\Theta(n)$ auxiliary space. Since we pass S along (rather than re-allocating new arrays), we otherwise only use $O(1)$ auxiliary space for local variables. But we must also consider how much space is used for the *recursion stack*. Recall that every time we enter a recursion, the current status of the memory is stored (on a stack) so that when the recursion finishes, the prior status can be restored. So the size of this stack is as big as the maximum number of recursions that are nested inside each other. (This is the reason why recursions tend to be slow in practice: creating entries for this stack takes time.) This space also counts as auxiliary space! In the case of *merge-sort*, this is not very important, because there are at most $\log n$ nested recursions (the size of the sub-problem halves with every recursion), so the stack uses $O(\log n)$ space, which is dominated by the $\Theta(n)$ space needed for S . But as we will see, in some other algorithms the recursion stack may be the dominating term for the auxiliary space.

Precise run-time analysis of *merge-sort* (cs240e) When stating the sloppy recursion for the run-time of *merge-sort*, we ignored a number of issues. In this part, we will do a completely correct (but much more tedious) analysis, so that you can see at least on an example why ignoring the issues does not pose problems. We will do this on a recursive formula that resembles the one of *merge-sort*.

Lemma 1.5. *Let $T(n)$ be a positive function that is defined on all integers $n \geq 0$. Assume there exists an $n_0 \geq 0$ such that $T(n)$ satisfies the following recursive formula for all $n \geq n_0$:*

$$T(n) \leq T(n') + T(n'') + f(n)$$

where $n', n'' \leq \frac{n}{2} + o(n)$ and $f(n) \in \Theta(n)$. Then $T(n) \in O(n \log n)$.

Before we prove this lemma, we point out where the difficulties are, and sketch how we overcome them:

- The parameter n' for the recursive part is roughly half as big as n , but there is an additive term that we must deal with and that is not even required to be a constant. Let us write k for this additive term (keeping in mind that it may depend on n). We will assume here that k is non-negative (otherwise replace k by $|k|$ in all occurrences below). For *merge-sort*, we would have had $n' \leq \lceil \frac{n}{2} \rceil \leq \frac{n}{2} + \frac{1}{2}$, so $k = \frac{1}{2}$. We deal with this additive term ‘ $+k$ ’ by inserting suitably placed ‘ $-k$ ’ in the upper bound that we will prove on $T(n)$.
- On the other hand, n' is allowed to be much smaller than $\frac{n}{2} + k$. (For *merge-sort* it would not be much smaller, but we will later see some algorithms where we sometimes recurse in a sub-array that is much smaller than what the bound allows.) It is tempting to think that we could simply replace $T(n')$ by $T(\frac{n}{2} + k)$, i.e., to think that $T(n)$ is monotone. But

it is not at all obvious that $T(n)$ is monotone! (Plus, we do not know whether $\frac{n}{2} + k$ is an integer, so $T(\cdot)$ may not even be defined on it.)

What we do instead is that we replace $T(n')$ with the upper bound that we know for it by induction. This upper bound is a function that we know (here it will be $n \log n$ with a few corrective terms), so it is monotone and we can then replace n' by its upper bound.

- Function $f(n)$ is in $\Theta(n)$, but it might not have exactly the form $c \cdot n$ for some constant c ; it might hide some lower-order term. We deal with this by taking an upper bound of $c \cdot n$ that we know exists beyond a suitable cutoff point.
- One last subtlety concerns the upper bound that we will prove. We need to make sure that this is actually defined! In particular, our upper bound will involve $\log(n - 2k)$; this is undefined if $n \leq 2k$, and non-positive (hence probably not useful) if $n \leq 2k + 1$.

We deal with this by defining multiple upper bounds on $T(n)$, depending on n .

Proof. We claim that there exists some constant $c \geq 0$ and $N \geq 0$ such that

$$T(n) \leq \begin{cases} c & \text{for } n = 0 \\ c \cdot n & \text{for } 1 \leq n \leq N, \\ c \cdot (n-2k) \cdot \log(n-2k) & \text{for } n \geq N \end{cases}$$

clearly then $T(n) \in O(n \log n)$. We define N and c in multiple steps.

- Recall that $f(n) \in O(n)$, so there exists some $c_1 > 0, n_1$ such that $f(n) \leq c_1 n$ for all $n \geq n_1$.
- Set $n_2 \geq 2$ be so big that $6k + 8 \leq n$ for all $n \geq n_2$; this exists since $k \in o(n)$. This implies that for $n \geq n_2$ we have $\log(\frac{n}{2} - k) \geq \log(k + 4) \geq 2$, and therefore $\log(\frac{n}{2} - k)(\frac{n}{2} - k) \geq n - 2k \geq \frac{n}{2} + 3k + 4 - 2k > \frac{n}{2} + k \geq 1$.
- Set $N = \max\{n_0, n_1, n_2\}$.
- Set $c_2 = \max_{1 \leq n \leq N} \{T(n)/n\}$. This is a positive constant since $T(n)$ is positive and defined on all integers.
- Set $c = \max\{T(0), 2c_1, c_2, c_3\}$.

By the definition of c_2 and c_3 , we immediately know that $T(0) \leq c$, and $T(n) \leq cn$ for $1 \leq n \leq N$. So we only need to prove the bound for $n \geq N$ where (by $N \geq n_0$) the recursive formula applies. Let us first prove a bound on $T(n')$. If $n' \leq N$, then

$$T(n') \leq \max\{c, cn'\} \leq \max\{c, c(\frac{n}{2} + k)\} = c(\frac{n}{2} + k) \leq c(\frac{n}{2} - k) \log(\frac{n}{2} - k)$$

where the last two re-formulations hold since $n \geq n_2$. If on the other hand $n' \geq N$, then

$$T(n') \leq c(n' - 2k) \log(n' - 2k) \leq c(\frac{n}{2} + k - 2k) \log(\frac{n}{2} + k - 2k) = c(\frac{n}{2} - k) \log(\frac{n}{2} - k).$$

So the same upper bound holds in both cases, and similarly $T(n'') \leq c(\frac{n}{2} - k) \log(\frac{n}{2} - k)$. Finally

since $c \geq 2c_1$ and $N \geq n_1$, we have $f(n) \leq \frac{1}{2}cn$. Putting it all together, therefore

$$\begin{aligned} T(n) &\leq T(n') + T(n'') + f(n) \leq 2c \underbrace{\left(\frac{n-2k}{2}\right) \log\left(\frac{n-2k}{2}\right)}_{=\log(n-2k)-1} + \frac{1}{2}cn \\ &= c(n-2k) \log(n-2k) - c \underbrace{(n-2k)}_{\geq \frac{n}{2} \text{ by } n \geq n_2} + \frac{1}{2}cn \leq c(n-2k) \log(n-2k). \end{aligned}$$

901

□

902 1.5 Run-time analysis in special situations

903 [The contents of this section will typically be covered in class when the special situations actually
904 occur; you may want to delay reading this section until when it is needed.]

905 By default ‘analyze the run-time’ means to analyze the worst-case run-time. Sometimes we
906 look at the best-case run-time as well, especially when comparing two algorithms that seem
907 similarly fast. But there are other ways of doing run-time analysis that are more complicated;
908 we list them here.

909 1.5.1 Average-case analysis

We briefly hinted earlier that the fairest way to measure the run-time of an algorithm would be to compute its *average-case run-time*, which is defined as follows:

$$T_{\mathcal{A}}^{\text{avg}}(n) := \text{avg}_{I \in \mathcal{I}_n} T_{\mathcal{A}}(I) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T_{\mathcal{A}}(I).$$

910 where as before \mathcal{I}_n denotes the set of instances of size n . Note that we are silently assuming
911 that $|\mathcal{I}_n|$ is finite, which is frequently not true. But it will (at least for the examples that we
912 show) be feasible to map an infinite set of instance to a finite set of equivalence classes, which
913 is good enough.⁸ It is also silently assumed that all instances occur equally often. Perhaps one
914 should weigh the contribution of each instance by the frequency with which it occurs? There
915 are two problems with this. First, we usually don’t know these frequencies. Second, even if we
916 knew them, it would be extremely difficult to do the analysis.

917 The average-case run-time is usually much harder to compute than the worst-case run-time
918 (where we can simply take upper bounds everywhere) or the best-case run-time (where we can
919 pick an especially nice instance). It also often turns out that asymptotically the average-case run-
920 time is no better than the worst-case run-time. Therefore, we will compute it only occasionally
921 (and only in cases where it is asymptotically better). We give here with two simple examples;
922 more complicated ones will appear in Chapter 3.

⁸One could do average-case analysis directly on an infinite set \mathcal{I}_n , but then one would have to replace $1/|\mathcal{I}_n|$ by a density function measuring how often instances occur. This seems more bother than its worth.

923 **Example:** Consider the following algorithm that tests whether a given array is sorted by
 924 comparing consecutive elements.

Algorithm 1.5: *sortedness-tester*(A, n)

Input : Array A stores n numbers
 1 **for** $i \leftarrow 1$ **to** $n - 1$ **do**
 2 **if** $A[i - 1] > A[i]$ **then** return “ A is not sorted”
 3 return “ A is sorted”

We want to analyze the run-time of this algorithm, but to make our life easier, we let $T(\cdot)$ count the number of key-comparisons; clearly the run-time of *sortedness-tester* is proportional to this. In the worst case, A is sorted and we never break out of the for-loop, so $T^{\text{worst}}(n) \geq n - 1 \in \Omega(n)$. In the best case, we find an incorrectly ordered pair with the first comparison, so $T^{\text{best}}(n) \leq 1 \in O(1)$. (Both bounds are clearly tight.) But what is $T(n)$ in the average-case? Recall that

$$T^{\text{avg}}(n) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T(I)$$

925 where $T(I)$ is the bound (here: the number of key-comparison) for one particular instance I .
 926 This needs that \mathcal{I}_n is finite. But for algorithm *sortedness-tester* we have infinitely many instances
 927 of size n (because there are infinitely many numbers). So we first need to study how to map \mathcal{I}_n
 928 to a finite set that represents instances adequately.

Detour: Sorting permutations and assumptions: Observe that *sortedness-tester* is *comparison-based*: it uses no information about the instance except the outcome of comparisons among the items. In particular, the actual numbers in the array are irrelevant as far as the run-time is concerned, all that matters for the algorithm is their relative order. To see one example, note that *sortedness-tester* (and likewise comparison-based sorting algorithms such as *merge-sort*) would do the exact same set of key-comparisons whether we have array

14, 3, 2, 6, 1, 11, 7

or array

14, 4, 2, 6, 1, 12, 8

929 because the relative order of items in the array is exactly the same. Let us formalize this:

Definition 1.5. Let A be an array of n integers. A sorting permutation of A is a permutation π of $\{0, \dots, n-1\}$ such that

$$A[\pi(0)] \leq A[\pi(1)] \leq \dots \leq A[\pi(n-1)].$$

To give an example, for the above array $A[0..6] = [14, 3, 2, 6, 1, 11, 7]$, the sorting permutation is $\pi = \langle 4, 2, 1, 3, 6, 5, 0 \rangle$ because

$$A[\underbrace{\pi(0)}_4] = 1 < A[\underbrace{\pi(1)}_2] = 2 < A[\underbrace{\pi(2)}_1] = 3 < \cdots < A[\underbrace{\pi(6)}_0] = 14.$$

930 A different way to say the same thing is to view A as a function from $\{0, \dots, n-1\}$ to some
 931 numbers; then π is the sorting permutation of A if $A \circ \pi$ (viewed as an array) is sorted.

932 Note that the sorting permutation is not unique if items in the array repeat. To keep the
 933 average-case analysis simpler, we hence often make the following assumption:

934 All numbers in the input are distinct.

935 For problems relating to sorting, this is not unrealistic, because we can always store arbitrary
 936 tie-breakers with the items. We secondly make the following non-trivial assumption:

937 All permutations are equally frequently the sorting permutation of the input.

938 This assumption is clearly violated in practice. Especially for the Sorting problem, the input
 939 very frequently is “almost sorted”, i.e., we have changed just a few entries in the array (or added
 940 a few new ones) and now want to restore order. In this case, the sorting permutations that are
 941 close to $\{0, \dots, n-1\}$ are a lot more frequent than the sorting permutations that are close to
 942 $\{n-1, \dots, 0\}$. But as discussed earlier, we wouldn’t know how to get the frequencies of sorting
 943 permutations, or to do the analysis if we had them.

944 We will sometimes need the inverse π^{-1} of a sorting permutation π . In the example $A =$
 945 $[14, 3, 2, 6, 1, 11, 7]$, we have $\pi = \langle 4, 2, 1, 3, 6, 5, 0 \rangle$ and therefore $\pi^{-1} = \langle 6, 2, 1, 3, 0, 5, 4 \rangle$. Note
 946 that if we view π^{-1} as an array $[6, 2, 1, 3, 0, 5, 4]$, then its entries have exactly the same relative
 947 order as in A . (This holds since $\pi^{-1} \circ \pi = id$, and therefore π is the sorting permutation of
 948 π^{-1} .) Hence whenever we speak of “an instance that has sorting permutation π ”, we might as
 949 well consider the array π^{-1} as its input.

Back to analyzing *sortedness-tester*: With the concept of sorting permutations in place, we can re-state the average-case number of comparisons for *sortedness-tester* as follows:

$$T^{\text{avg}}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

where Π_n is the set of all permutations of $\{0, \dots, n-1\}$ and $T(\pi)$ is the number of comparisons on instance π^{-1} (equivalently, on all instances that have sorting permutation π). To analyze this, break the set of permutations into subsets based on how many comparisons are needed, and hence define

$$\Pi_n^k = \{\pi \in \Pi_n : \textit{sortedness-tester} \text{ needs exactly } k \text{ comparisons on instance } \pi^{-1}\}.$$

950 We also need the set $\Pi_n^{\geq k}$, which is defined as above except that ‘exactly k ’ is replaced by ‘at
 951 least k ’. Note that $\Pi_n^{\leq k} = \Pi_n^{\geq k} \setminus \Pi_n^{\geq k+1}$. We can now bound $T^{\text{avg}}(n)$ as follows:

$$\begin{aligned} T^{\text{avg}}(n) &= \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi) = \frac{1}{n!} \sum_{k=1}^{n-1} \sum_{\pi \in \Pi_n^{\leq k}} k = \frac{1}{n!} \sum_{k=1}^{n-1} k \cdot |\Pi_n^{\leq k}| = \frac{1}{n!} \sum_{k=1}^{n-1} k (|\Pi_n^{\geq k}| - |\Pi_n^{\geq k+1}|) \\ &= \frac{1}{n!} \sum_{k=1}^{n-1} |\Pi_n^{\geq k}| (k - (k-1)) + 0 \cdot |\Pi_n^{\geq 0}| - (n-1) \underbrace{|\Pi_n^{\geq n}|}_0 \quad (\text{telescoping sum}) \\ &= \frac{1}{n!} \sum_{k=1}^{n-1} |\Pi_n^{\geq k}| \end{aligned}$$

952 **Claim 1.1.** $|\Pi_n^{\geq k}| = \binom{n}{k} (n-k)! = \frac{n!}{k!(n-k)!} (n-k)! = \frac{n!}{k!}$.

953 *Proof.* We count the number of permutations $\pi \in \Pi_n^{\geq k}$ by considering the structure of instance
 954 π^{-1} . Since we need at least k comparisons, the first $k-1$ comparisons found items in cor-
 955 rect order, or in other words, $\pi^{-1}[0] \leq \pi^{-1}[1] \leq \dots \pi^{-1}[k-1]$. The numbers that appear in
 956 $\pi^{-1}[0 \dots k-1]$ can be arbitrarily chosen from $\{0, \dots, n-1\}$, so there are $\binom{n}{k}$ choices for them.
 957 The rest of π^{-1} consists of the remaining $n-k$ numbers, and they can be in any of the $(n-k)!$
 958 possible orders. Thus there are $\binom{n}{k} (n-k)!$ many choices for the inverse of π and the bound
 959 follows. \square

With this we can bound $T^{\text{avg}}(n)$:

$$T^{\text{avg}}(n) = \frac{1}{n!} \sum_{k=1}^{n-1} |\Pi_n^{\geq k}| = \frac{1}{n!} \sum_{k=1}^{n-1} \frac{n!}{k!} = \sum_{k=1}^{n-1} \frac{1}{k!}.$$

960 Calculus-lovers will recognize this as the Taylor-expansion of $e^1 - 1 \approx 1.718$. Alternatively,
 961 observe that $k! = k \cdot (k-1) \cdot \dots \cdot 2 \cdot 1 \geq 2^{k-1}$, so $\sum_{k=1}^{n-1} \frac{1}{k!} \leq \sum_{k=1}^{n-1} (\frac{1}{2})^{k-1} < 2$. Either way, on
 962 average *sortedness-tester* uses fewer than two comparisons, and its average-case run-time is $O(1)$.

963 **Example:** We will do a second example, which deals with a recursive algorithm and so we will
 964 need a recursive formula for $T^{\text{avg}}(n)$. Consider the algorithm *avgCaseDemo* in Algorithm 1.6,
 965 which has no particular purpose other than being an interesting example for average-case anal-
 966 ysis.

967 In this algorithm, we compare the two numbers in the array, and depending on which one
 968 is smaller, either cut the size of the array in half or reduce it by 2. Clearly the worst-case run-
 969 time of *avgCaseDemo* is $O(n)$, because we use constant time in each recursion and have $O(n)$
 970 recursions since the size always gets smaller. This is tight, for example on a reversely sorted
 971 array we would always be in the bad case and so reduce the size by only two each time.

972 On the other hand, the best-case run-time is $O(\log n)$. To see this, consider the instance of
 973 a sorted array. Here we always are in the good case, and cut the size in half (or even less, due to

Algorithm 1.6: *avgCaseDemo*(A, n)

Input : Array A stores n distinct numbers

```

1 if  $n \leq 2$  then return
2 if  $A[n-2] \leq A[n-1]$  then
3   | avgCaseDemo( $A[0..\lfloor n/2 \rfloor - 1], \lfloor n/2 \rfloor$ )           // good case
4 else
5   | avgCaseDemo( $A[0..n-3], n-2$ )           // bad case
```

974 rounding) with every recursion. So we have at most $\log n$ recursions, and the best-case run-time
 975 is $O(\log n)$. This is tight since after i recursions the array has size at least $n/2^i - 1$, so we need
 976 $\Omega(\log n)$ recursions to reach the exit-condition. But what is the average-case run-time?

977 **Theorem 1.1.** *The average-case run-time of *avgCaseDemo* is in $O(\log n)$.*

978 *Proof.* To avoid having to use constants, we let $T(\cdot)$ count the number of recursions; clearly its
 979 run-time is proportional to this. There are infinitely many instances, but since *avgCaseDemo*
 980 is comparison-based and all numbers are distinct, we can again switch to sorting permutations.
 981 So for $\pi \in \Pi_n$ define $T(\pi)$ to be the number of recursions done if the input-array has sorting
 982 permutation π (or equivalently, if the input-array is $A := \pi^{-1}$). Note that (assuming $n \geq 3$) we
 983 have two kinds of permutations:

- 984 • We call a permutation *good* if $A[n-2] \leq A[n-1]$ and write Π_n^{good} for the set of good
 985 permutations. In this case, we recurse on $A[0 \dots \lfloor \frac{n}{2} \rfloor - 1]$, which has size $\lfloor n/2 \rfloor$.
- 986 • We call a permutation *bad* if $A[n-2] > A[n-1]$, and write Π_n^{bad} for the bad permutations.
 987 In this case, we recurse on $A[0 \dots n-3]$, which has size $n-2$.

It would be tempting to conclude that therefore $T^{\text{avg}}(n) \leq 1 + T^{\text{avg}}(\lfloor n/2 \rfloor)$ if the permutation
 is good, because we do one more recursion on an array of size $\lfloor n/2 \rfloor$. This is unfortunately *not*
 correct. First of all, there is no permutation mentioned on the left side, so ‘if the permutation is
 good’ is meaningless. (To fix this, the left-hand-side should be $T(\pi)$ for some permutation π .)
 Second, writing ‘ T^{avg} ’ on the right-hand side is incorrect. Instance A gives rise to a sub-array
 of size $n/2$ (namely, $A' = A[0 \dots \lfloor \frac{n}{2} \rfloor - 1]$). But who says that the instance A' is *average*? We
 could certainly create a good permutation where the resulting sub-array A' hits the worst-case.
 So the above formula is not correct as stated; the only thing we know is

$$T(\pi) \leq 1 + T(A[0 \dots \lfloor \frac{n}{2} \rfloor - 1]) \quad \text{if } \pi \text{ is good}$$

988 where as before $A = \pi^{-1}$. However, if we sum up over *all* good permutations, then we can use
 989 T^{avg} on the right-hand side.

Lemma 1.6. *If $n \geq 3$, then*

$$\sum_{\pi \in \Pi_n^{\text{good}}} T(\pi) \leq |\Pi_n^{\text{good}}| \left(1 + T^{\text{avg}}(\lfloor n/2 \rfloor)\right) \text{ and } \sum_{\pi \in \Pi_n^{\text{bad}}} T(\pi) \leq |\Pi_n^{\text{bad}}| \left(1 + T^{\text{avg}}(n-2)\right).$$

990 The proof of this lemma is surprisingly complicated and therefore left to the enriched section
 991 (see below). Observe that $|\Pi_n^{\text{good}}| = |\Pi_n^{\text{bad}}|$, because any bad permutation π can be mapped to a
 992 good permutation π' by exchanging $A[n-2]$ and $A[n-1]$, and any good permutation is obtained
 993 this way exactly once. Therefore $|\Pi_n^{\text{good}}| = \frac{1}{2}n! = |\Pi_n^{\text{bad}}|$. With this bound, we now get

$$\begin{aligned}
 T^{\text{avg}}(n) &\leq \frac{1}{|\Pi_n|} \sum_{I \in \Pi_n} T(I) \leq \frac{1}{|\Pi_n|} \left(\sum_{I \in \Pi_n^{\text{good}}} T(I) + \sum_{I \in \Pi_n^{\text{bad}}} T(I) \right) \\
 &\leq \frac{1}{|\Pi_n|} \left(|\Pi_n^{\text{good}}| (1 + T^{\text{avg}}(\lfloor n/2 \rfloor)) + |\Pi_n^{\text{bad}}| (1 + T^{\text{avg}}(n-2)) \right) \\
 &= 1 + \frac{1}{2} T^{\text{avg}}(\lfloor n/2 \rfloor) + \frac{1}{2} T^{\text{avg}}(n-2)
 \end{aligned} \tag{1.1}$$

994 **Getting an idea of the bound:** If some birdie told us now that $T^{\text{avg}}(n)$ should be at most
 995 $2 \log n$, then it would not be difficult to prove by induction that indeed $T^{\text{avg}}(n) \leq 2 \log n$. (See
 996 the next paragraph.) But it may be instructive to see how one could come up with such a bound.
 997 For doing so, we will make an assumption and ignore an issue, because this paragraph here is
 998 only ‘rough work’ while the next one is precise.

999 Let us assume that $T^{\text{avg}}(n-2) \leq T^{\text{avg}}(n)$. (Run-times usually do not get smaller as the
 1000 instance gets bigger, so this assumption does not seem too far-fetched. However, we cannot rely
 1001 on run-times being monotone, especially when taking the average.) Then

$$T^{\text{avg}}(n) \leq 1 + \frac{1}{2} T^{\text{avg}}(\lfloor n/2 \rfloor) + \frac{1}{2} T^{\text{avg}}(n-2) \stackrel{\text{under assumption}}{\leq} 1 + \frac{1}{2} T^{\text{avg}}(\lfloor n/2 \rfloor) + \frac{1}{2} T^{\text{avg}}(n).$$

1002 This formula should make you nervous, because $T^{\text{avg}}(n)$ appears on both sides, so this is
 1003 not a proper recursive formula. Ignoring this issue, we can rephrase the formula as $\frac{1}{2} T^{\text{avg}}(n) \leq$
 1004 $1 + \frac{1}{2} T^{\text{avg}}(\lfloor n/2 \rfloor)$ or $T^{\text{avg}}(n) \leq 2 + T^{\text{avg}}(\lfloor n/2 \rfloor)$. This fits one of the recursive formulas that we
 1005 saw in Table 8 and resolves to $O(\log n)$. Specifically, expanding a few terms, one easily sees that
 1006 $T^{\text{avg}}(n) \leq 2 \log n$ under our assumption. (But remember that this was only rough work, so we
 1007 are not yet done.)

1008 **Actual proof:** We will prove that $T^{\text{avg}}(n) \leq 2 \log n$ by induction on n . This clearly holds
 1009 for $n \leq 2$, since we use no recursion (and $\log n \geq 0$). So assume $n \geq 3$. Our above analysis of
 1010 $T^{\text{avg}}(n)$ made no assumptions up until Equation (1.1), and we continue from there. Applying
 1011 the inductive hypothesis, we have

$$\begin{aligned}
 T^{\text{avg}}(n) &\leq 1 + \frac{1}{2} T^{\text{avg}}(\lfloor n/2 \rfloor) + \frac{1}{2} T^{\text{avg}}(n-2) \\
 &\leq 1 + \frac{1}{2} (2 \underbrace{\log(\lfloor n/2 \rfloor)}_{\leq \log(n/2) = \log n - 1}) + \frac{1}{2} (2 \underbrace{\log(n-2)}_{\leq \log n}) \leq 1 + (\log n - 1) + \log n = 2 \log n
 \end{aligned}$$

1012

□

Note that the assumption from the previous paragraph is no longer needed, because we replaced $T^{\text{avg}}(n-2)$ by $2 \log(n-2)$, and $\log n$ is monotone.

With this, the average-case run-time of *avgCaseDemo* is $O(\log n)$ (presuming you believe Lemma 1.6 whose proof will be given below). This is tight because already the best-case run-time was $\Omega(\log n)$, and the avg-case run-time cannot be smaller.

The proof of Lemma 1.6: (cs240e) We will only prove the claim for good permutations; the other claim is similar (replace ‘good’ by ‘bad’ and ‘ $\lfloor n/2 \rfloor$ ’ by ‘ $n-2$ ’ everywhere). For any good permutation π , define π_{half} to be the permutation for the recursion, i.e., π_{half} is the sorting permutation of $\pi^{-1}[0 \dots \lfloor \frac{n}{2} \rfloor - 1]$ and $T(\pi) = 1 + T(\pi_{\text{half}})$. (In terms of our example $\pi = \{4, 2, 1, 3, 6, 5, 0\}$, we had $\pi^{-1} = \{6, 2, 1, 3, 0, 5, 4\}$, so the first $\lfloor n/2 \rfloor = 3$ numbers are $\{6, 2, 1\}$, of which the sorting permutation is $\pi_{\text{half}} = \{2, 1, 0\}$.) Note that π_{half} is a sorting permutation of $\lfloor n/2 \rfloor$ numbers, hence belongs to $\Pi_{\lfloor n/2 \rfloor}$.

The crucial ingredient for the proof is now to distinguish which good permutations π have one particular permutation of $\Pi_{\lfloor n/2 \rfloor}$ as π_{half} .

$$\begin{aligned} \sum_{\pi \in \Pi_n^{\text{good}}} T(\pi) &= \sum_{\pi \in \Pi_n^{\text{good}}} (1 + T(\pi_{\text{half}})) = |\Pi_n^{\text{good}}| + \sum_{\pi \in \Pi_n^{\text{good}}} T(\pi_{\text{half}}) \\ &= |\Pi_n^{\text{good}}| + \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} \left| \left\{ \pi \in \Pi_n^{\text{good}} \text{ for which } \pi_{\text{half}} = \pi' \right\} \right| \cdot T(\pi'). \end{aligned}$$

Define $\Pi_n^{\text{good}}(\pi')$ (for a given permutation $\pi' \in \Pi_{\lfloor n/2 \rfloor}$) to be the set that we needed above, i.e., the good permutations π for which $\pi_{\text{half}} = \pi'$.

Claim 1.2. *If $n \geq 3$, then $|\Pi_n^{\text{good}}(\pi')| = \frac{n!}{2(\lfloor n/2 \rfloor)!}$ for all $\pi' \in \Pi_{\lfloor n/2 \rfloor}$.*

Proof. Fix one permutation π in $\Pi_n^{\text{good}}(\pi')$. We can analyze the structure of the corresponding array $A := \pi^{-1}$, and therefore count the number of possibilities for π^{-1} (and hence π).

- The first $\lfloor n/2 \rfloor$ items of A must have sorting permutation π' by definition of $\Pi_n^{\text{good}}(\pi')$. As such, they can be any $\lfloor n/2 \rfloor$ distinct numbers of $\{0, \dots, n-1\}$, but the order of these numbers is fixed. We hence have $\binom{n}{\lfloor n/2 \rfloor}$ choices here.
- The next $n - \lfloor n/2 \rfloor - 2 = \lceil n/2 \rceil - 2$ numbers of A can be chosen arbitrarily among the remaining numbers in $\{0, \dots, n-1\}$. Furthermore, they can be ordered arbitrarily. Hence there are $\binom{\lceil n/2 \rceil}{\lceil n/2 \rceil - 2} (\lceil n/2 \rceil - 2)!$ many choices here.
- The last two numbers of A are the two remaining numbers of $\{0, \dots, n-1\}$. (Here we use $n \geq 3$: the last two numbers do not overlap the first $\lfloor n/2 \rfloor$ numbers.) The order among them is determined because we must have $A[n-2] < A[n-1]$ since π is good. So there are no further choices here.

Summarizing, the number of choices for π^{-1} is

$$\binom{n}{\lfloor n/2 \rfloor} \cdot \binom{\lceil n/2 \rceil}{\lceil n/2 \rceil - 2} (\lceil n/2 \rceil - 2)! = \frac{n!}{\lfloor n/2 \rfloor! \cdot \lceil n/2 \rceil!} \cdot \frac{\lceil n/2 \rceil!}{(\lceil n/2 \rceil - 2)! \cdot 2!} (\lceil n/2 \rceil - 2)! = \frac{n!}{\lfloor n/2 \rfloor! \cdot 2}.$$

1042

□

1043 Since $|\Pi_n^{\text{good}}| = |\Pi_n|/2 = n!/2$, we can rephrase this as $|\Pi_n^{\text{good}}(\pi')| = |\Pi_n^{\text{good}}|/|\Pi_{\lfloor n/2 \rfloor}|$ and
 1044 then obtain the desired result:

$$\begin{aligned} \sum_{\pi \in \Pi_n^{\text{good}}} T(\pi) &= |\Pi_n^{\text{good}}| + \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} |\Pi_n^{\text{good}}(\pi')| \cdot T(\pi') \\ &= |\Pi_n^{\text{good}}| + \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} \frac{|\Pi_n^{\text{good}}|}{|\Pi_{\lfloor n/2 \rfloor}|} \cdot T(\pi') = |\Pi_n^{\text{good}}| \left(1 + \underbrace{\frac{1}{|\Pi_{\lfloor n/2 \rfloor}|} \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} T(\pi')}_{T_{\text{avg}}(\lfloor n/2 \rfloor)} \right). \end{aligned}$$

1045 1.5.2 Randomized algorithms

1046 A *randomized algorithm* is an algorithm that uses not only the input but also some random
 1047 numbers to determine the output. There are actually two different kinds of randomized algo-
 1048 rithms; one kind (“Monte Carlo algorithms”) are permitted to give incorrect output but should
 1049 always be fast, whereas the other kind (“Las Vegas algorithms”) guarantee that the output will
 1050 be correct, but it may not always be found quickly. In this course we will *only* study Las Vegas
 1051 algorithms, i.e., we only consider algorithms that always give the correct answer. (Take cs466 if
 1052 you are curious about Monte Carlo algorithms.)

1053 However, computers cannot actually generate random numbers. (Or at least they shouldn’t....
 1054 though some randomness in the form of electrical interference with the bits in a computer can
 1055 happen.) For what it’s worth, humans can’t really generate random numbers either: the out-
 1056 come of a coin toss is completely determined by the physics of how and from what height the
 1057 coin is tossed and the wind conditions. However, this dependence is so complex and chaotic
 1058 that for all practical purposes a coin toss acts like a random 0/1-variable. In the same spirit,
 1059 computers have built-in programs called *pseudo-random number generators* (PRNG). This is a
 1060 deterministic program that outputs a sequence of numbers that seemingly are randomly. From
 1061 this, we can determine a random bit, or a random number in a given range, etc.

1062 The pseudo-random number generator is initialized with a value called a *seed* (using the same
 1063 seed results in the same sequence of numbers); the usual approach to get random numbers is to
 1064 use as seed the current time so that different runs of the program result in different outputs.
 1065 How closely the generated sequence really resembles a random sequence of numbers depends
 1066 very much on the quality of the pseudo-random number generator; many different options have
 1067 been studied, but this is a topic beyond the scope of cs240.

1068 For our pseudo-codes, it will suffice to assume the existence of a function *random*(*n*) that
 1069 returns in constant time an integer in $\{0, \dots, n-1\}$, and any of these *n* integers is equally likely.
 1070 In particular, doing a comparison ‘if *random*(2) = 1’ is also called a *coin toss*, because just like
 1071 with a coin toss there are two possible outcomes, and each has probability $\frac{1}{2}$. We will *only* be
 1072 using random variables that follow a discrete uniform distribution. For simplicity, we will usually
 1073 drop ‘discrete uniform’ and frequently also the universe from which we are choosing, since this

should be clear from context. For example, when we say “pick a random permutation”, we really mean “pick a permutation from the set of all permutations, following a uniform distribution”.

Randomized algorithms need their own model for how to measure the run-time, because it now depends not only on the input but also on the random outcomes. Define $T_{\mathcal{A}}(I, R)$ to be the running time of a randomized algorithm \mathcal{A} for an instance I if we fix the outcomes R of the calls to *random()* that are done during the execution of \mathcal{A} on instance I . (We will drop \mathcal{A} when it should be clear from context.) The *expected running time* $T^{\text{exp}}(I)$ for instance I is then the expected value for $T(I, R)$:

$$T_{\mathcal{A}}^{\text{exp}}(I) = E[T_{\mathcal{A}}(I, R)] = \sum_R T_{\mathcal{A}}(I, R) \cdot P(R).$$

Recall that for the (non-randomized) run-time analysis, we now defined different terms, depending on whether we considered the worst-possible instance I (among all instances of size n), or the best, or the average. In the same spirit we could now define three different versions of expected running time $T^{\text{exp}}(n)$, called *worst-instance*, *best-instance* and *average-instance* expected run-time. For example, the worst-instance expected running time would be

$$T_{\mathcal{A}}^{\text{exp}}(n) := \max_{I \in \mathcal{I}_n} T_{\mathcal{A}}^{\text{exp}}(I),$$

and unless specified otherwise, *expected run-time* refers to this worst-instance expected running time. For any well-designed randomized algorithm (and in particular all examples you will see in these notes), all instances of size n have the same expected running-time. (This is exactly the point of randomization: it should make all instances equally good.) Therefore the running time should be the same regardless of whether we take the worst, best, or average instance.

The run-time depends not only on the instance, but also on the random outcomes. Here we took the expected value, or in other words, the weighted average over all possible outcomes. Occasionally we may be interested in what one could call the *worst-luck* or *best-luck* running time. For example, the *worst-instance worst-luck running time* is $\max_{I \in \mathcal{I}_n} \max_R T(I, R)$, where the second maximum is over all possible sequences of random outcomes. This bound tells us the very worst that could happen. However, if we have set up a suitable randomization then the likelihood of this happening should be so small that we will not worry overmuch if this bound is large.

To summarize, the idea of a randomized algorithm is to shift the dependency of the run-time from what we can't control (the input) to what we *can* control (the random numbers).

No more bad instances, just unlucky random draws.

Example: Let us show how to analyze the expected run-time on the algorithm *expectedDemo* (Algorithm 1.7), which much resembles *avgCaseDemo* (Algorithm 1.6) and which again has no particular purpose other than being an interesting example for analyzing the expected run-time.

The main difference between *expectedDemo* and *avgCaseDemo* is that we use a coin toss to decide whether to swap $A[n-2]$ and $A[n-1]$. Hence for $n \geq 3$ it depends on the coin toss

Algorithm 1.7: *expectedDemo*($A, n \leftarrow A.size$)

Input : Array A stores n distinct numbers

```

1 if  $n \leq 2$  then return
2 if  $random(2) = 1$  then swap  $A[n-2]$  and  $A[n-1]$ 
3 if  $A[n-2] \leq A[n-1]$  then
4   | expectedDemo( $A[0.. \lfloor n/2 \rfloor]$ )                                // Good case
5 else
6   | expectedDemo( $A[0..n-3]$ )                                // Bad case
```

1097 whether we go into the good case (where we recurse on an array of size $\lfloor n/2 \rfloor$) or the bad case
1098 (where we recurse on an array of size $n-2$). Let X be the random variable that describes
1099 this outcome, i.e., $X = \text{'good'}$ means that the first coin toss led to the good case and $X =$
1100 'bad' otherwise. Note that $P(X = \text{'good'}) = \frac{1}{2}$, because we know $A[n-2] \neq A[n-1]$ by the
1101 precondition and with probability $\frac{1}{2}$ we rearrange the array such that we end in the good case.

For an instance A of size n and a sequence R of random outcomes, we let $T(A, R)$ be the number of recursions executed on instance A if these outcomes happen. (The actual run-time is proportional to this). We can write $R = \langle x, R' \rangle$, where $x \in \{\text{'good'}, \text{'bad'}\}$ is the outcome for X while R' is the sequence of outcomes in the recursions. Then

$$T(A, R) = T(A, \langle x, R' \rangle) = \begin{cases} 1 + T(A[0 \dots \lfloor \frac{n}{2} \rfloor - 1], R') & \text{if } x \text{ is 'good'} \\ 1 + T(A[0 \dots n-2], R') & \text{if } x \text{ is 'bad'} \end{cases}.$$

1102 This loosely suggests a formula such as $T(A, R) \leq 1 + T^{\text{exp}}(\lfloor n/2 \rfloor)$ if x is good, but again we
1103 have to be careful because we do not know whether R' leads us to the expected case. But this
1104 time it is much easier⁹ to argue a similar formula if we sum over all possible outcomes:

Lemma 1.7. *If $n \geq 3$, then*

$$\sum_R P(R) T(A, R) \leq 1 + \frac{1}{2} T^{\text{exp}}(\lfloor n/2 \rfloor) + \frac{1}{2} T^{\text{exp}}(n-2).$$

1105 *Proof.* Write again $R = \langle x, R' \rangle$. Because random variable X is independent of later random
1106 choices, we have $P(\langle x, R' \rangle) = P(X = x) \cdot P(R')$. This allows us to split the sum as follows:

⁹Why is it so much easier, compared to Lemma 1.6? The main reason is that the expected run-time uses the *maximum* over all instances, so we do not have to worry what the particular sub-arrays created by instance A are. It also helps that we no longer need to use sorting permutations.

$$\begin{aligned}
\sum_R P(R)T(A, R) &= \sum_{\langle x, R' \rangle} P(\langle x, R' \rangle)T(A, \langle x, R' \rangle) = \sum_x \sum_{R'} P(X = x)P(R')T(A, \langle x, R' \rangle) \\
&= P(X \text{ good}) \sum_{R'} P(R')T(A, \langle \text{good}, R' \rangle) + P(X \text{ bad}) \sum_{R'} P(R')T(A, \langle \text{bad}, R' \rangle) \\
&= \frac{1}{2} \sum_{R'} P(R') \left(1 + T(A[0 \dots \lfloor \frac{n}{2} \rfloor - 1], R') \right) + \frac{1}{2} \sum_{R'} P(R') \left(1 + T(A[0 \dots n-2], R') \right) \\
&= 1 + \frac{1}{2} \sum_{R'} P(R') \cdot T(A[0 \dots \lfloor \frac{n}{2} \rfloor - 1], R') + \frac{1}{2} \sum_{R'} P(R') \cdot T(A[0 \dots n-2], R') \\
&\leq 1 + \frac{1}{2} \underbrace{\max_{A' \in \mathcal{I}_{n/2}} \sum_{R'} P(R') \cdot T(A', R')}_{T^{\text{exp}}(\lfloor n/2 \rfloor)} + \frac{1}{2} \underbrace{\max_{A' \in \mathcal{I}_{n-2}} \sum_{R'} P(R') \cdot T(A', R')}_{T^{\text{exp}}(n-2)}
\end{aligned}$$

1107

□

1108 Note in particular that the upper bound is independent of instance A ! (This is exactly the
 1109 point of a well-designed randomization—all instances should have the same expected run-time.)
 1110 Therefore

$$T^{\text{exp}}(n) = \max_{I \in \mathcal{I}_n} \sum_R P(R) \cdot T(I, R) \leq 1 + \frac{1}{2} T^{\text{exp}}(\lfloor n/2 \rfloor) + \frac{1}{2} T^{\text{exp}}(n-2).$$

1111 This is the same recursive formula as we had for *avgCaseDemo* in Section 1.5, and the same
 1112 inductive proof shows that $T^{\text{exp}}(n) \leq 2 \log n$. Thus the expected run-time of *expectedDemo* is
 1113 $O(\log n)$.

1114 **Expected run-time vs. average-case run-time.** In our example, the expected run-time
 1115 and the average-case run-time ended up with exactly the same recursion. It is therefore tempt-
 1116 ing to think that in order to analyze the average-case run-time, we could simply randomize
 1117 the algorithm, and analyze the expected run-time instead. In general, this is *false*. More
 1118 specifically, whether there is a relationship between the average-case run-time and the expected
 1119 run-time of some randomization depends *entirely* on how the randomization was done! *If* the
 1120 problem instances satisfy some properties, and *if* the randomization is done carefully, then we
 1121 can indeed obtain bounds on the average-case run-time via randomization. (We will see this in
 1122 Section 3.1.2.) But in general the average-case run-time and the expected run-time are two very
 1123 different things.

1124 1.5.3 Amortized analysis and potential functions (cs240e)

1125 You should have seen *dynamic arrays* in some predecessor course: These store items exactly as
 1126 in an array, but also keep track of the capacity of the array, and double this capacity whenever

the array is full. One can easily argue that on average, the time to do an insertion in a dynamic array is constant. More precisely, occasionally we must copy the entire array to a new (bigger) array, which takes $\Theta(n)$ time. But every time this happens, we previously must have had $n/2$ insertions that were fast, i.e., took $\Theta(1)$ time. Summing up, therefore we are using $\Theta(n)$ time for $\Theta(n)$ operations, and so on average they take $\Theta(1)$ time.

This is a prime example of *amortized analysis*, where we permit some executions of an operation to be quite slow while most other executions are fast.¹⁰ For some examples (such as dynamic arrays) a simple argument suffices: sum up how many cheap and expensive operations we had and divide by the number of operations. But for some examples to come, more powerful methods are needed.

Let us first define formally what we mean by amortized run-time. For this and later definitions, we assume that some data structure that supports some operations has been fixed.

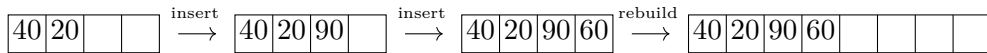
Definition 1.6. Let $T^{\text{actual}}(\mathcal{O})$ be the run-time of operation \mathcal{O} . Let $T^{\text{amort}}(\cdot)$ be a function on the operations. We say that $T^{\text{amort}}(\cdot)$ upper-bounds the amortized run-time if for any sequence $\mathcal{O}_1, \dots, \mathcal{O}_k$ of operations that could occur we have

$$\sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) \leq \sum_{i=1}^k T^{\text{amort}}(\mathcal{O}_i).$$

We usually just say “ \mathcal{O} has amortized run-time $T^{\text{amort}}(\mathcal{O})$ ”, ignoring the fact that $T^{\text{amort}}(\cdot)$ could be bigger than needed.

Note the difference between average-case analysis and amortized analysis: In average-case analysis, we take the average over all *instances*; typically this is used for algorithms where we are given one instance in which we search for a solution. In contrast, in amortized analysis we take the average over all *operations*; typically this is used for data structures where we repeatedly apply some operation, and sometimes the operation takes a long time while usually the operation is fast.

Example: Dynamic arrays Let us assume that we only do insertions (having deletions can only make life easier). Let us also only look at a subsequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$, from when we had just finished rebuilding to the next rebuild. (The argument then holds for all sequences, because they can be broken into such subsequences.) For ease of description, we treat *insert* (without rebuilding) and *rebuild* here as separate operations, though the latter is only triggered by the former.



The actual run-time for *insert* is $\Theta(1)$, and for *rebuild* it is $\Theta(n)$, where n is the current number of items in the array.

¹⁰Amortized analysis can also be applied to space usage, but we will see here only examples of amortized run-time.

The definition of amortized time implies that we will need to do arithmetic with these run-time bounds. We should not intermix arithmetic and asymptotic notations! For this reason, we define a *time unit* such that “one unit of time” hides the constant in the asymptotic notation. For dynamic arrays, we choose time units such that *insert* takes at most one time unit and *rebuild* takes at most n time units.

Now define $T^{\text{amort}}(\text{insert})$ to be 3 time units, and $T^{\text{amort}}(\text{rebuild})$ to be 0 time units.¹¹ We claim that this is indeed an amortized run-time bound. Observe that *rebuild* (with n items) is preceded by $n/2$ *insert* operations that filled the free slots. Summing over these $k := n/2 + 1$ operations $\mathcal{O}_1, \dots, \mathcal{O}_k$, we have

$$\begin{aligned} \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) &= \frac{n}{2} T^{\text{actual}}(\text{insert}) + 1 \cdot T^{\text{actual}}(\text{rebuild}) \leq \frac{n}{2} \cdot 1 + 1 \cdot n = \frac{3}{2}n \\ &= \frac{n}{2} \cdot 3 + 0 = \frac{n}{2} T^{\text{amort}}(\text{insert}) + 1 \cdot T^{\text{amort}}(\text{rebuild}) = \sum_{i=1}^k T^{\text{amort}}(\mathcal{O}_i). \end{aligned}$$

Unfortunately, in general doing amortized analysis is not as easy as in the above example. We therefore now introduce a powerful method that makes amortized analysis easier.

Definition 1.7. A potential function is a function $\Phi(\cdot)$ that depends on the status of the data structure and satisfies the following for any sequence $\mathcal{O}_1, \dots, \mathcal{O}_k$ of operations that could occur:

- $\Phi(i) \geq 0$ for all $i \geq 0$, where $\Phi(i)$ denotes the value after $\mathcal{O}_1, \dots, \mathcal{O}_i$ have been executed,
- $\Phi(0) = 0$.

The above definition is very vague; there are many ways of defining a potential function. But a potential function defines in a natural way an amortized run-time bound, and we are interested in those potential functions where this amortized bound is small. To state this precisely, the following notation is helpful. Given an operation \mathcal{O} (which is part of a sequence of operations), we write Φ_{before} and Φ_{after} for the state of the potential function before and after this operation.

Lemma 1.8. For any potential function $\Phi(\cdot)$, the function

$$T^{\text{amort}}(\mathcal{O}) := \underbrace{T^{\text{actual}}(\mathcal{O})}_{\text{actual run-time}} + \underbrace{\Phi_{\text{after}} - \Phi_{\text{before}}}_{\text{potential-difference } \Delta\Phi}$$

upper-bounds the amortized run-time.

Proof. Fix an arbitrary sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$. Summing up the amortized times

¹¹You may be puzzled at the idea that some operation could take 0 time. This is not a contradiction, because we are amortizing over operations; the actual run-time is not 0, but previous operations “overpaid” enough that this operation does not need to add anything.

1178 and using a telescoping sum we get:

$$\begin{aligned}
 \sum_{i=1}^k T^{\text{amort}}(\mathcal{O}_i) &= \sum_{i=1}^k \left(T^{\text{actual}}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1) \right) \\
 &= \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) + \sum_{i=1}^k \left(\Phi(i) - \Phi(i-1) \right) \\
 &= \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) + \underbrace{\Phi(k)}_{\geq 0} - \underbrace{\Phi(0)}_{=0} \geq \sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i)
 \end{aligned}$$

□

Dynamic arrays again: Define time units as before, and define a potential function $\Phi(\cdot)$ as follows:

$$\Phi(i) = \max\{0, 2 \cdot \text{size} - \text{capacity}\},$$

1180 where as usual *size* denotes the number of items in the array, while *capacity* is the length of the
 1181 array. Technically we should write *size*(*i*) and *capacity*(*i*) to indicate that these also change with
 1182 each operation. Usually the ‘(*i*)’ should be clear from context, and so will be omitted.

Clearly $\Phi(i) \geq 0$. Also initially *size* = 0 and *capacity* ≥ 0 , so $\Phi(0) = 0$ as desired. Now the amortized run-time for *insert* is

$$T^{\text{amort}}(\text{insert}) = T^{\text{actual}}(\text{insert}) + \Phi_{\text{after}} - \Phi_{\text{before}} \leq 3,$$

because the actual time is at most one unit, the size increases by one and the capacity does not change. The amortized run-time for *rebuild* is

$$T^{\text{amort}}(\text{rebuild}) = T^{\text{actual}}(\text{rebuild}) + \Phi_{\text{after}} - \Phi_{\text{before}} \leq n + (0 - n) = 0.$$

1183 Namely, the actual time is at most $n = \text{size}$ time units. Since we are rebuilding, the size equaled
 1184 the capacity before, so $\Phi_{\text{before}} = 2n - n = n$. After the rebuilt, the capacity is twice the size, so
 1185 $\Phi_{\text{after}} = 2n - 2n = 0$. Note that we get exactly the same amortized run-time bounds as before,
 1186 but the method to obtain them was quite different.

1187 Summarizing, to do amortized analysis using the potential function method consists of the
 1188 following steps:

- 1189 • Define what “one time unit” means. Typically we use that an operation with run-time $\Theta(k)$ takes at most k time units.
- 1190
- 1191 • Define a potential function Φ and verify that $\Phi(0) = 0$ and $\Phi(i) \geq 0$. Coming up with a
 1192 potential function that gives good amortized bounds is an art, we will give a glimpse on
 1193 how to do this below.
- 1194 • For each operation \mathcal{O} , compute $T^{\text{amort}}(\mathcal{O}) := T^{\text{actual}}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}}$, and find an
 1195 asymptotic upper bound for it.

1.6 Take-home messages

Let us summarize the most important insights from this chapter.

- To describe an algorithm, we should state the main idea, give a detailed description and/or pseudo-code, argue correctness, and analyze the run-time and the auxiliary space.
- To compare two algorithms, we compare the worst-case asymptotic run-time, and break ties by other considerations such as auxiliary space and/or best-case run-time. But we need to keep in mind that this is only a rough guidance and in practice the apparently slower algorithm may be faster.
- Asymptotic notation, and especially O, Ω, Θ are useful tools to express bounds on run-time and space without having to deal with constants. Table 1.8 gives a summary of these notations and how they are used.
- Bounds on run-time and space should always be stated as *tight* bounds (using Θ -notation), else the comparison between bounds is meaningless.
- Algorithm *merge-sort* solves the Sorting problem in $\Theta(n \log n)$ worst-case run-time and uses $\Theta(n)$ auxiliary space.
- There are other ways of analyzing algorithms, e.g. average-case run-time and amortized run-time and (for randomized algorithms) expected run-time.

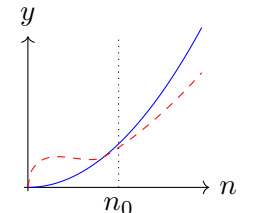
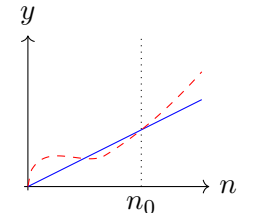
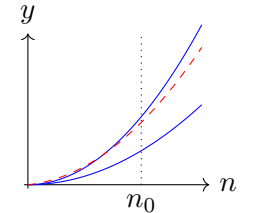
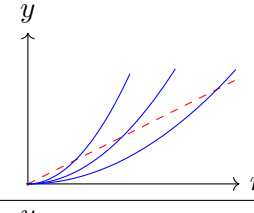
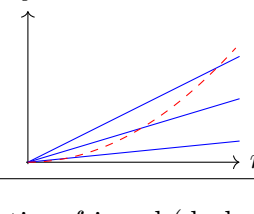
			definition	picture	limit-rule	Typical use
O	big- O	asymptotic upper bound	$\exists c > 0 \quad \exists n_0 \geq 0$ $ f(n) \leq c g(n) $ for $n \geq n_0$		$\lim < \infty$	<i>merge-sort</i> takes $O(n \log n)$ time.
Ω	big-Omega	asymptotic lower bound	$\exists c > 0 \quad \exists n_0 \geq 0$ $ f(n) \geq c g(n) $ for $n \geq n_0$		$\lim > 0$	<i>insertion-sort</i> takes $\Omega(n^2)$ time on some input.
Θ	Theta	asymptotically the same	$\exists c_1, c_2 > 0 \quad \exists n_0 \geq 0$ $c_1 g(n) \leq f(n) \leq c_2 g(n) $ for $n \geq n_0$		$0 < \lim < \infty$	<i>merge-sort</i> and <i>heap-sort</i> have asymptotically the same worst-case run-time.
o	little-o	asymptotically strictly smaller	$\forall c > 0 \quad \exists n_0 \geq 0$ $ f(n) \leq c g(n) $ for $n \geq n_0$		$\lim = 0$	In the worst case, <i>merge-sort</i> is asymptotically faster than <i>insertion-sort</i> .
ω	little-omega	asymptotically strictly bigger	$\forall c > 0 \quad \exists n_0 \geq 0$ $ f(n) \geq c g(n) $ for $n \geq n_0$		$\lim = \infty$	<i>merge-sort</i> uses asymptotically more auxiliary space than <i>insertion-sort</i> .

Table 1.8: Summary-table for asymptotic notation. Function f is red (dashed) while scaled versions of function g are blue (solid).

1.7 Historical remarks

Describing algorithms via pseudocode and arguing why they are correct was done long before computer were invented, consider for example the sieve of Eratosthenes for finding prime numbers (attributed to Eratosthenes, 3rd century BCE) or the program to compute Bernoulli numbers added by Lovelace in her translation of Menabrea's book [Men43]. A good reference for the early history of data structures in Section 2.6 of Knuth's book (volume 1) [Knu97]. The O -notation was invented by Bachmann [Bac94], and formalized and expanded to o -notation by Landau [Lan09], though these authors defined the notations via the limit-rule. It was popularized for CS by Knuth's textbook [Knu97] whose first edition appeared in 1968; he also introduced Ω and Θ . Basic sorting algorithms (such as *insertion-sort* and *merge-sort*) have been discovered repeatedly and independently; see Section 5.5 in volume 3 of Knuth's textbook [Knu98] for a longer history.

Chapter 2

Priority Queues

Contents

2.1	ADT Priority Queue	55
2.1.1	Applications of priority queues	58
2.1.2	Elementary implementations	58
2.2	Non-trivial PQ realization: Binary heaps	60
2.2.1	Binary Tree Review	61
2.2.2	Heap definition	63
2.2.3	Storing heaps	63
2.2.4	Heap operations	64
2.3	Heapsort	67
2.3.1	Making the algorithm in-place	67
2.3.2	Heapify	68
2.4	More PQ operations (cs240e)	71
2.4.1	Meldable heap	72
2.4.2	Binomial heaps	76
2.5	Take-home messages	80
2.6	Historical remarks	81

2.1 ADT Priority Queue

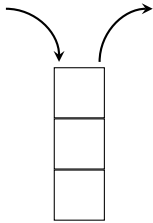
You should have seen the concept of an *abstract data type* (or ADT) in previous CS courses. This is a common tool for software development; the main idea is to hide implementation details from a user that does not need to know them. So in particular, for an ADT we specify

- what is stored and
- what can be done with it

but we do *not* say how the ADT is *realized*, i.e. exactly how the data is stored and how the operations are performed. Switching from one realization to another should not affect at all

1256 what is being done, as far as the outside world is concerned. (It may affect the run-time and/or
 1257 space-usage though.) There are two abstract data types that we will review briefly, and then
 1258 we introduce one that generalizes them both.

1259 **ADT Stack** The abstract data type Stack promises to store items (of any kind) and to support
 1260 the following two operations:



- *push*: Add an item to the stack.
- *pop*: Remove and return the most recently added item. This assumes that the stack is non-empty.

1262 A stack is also described as a LIFO (last-in-first-out) data structure: the item that was
 1263 added last has highest priority for removing. A stack typically also has helper-functions, such
 1264 as *isEmpty* and/or *size*. (One of the two must exist so that we can check the precondition for
 1265 *pop*.) Often we also have a function *top* that returns the most recently added item, but does not
 1266 remove it from the stack.

1267 You should have seen how to implement stacks with lists and with arrays.

1268 **ADT Queue** The abstract data type Queue promises to store items (of any kind) and to
 1269 support the following two operations:



- *enqueue* or *append*: Add an item to the queue
- *dequeue*: Remove and return the item that has been on the queue the longest. This assumes that the queue is non-empty.

1271 A queue is also described as a FIFO (first-in-first-out) data structure: the item that was
 1272 added first has highest priority for removing. Again we typically have helper-functions, such
 1273 as *isEmpty* and/or *size*, and often have a function *peek* that returns the item that would be
 1274 removed next, but does not remove it from the queue.

1275 You should have seen how to implement queues with lists or with arrays that are filled in a
 1276 circular fashion.

1277 **ADT Priority Queue** Both stacks and queues can be viewed as a data structure where
 1278 items are added with some *priority* and the highest-priority item is the next one to be removed.
 1279 (For stacks added items have increasingly higher priority, which for queues added items have
 1280 increasingly smaller priority.) This can be generalized to a data structure where we can give
 1281 an *arbitrary* priority to every item. This is the abstract data type *Priority Queue*. This stores
 1282 items that have two fields: a value and a priority.

- The *priority* is an object that can be compared (here we use integers). It is permitted for different items to have the same priorities, and one could also have negative priorities (though we will avoid this in our examples).
The priority is typically set by the user to achieve the desired order in which items are removed.
- The *value* can be anything. This is the actual information that is of interest to the user (as opposed to the priority, which is usually only needed so that the item is extracted at the correct time).

The priority is sometimes also called a *key* (a concept that we will see extensively in later chapters), so the elements in a priority queue are *key-value-pairs* (or KVPs for short). In our figures below, we represent the whole key-value pair by only showing the key. One should understand this key to be the placeholder for a reference to some object that stores the actual key and the value as fields.

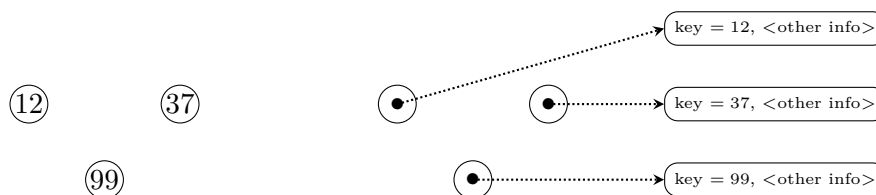


Figure 2.1: Abstract representations that we use, and what it represents.

Definition 2.1. *The abstract data type Priority Queue stores key-value pairs and supports the following two operations:*

- *insert:* This adds a given key-value pair to the priority queue, with the priority specified by the key.
- *deleteMax:* This removes and returns the item with the highest priority.

There are also typically helper-functions such as *isEmpty* or *size*, and sometimes a function *findMax* that gets the item of highest priority without removing it. We will not give details how to implement these; it should be obvious enough.

Definition 2.1 is the *max-oriented version* of a priority queue, where we have an operation that allows us to retrieve the item of highest priority. In some applications that we will see later in the course, it will help to have a *min-oriented priority queue* instead, where we have an operation *deleteMin* that removes and returns the item with least priority. We will not discuss how to implement this min-oriented version; it is not difficult to do directly (typically one only needs to reverse inequalities in the code), or one could simply use the negative of the given priorities. From now on, whenever we speak of priority queue, we mean a max-oriented priority queue unless explicitly specified otherwise.

2.1.1 Applications of priority queues

Priority queues are useful for any application where the order in which you want to parse items may or may not correlate with the order in which items are added. Typical examples are todo-lists of any kind. One particular application of interest is discrete-event simulation, a simulation technique where each event creates the next few events (which are at some point in the future and are hence tagged with their time as their priority), and we repeatedly need to extract the event whose time is smallest.

The first major application that we will see here of priority queues is sorting (there will be other applications in Section 10.1.2 and 11.2.1). In particular, we can sort using a priority queue by inserting the items to be sorted, and then repeatedly extracting the maximum, hence obtaining the reverse sorted order. Algorithm 2.1 shows the pseudo-code of *priority-queue-sort*.

Algorithm 2.1: *priority-queue-sort*($A, n \leftarrow A.size$).

Input : Array A of size at least n

```

1 Initialize a priority queue  $PQ$ 
2 for  $i \leftarrow 0$  to  $n - 1$  do
3    $x \leftarrow$  key-value pair where the key is  $A[i]$  (the value is unused)
4    $PQ.insert(x)$ 
5 for  $i \leftarrow n - 1$  down to  $0$  do
6    $x \leftarrow PQ.deleteMax()$ 
7    $A[i] \leftarrow x.key$ 

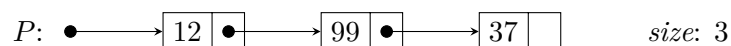
```

What is the run-time of *priority-queue-sort*? We cannot say! The run-time depends entirely on how the priority queue is realized, which is (intentionally) hidden from the user. Sometimes we say (somewhat informally) that the run-time is $n*insert+n*deleteMax$, meaning that is we know the run-time for *insert* and *deleteMax* then the run-time of *priority-queue-sort* depends on them in this way.

2.1.2 Elementary implementations

There are a few straightforward ways to implement a priority queue, using data structures that you have seen in previous courses.

- **Unsorted list.** We store the items in a list in no particular order. This means that insertion is trivially doable in $O(1)$ time; simply insert the new item somewhere in the list. (Usually insertion in the front is easiest.) On the other hand, to find (and remove) the item of maximum priority, we must search through the entire list, which takes $\Theta(n)$ time.



What is the run-time of *priority-queue-sort* with this realization? It becomes $\Omega(n^2)$, because we execute *deleteMax* $\Omega(n)$ times, and each execution takes $\Omega(n)$ time. It also is in $O(n^2)$ since all operations take $O(n)$ time, so it is in $\Theta(n^2)$.

- **Unsorted array.** The idea for this is very similar as for the unsorted list: Store the items in an array in no particular order. Insertion hence is again very easily done in $O(1)$ time by simply adding the item at the end. Deleting the maximum is a touch more complicated, because we should not simply delete an item in the middle of an array, we should also fill the resulting gap. (This can for example be done by exchanging the item with the last item in the array.) The bottleneck for deleting the maximum is (just like for unsorted lists) the process of *finding* the maximum; to do this we must search through the entire array, which takes $\Theta(n)$ time.

0 1 2 3 4
 P:

12	99	37		
----	----	----	--	--

 size: 3

Hence *priority-queue-sort* with unsorted arrays again takes $\Theta(n^2)$ time. This algorithm is actually the well-known sorting algorithm *selection-sort* (you should have seen it in previous courses); see Algorithm 2.2. The code can be simplified much because A initially already contains all items, and so we can skip the insertion-part.

Algorithm 2.2: *selection-sort*($A, n \leftarrow A.size$)

Input : Array A of size at least n

```

1 ; for  $i \leftarrow n - 1$  down to 0 do
    // find the maximum element in  $A[0..i]$  and move it to  $A[i]$ 
2   for  $j \leftarrow i - 1$  down to 0 do
3     if  $A[j] > A[i]$  then
4       swap  $A[j]$  and  $A[i]$ ;

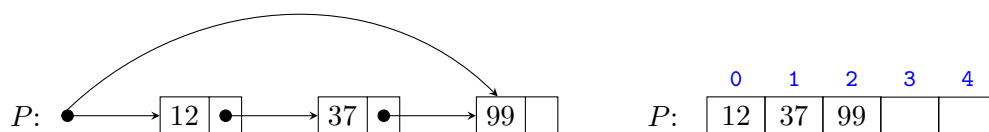
```

Brief detour: dynamic arrays. There is one important detail that we have glossed over when discussing how to do *insert* in an implementation with arrays. The idea was to insert the item at the end of the current array. But as you should know, an array has a *capacity*, i.e., a maximum number of items that it could store. (This is in contrast to the *size* of the array, which is the number of items that the array actually stores.) We cannot add further items to the array if it is at capacity.

However, there is a standard method for dealing with this. We turn the array into a *dynamic array* (sometimes also called *vector*) which expands as needed to make capacity available. It does so by allocating a new array with twice the capacity, and copying all elements over. This takes $\Theta(n)$ time (where n is the current size), but happens only after $\Theta(n)$ insert-operations that took constant time. Therefore when averaging over all

operations the time to increase the capacity of the array is $\Theta(1)$. This is an example of so-called *amortized analysis*: consider the run-time as averages over many operations. (The enriched section will see more details of this.) We will assume from now on that any array that we use is actually a dynamic array, and automatically increases the capacity when needed. We will also often omit “amortization” in our run-time statements when dynamic arrays are used; for example we say “insertion in an unsorted array takes $O(1)$ worst-case time” though it really should be “worst-case amortized time”.

- **Sorted list/array.** Implementing a priority queue with a sorted list/array means that we store the items sorted according to their priorities. This makes it completely trivial to find the maximum—presuming that we sort in increasing order, it is simply the last element. (One hence should maintain a reference to the last element of the list. Or store the list in reverse order.) Therefore *deleteMax* can be implemented in constant time.



The price to pay is during insertion. Here we must put the item in the right place according to the sorted order. This involves searching through the list/array until we find the right place. For arrays, we additionally also need to move all larger items to the right. Either way, the worst-case run-time is $\Theta(n)$. However, in the best case the run-time is only $O(1)$, because we may be lucky and the new item fits exactly into the sorted order at the place where we begin searching for it.

Hence *priority-queue-sort* with sorted lists/arrays takes $\Theta(n^2)$ time in the worst case, but $O(n)$ time in the best case. The implementation with sorted arrays is actually the well-known sorting algorithm *insertion-sort* that we encountered earlier. (The code in Algorithm 1.2 is simplified much compared to *priority-queue-sort*, because after inserting all elements array *A* contains all items in sorted order, and so we can skip the *deleteMax*-part.)

2.2 Non-trivial PQ realization: Binary heaps

We now give a third realization of ADT Priority Queue, which combines the best of the above two approaches: Be sufficiently sorted so that we can find the maximum quickly, but not so sorted that insertion takes too long. The actual implementation will use a (dynamic) array, but in terms of understanding what is going on, it is easiest to visualize this implementation as a binary tree. So we first review some definitions/conventions for binary trees.

2.2.1 Binary Tree Review

A *binary tree* is a recursively defined data structure. It can have one of two forms: It can either be an *empty tree*, or it can consist of a *node* (an element of the data structure that stores information) and two binary trees (the *left subtree* and *right subtree* of this node). In our pictures, we will occasionally use a symbol such as \emptyset to remind that there is an empty tree, but most of the time we do not show the empty subtrees. Pseudo-code will assume that an empty subtree is stored as NIL.

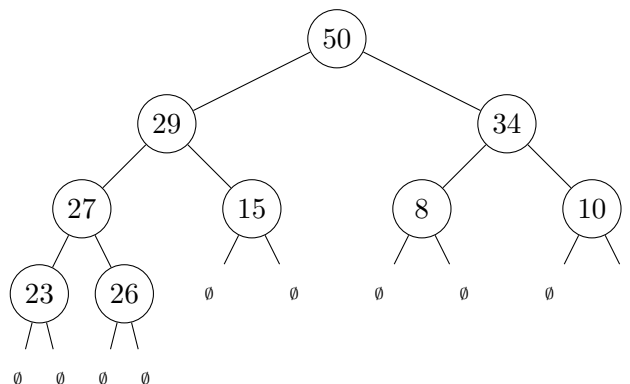


Figure 2.2: A binary tree that is also a heap.

Recall the following terms for binary trees. The *root* is the node that is not inside a subtree of some bigger tree; in Figure 2.2 the root is the node that stores 50. (To simplify descriptions, we will from now on identify a node by the priority that it stores, unless this leads to confusion.) A *child* of a node v is a node c that is the root of either the left or the right subtree of v ; for example nodes 29 and 34 are children of 50. The *parent* of a node v is the node p for which v is a child; for the root the parent is undefined or NIL. A *link* is the connection between a child and its parent.

An *ancestor* of a node v is node a that is the parent of the parent of ... of the parent of v . We include v in this definition; all other ancestors of v are called *strict ancestors*. A node v is a *descendant* of node w if w is an ancestor of v . Every node v has the root as an ancestor. The *depth* of v is the distance that it has from the root, i.e., how often we have to invoke “parent” to get from v to the root. In particular, the root has depth 0, its left and right child (if they exist) have depth 1, etc. In the above example, node 15 has depth 2 while node 23 has depth 3.¹ The *level i* of a tree is the set of nodes that all have the same depth i , thus level 0 contains only the root, level 1 contains the children of the root, etc. In the above example, level 2 contains 27, 15, 8, 10.

¹Warning! The community is split over whether the root should be considered to have depth 0 or depth 1, and you may encounter a different definition in some textbooks or other courses. In CS240 the root will always have depth 0.

The height of the tree is the maximum depth among its nodes. In particular, a tree that has exactly one node has height 0, and the tree in the example has height 3. Note that empty subtrees are *not* considered to be nodes, so they do not count towards the height. To make some formulas work out, we define the height of an empty tree to be -1 . A *leaf* is a node where both left and right subtree are empty; in Figure 2.2 the leaves are 23, 26, 15, 8 and 10. A node that is not a leaf is an *interior node*; in Figure 2.2 the interior nodes are 50, 29, 34, and 27.

There are some standard ways to enumerate the nodes of a tree, all of which can be done in linear time. In the *pre-order enumeration*, we first list the root, then all items of the left subtree, then all items of the right subtree. The *post-order* lists first subtree items and then the root, while the *in-order* lists the left subtree items, then the root, and then the right subtree items. In Figure 2.2 the three orders are

- Pre-order: 50, 29, 27, 23, 26, 15, 34, 8, 10
- Post-order: 23, 26, 27, 15, 29, 8, 10, 34, 50
- In-order: 23, 27, 26, 29, 15, 50, 8, 34, 10

Another way to enumerate nodes is the *level-order*. Here we list all nodes on level 0, then on level 1, then on level 2, etc. Within each level, we list the nodes from left to right. In Figure 2.2, the level-order is 50, 29, 34, 27, 15, 8, 10, 23, 26.

You should be familiar with properties of binary trees, but we state here a few that will be needed later. Their proofs (which are easily done by induction) are left as exercise.

Observation 2.1. *The following holds for binary trees.*

1. Level ℓ in a binary tree contains at most 2^ℓ nodes. We call the level full if it contains exactly 2^ℓ nodes.
2. Any binary tree of height h has at most 2^h leaves.
3. Any binary tree with L leaves has at least $L - 1$ interior nodes.
4. Any binary tree with L leaves where every interior node has two children has exactly $L - 1$ interior nodes.
5. A binary tree of height h contains at most $2^{h+1} - 1$ nodes. We call a binary tree complete if it has exactly $2^{h+1} - 1$ nodes, i.e., all levels are full.
6. Any binary tree has height at least $\log(n + 1) - 1 \in \Omega(\log n)$.

Trees of higher arity

We will for some data structures later encounter *multi-way trees*, also known as *trees of higher arity*. Formally, a tree has *arity* or *order* d if every node has up to d subtrees (possibly empty ones). (Such nodes are called *multi-way nodes*.) Any tree of order d has height at least $\log_d(n + 1) - 1 \in \Omega(\log_d n)$. (While d is often a constant, and then $\Omega(\log_d n) = \Omega(\log n)$, we often keep d as a separate parameter to emphasize that the height is noticeably smaller as d gets bigger.)

2.2.2 Heap definition

We are now ready to define a (*binary*) *heap*. (Most of the time we simply call this a *heap*, but the enriched section will study other realizations that are similar and then use *binary heap* to distinguish this realization from others.) Also, to clarify: You may have encountered the term “heap” for the part of the memory on your computer that you have control over. This has *nothing* to do with the concept we are about to see (other than the name).

Definition 2.2. A binary heap is a binary tree that stores key-value pairs and has two properties:

- **Heap-order-property:** For any node v , all keys at descendants are no bigger than $v.key$. Put differently, as we go up in the binary tree, the keys do not decrease.
- **Structural property:** Any level except the last one is full, and the last level is filled from the left.

The example in Figure 2.2 is in fact a heap. Observe that while we impose an order of keys from child to parent, we do *not* impose an order of keys between siblings; the left child’s key can be smaller or bigger (or even equal) to the right child’s key. The following insight is straightforward but crucial:

Lemma 2.1. The height of a heap with n nodes is in $\Theta(\log n)$.

Proof. Let h be the height of the heap, so we have levels $0, 1, \dots, h$ and all levels except level h are full. Therefore we have 2^ℓ nodes on level ℓ for $\ell < h$ and at least one node on level h . In consequence

$$n = \sum_{\ell=0}^h \# \text{nodes on level } \ell \geq \sum_{\ell=0}^{h-1} 2^\ell + 1 = (2^h - 1) + 1 = 2^h$$

by the geometric sum formula. Resolving for h , we get $h \leq \log n \in O(\log n)$, and we know $h \in \Omega(\log n)$ since this holds for any binary tree. \square

2.2.3 Storing heaps

We usually depict heaps as binary trees, because this makes the algorithms to come much easier to visualize. However, for storing in a computer, using a binary tree (with two or maybe even three references per node) wastes a lot of space. Instead, we store the binary tree as an array. Specifically, due to the structural property the level-order of a heap would have no *gap*, i.e., a place where a node could have been but where there is no node. Therefore to store a heap in an array A , store the i th node in level-order at $A[i]$.² Figure 2.3 illustrates this conversion.

Given an array of keys, it is really hard for a human to see whether it satisfies the heap-order property, which is why we usually draw the binary tree instead. But for purposes of

²In this course, arrays start at index 0, so we store the root at $A[0]$. Note that some textbooks do this differently and store the root at $A[1]$, which noticeably affects the formulas for getting from parent to child and vice versa.

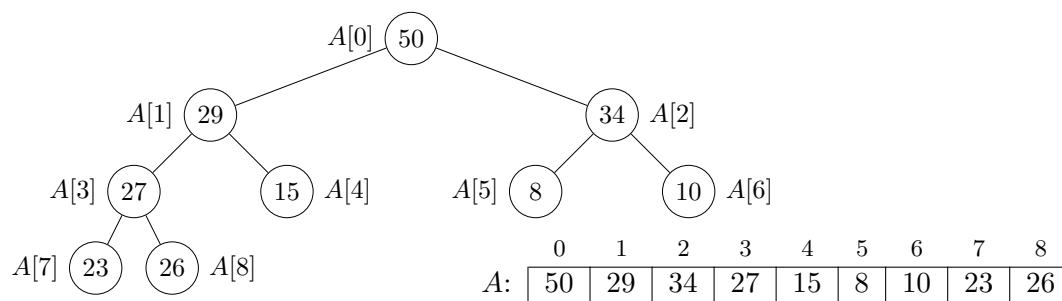


Figure 2.3: A heap as binary tree and as corresponding array.

implementing, an array uses much less space, and just as easy to work with if we have *accessor-functions*, i.e., helper-functions that provide navigation as if we had a tree-based implementation. Observe the following:

- The *root* node is at index 0. (Normally “node” refers to the tree and “index” refers to the array, but because for a heap the two are in 1-1-correspondence, we use the two terms quite interchangeably from now on.)
- The *last* node is at index $n - 1$, where n is the current *size*.
- The *left child* of node i (if it exists) is node $2i + 1$.
- The *right child* of node i (if it exists) is node $2i + 2$.
- The *parent* of node i (if it exists) is node $\lfloor \frac{i-1}{2} \rfloor$.
- A node (such as left/right child or parent) exists if and only if the corresponding index falls in the range $\{0, \dots, n-1\}$.

We assume that for each feature we have a corresponding helper-function so that the code *never* works with indices in the array directly and can hence pretend to have a binary tree. Some of these helper-function (such as *last* and ‘has a right child’) need to know the current size of the heap; we assume that this has been stored suitably.

2.2.4 Heap operations

Recall that we are studying heaps because we want another way to realize ADT Priority Queue. So we must discuss how to do *insert* and how to do *deleteMax*.

Insert: Assume that we are given a key-value pair (k, v) that we want to add. By the heap-structure property, there is a *unique* place where to add a new node, namely, to the right of the previously last node (or as the leftmost node of a new level if the currently last level is full). Thus we do not have any choice where to put (k, v) . But putting it there may destroy the heap-order property. So we must restore the heap-order property, and do this with routine *fix-up*, which exchanges the item with its parent until it has “bubbled up” to the place where it should be. See Algorithm 2.3 for the code and Figure 2.4 for an example.

Algorithm 2.3: *fix-up*(A, i)

Input : Array A stores a heap. i is a node of the heap.

```

1 while  $\text{parent}(i)$  exists and  $A[\text{parent}(i)].\text{key} < A[i].\text{key}$  do
2   | swap  $A[i]$  and  $A[\text{parent}(i)]$ 
3   |  $i \leftarrow \text{parent}(i)$ 

```

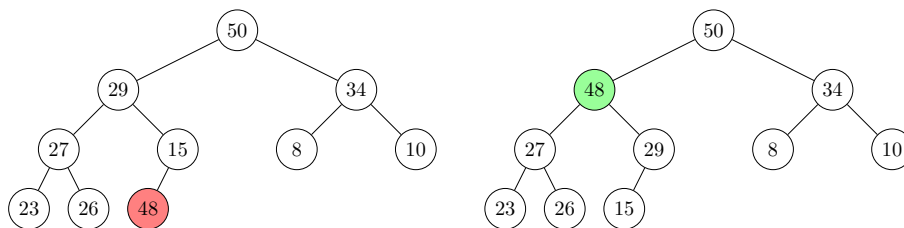


Figure 2.4: An example of operation *fix-up* where initially i points to the last node. Not all intermediate steps are shown.

We already explained above how to use *fix-up* to do insertion in a binary heap; Algorithm 2.4 gives the pseudocode. Let us clarify briefly the notation *binaryHeap::insert*. Because we will see many insertion-routines, and they are usually called *insert*, we will (where needed) specify in this way exactly which class the routine belongs to. It is also assumed that this class stores the data in the manner explained in the text, e.g. class *binaryHeap* is assumed to store the array A that has the actual heap, and to know the current size of the heap.

Algorithm 2.4: *binaryHeap::insert*(k, v)

Input : key-value pair (k, v)

```

1  $\text{size}++$ ,  $\ell \leftarrow \text{last}()$  // Increased  $\text{size}$  first, so  $\ell$  is the first free spot
2  $A[\ell] \leftarrow (k, v)$ 
3  $\text{fix-up}(A, \ell)$ 

```

The run-time of the insertion is dominated by the time for *fix-up*. This time is proportional to the height of the heap, and hence $O(\log n)$. So insertion takes $O(\log n)$ time in heaps.³ This is tight if the new key is big enough that it needs to move up all the way to the root.

deleteMax: Now let us turn towards finding and removing the maximum item in the heap. Directly from the heap-order property, we know that the maximum resides at the root, because if any other key (say at node k) were bigger, then the heap-order property would have to be

³We are again tacitly assuming that A is a *dynamic array*, so technically the worst-case run-time is $\Theta(n)$, while the amortized run-time is $O(\log n)$. But we will ignore this and pretend that the worst-case run-time is $O(\log n)$.

1517 violated somewhere along the path from k to the root. It is possible that there are multiple
 1518 copies of the maximum key, but one of them is definitely at the root, so finding the maximum
 1519 is easy.

1520 Removing the maximum is a bit harder. We cannot just remove the key-value pair, as this
 1521 would leave a gap and violate the heap-structure property. But we know exactly which node
 1522 can be removed: the one at the last position. So (after storing the contents of the root for
 1523 later returning) we move the key-value pair x from the last node to the root. With this, the
 1524 heap-order property may be violated between the root and its children. We restore it at the root
 1525 by moving x downward, exchanging it with the larger of the children. But now the heap-order
 1526 property may be violated at the new location of x , so we keep moving x downward until the
 1527 heap-order property holds. See Algorithm 2.5 for the code and Figure 2.5 for an example.

Algorithm 2.5: *fix-down*($A, i, n \leftarrow A.size$)

Input : Array A stores a heap of size n . i is a node of the heap.

```

1 while  $i$  is not a leaf do
2    $j \leftarrow$  left child of  $i$                                 // Find the child with the larger key
3   if  $i$  has a right child and  $A[\text{right child of } i].key > A[j].key$  then  $j \leftarrow$  right child of  $i$ 
4   if  $A[i].key \geq A[j].key$  then break
5   swap  $A[j]$  and  $A[i]$ 
6    $i \leftarrow j$ 

```

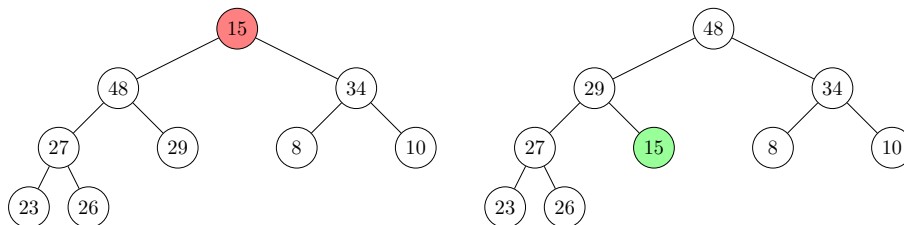


Figure 2.5: An example of operation *fix-down* where initially i points to the root. Not all intermediate steps are shown.

1528 The reader may also wonder why *fix-down* is passed n as a parameter, while *fix-up* did not
 1529 use this. The reason is that *fix-down* uses helper-functions such as ‘is i a leaf?’ and ‘does i
 1530 have a right child?’. All these helper-functions need to know the size n of the heap to work properly
 1531 (we did not show parameter n to keep the code less cluttered). We will occasionally use *fix-down*
 1532 in situations where the size of the heap is *less* than the size of the array, and so must pass along
 1533 a parameter that allows us control over this size.

1534 As described earlier, the actual *deleteMax* routine only has to do a few additional steps and
 1535 is given in Algorithm 2.6. The run-time is dominated by *fix-down*, and again proportional to

1536 the height of the heap, which is $O(\log n)$. This is tight if the last item of the heap was small
 1537 enough that *fix-down* needs to return it back to the lowest level.

Algorithm 2.6: *binaryHeap::deleteMax()*

```

1 i ← last()
2 swap A[root()] and A[i]
3 size-- // Now max-item no longer is in heap, so won't be moved by fix-down
4 fix-down(A, root(), size)
5 return A[i] // We had stashed the max-item in A[i]

```

1538 2.3 Heapsort

1539 Recall the *priority-queue-sort* algorithm (Algorithm 2.1). The run-time of this was n times the
 1540 time for *insert* plus n times the time for *deleteMax*. If we use heaps as implementation of priority
 1541 queues, then both *insert* and *deleteMax* take $O(\log n)$ time. In consequence, we can sort with
 1542 this method in $O(n \log n)$. This matches *merge-sort* in run-time, but it has the advantage that
 1543 there are no recursions. So this sorting algorithm is quite useful in practice. For this reason,
 1544 it has been optimized further. There are two obvious ways of improving it; the pseudo-code
 1545 for this improved version is in Algorithm 2.7 and we will discuss its ingredients in the following
 1546 sections.

Algorithm 2.7: *heap-sort*(*A*, $n \leftarrow A.size$)

Input : Array *A* to be sorted.

```

1 for i ← parent(last()) down to root() do // heapify: Build the whole heap at once
2   fix-down(A, i, n)
3 while n > 1 do // re-use array: do deleteMax directly in A
4   swap items at A[root()] and A[last()]
5   n--
6   fix-down(A, root(), n)

```

1547 2.3.1 Making the algorithm in-place

1548 In the sorting problem, the input is usually given as an array. A heap also uses an array. As it
 1549 turns out, we can use the *same* array for the input and the heap. Thus we can make heapsort
 1550 an *in-place* sorting algorithm: it sorts directly in the array that contained the input, and hence
 1551 uses only $O(1)$ auxiliary space. This has the advantage that less time is needed for allocating

1552 a new array and moving items between arrays, making the algorithm faster in practice (though
1553 not asymptotically faster).

1554 To see how to create an in-place sorting algorithm, observe that the heap uses the range
1555 $A[0..k-1]$ of array A , where k is the current size of the heap. If we call *deleteMax*, then the
1556 current maximum x is extracted from $A[0..k-1]$, and the size decreases so that the heap now
1557 uses only $A[0..k-2]$. Therefore we can place x directly at $A[k-1]$ without needing extra space.
1558 Conveniently, item x belongs at slot $A[k-1]$ in the sorted order (because there are $k-1$ items
1559 that are smaller than it—those that are still in the heap) and so it is automatically put into the
1560 correct place. So once the heap is empty the array is in sorted order.⁴

1561 2.3.2 Heapify

1562 Recall the priority-queue sort works in two parts: first insert all items into the priority queue,
1563 then repeatedly extract the maximum. The standard way to do the first part would be to
1564 insert in a priority queue n times. Thus, create an empty heap H , and call $H.insert(A[i])$ for
1565 $i = 0, \dots, n-1$. Then copy the array that stores H back to A . This approach uses n calls to
1566 *fix-down*, and takes $O(n \log n)$ time.

1567 One could improve this slightly by using the same array to store H and A , which is feasible.
1568 But a much bigger improvement is achieved by observing that we have *all* items of A available
1569 beforehand. It is therefore possible to consider all elements at once (rather than the “add one
1570 element at a time” approach that we used above). As we will see, this can be exploited to
1571 get a faster run-time. Consider the *heapify* method given in Algorithm 2.8 and illustrated in
1572 Figure 2.6.

Algorithm 2.8: *heapify*($A, n \leftarrow A.size$)

Input : Array A of size n .

Output: A satisfies the heap-order property.

1 **for** $i \leftarrow \text{parent}(\text{last}())$ **downto** $\text{root}()$ **do** *fix-down*(A, i, n)

1573 **Lemma 2.2.** *Algorithm heapify takes $\Theta(n)$ time on an array of size n .*

1574 The lower bound is obvious due to the for-loop, which goes from $\lfloor \frac{n}{2} \rfloor - 1$ to 0. The proof of
1575 the upper bound is not excessively complicated, but more difficult than other analyses that you
1576 have seen on trees. We will give two proofs, of two very different flavors. The first proof is of the
1577 kind that you might have been able to come up with yourself (though resolving the inequality
1578 is far from easy), while the second proof is much shorter, but involves an unusual re-think.

1579 *Proof.* We will only count the number of *key-comparisons*, i.e., the number of times that we
1580 execute a comparison of the form “is $A[i] < A[j]$ ” for some indices i and j . (This a standard

⁴This is the main reason for using a max-oriented heap for heapsort. In most other applications of priority queues, a min-oriented heap is preferred.

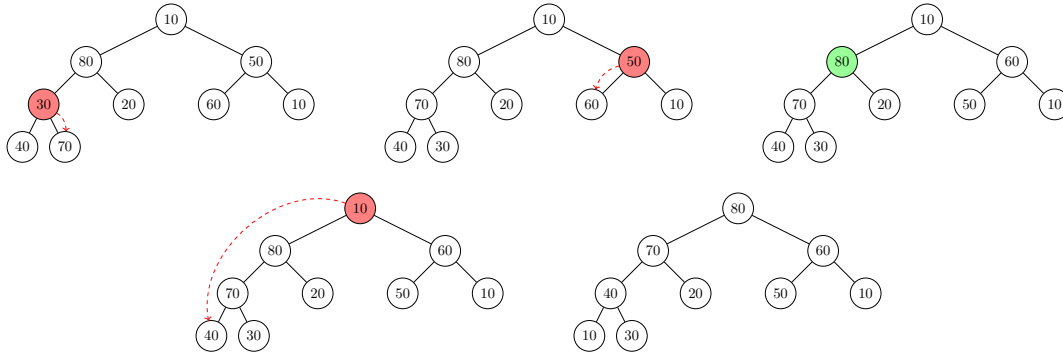


Figure 2.6: Heapify. We show the heap whenever *fix-down* is called on the next node, which may or may not be moved downwards.

trick that we will use again in Chapter 3.) Clearly the algorithm can't be faster than the number of key-comparisons that we use, and one easily sees that the number of other steps is proportional to the number of key-comparisons.

Let h be the height of the heap, and consider a node v on level ℓ for $0 \leq \ell \leq h$. How many comparisons occurred during *fix-down*(v)? We compare the two children of v , and then v to the larger child, so we use at most two key-comparisons when v is on its original level. Then (maybe) we move v one level downward and repeat. Since there are $h - \ell$ levels below v , the total number of key-comparisons during *fix-down*(v) is hence at most $2(h - \ell)$.

There are at most 2^ℓ nodes on level ℓ . Therefore all nodes on level ℓ together contribute at most $2(h - \ell) \cdot 2^\ell$ key-comparisons during their calls to *fix-down*. Thus we have

$$\# \text{ key-comparisons} \leq \sum_{\ell=0}^h 2(h - \ell)2^\ell = \sum_{\ell=0}^h \ell \cdot 2^{h-\ell+1} = 2^{h+1} \sum_{\ell=1}^h \frac{\ell}{2^\ell} \leq 2^{h+1} \sum_{\ell=1}^{\infty} \frac{\ell}{2^\ell}.$$

We will argue the following below:

Claim 2.1. We have $\sum_{\ell=1}^{\infty} \frac{\ell}{2^\ell} = 2$.

Therefore the number of key-comparisons is upper-bounded by $2^{h+1} \cdot 2 = 2^{h+2}$. But recall that $h \leq \log n$, so the number of key-comparisons is at most $2^{h+2} \leq 4n \in O(n)$.

It remains to prove the claim. We give here four proofs, which all are perfectly correct, but you may find one easier to understand than the other.

1. Use brute-force evaluates the sum as follows:

$$\sum_{\ell=1}^{\infty} \frac{\ell}{2^\ell} = \sum_{\ell=1}^{\infty} \sum_{j=\ell}^{\infty} \frac{1}{2^j} = \sum_{\ell=1}^{\infty} \frac{1}{2^\ell} \sum_{j=0}^{\infty} \frac{1}{2^j} = \sum_{\ell=1}^{\infty} \frac{1}{2^\ell} \cdot 2 = \sum_{\ell=1}^{\infty} \frac{1}{2^{\ell-1}} = \sum_{\ell=0}^{\infty} \frac{1}{2^\ell} = 2$$

2. Arrange the sum in a 2-dimensional array suitably, sum up within each row and then sum up the row-sums.

$$\begin{array}{rcl}
 \sum_{\ell=1}^{\infty} \frac{\ell}{2^{\ell}} & = & \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1 \\
 & & + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = \frac{1}{2} \\
 & & + \frac{1}{8} + \frac{1}{16} + \dots = \frac{1}{4} \\
 & & + \frac{1}{16} + \dots = \frac{1}{8} \\
 & & + \vdots + \dots = \vdots \\
 & & \hline
 & & = 2
 \end{array}
 \quad \begin{array}{l}
 \vdots \\
 \text{sum this way} \\
 \downarrow
 \end{array}$$

3. Evaluate a suitable differential. To do so, define shortcut $S_2 := \sum_{i=1}^{\infty} \frac{i}{2^i}$ and observe that

$$2 \cdot S_2 = 2 \cdot \frac{1}{2} + 2 \cdot \frac{2}{4} + 2 \cdot \frac{3}{8} + 2 \cdot \frac{4}{16} + \dots = \frac{1}{1} + \frac{2}{2} + \frac{3}{4} + \frac{4}{8} + \dots = 1 + \sum_{i=1}^{\infty} \frac{i+1}{2^i}$$

But then

$$S_2 = 2 \cdot S_2 - S_2 = 1 + \sum_{i=1}^{\infty} \frac{i+1}{2^i} - \sum_{i=1}^{\infty} \frac{i}{2^i} = 1 + \sum_{i=1}^{\infty} \frac{(i+1) - i}{2^i} = 1 + \sum_{i=1}^{\infty} \frac{1}{2^i} = 1 + 1 = 2$$

4. Argue convergence of the sum. (We do not get here the (tight) bound of 2, but only a looser upper bound, but it is still in $O(1)$ which is enough to get the linear run-time for *heapify*.) Namely, the *ratio convergence test* states that an infinite sum $\sum_{i=1}^{\infty} a_i$ converges if $\lim_{i \rightarrow \infty} \frac{a_{i+1}}{a_i}$ exists and is smaller than 1. In our case,

$$\frac{a_{i+1}}{a_i} = \frac{i+1}{2^{i+1}} \cdot \frac{2^i}{i} = \frac{1}{2} \left(1 + \frac{1}{i}\right) \rightarrow \frac{1}{2},$$

and therefore $\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges to some constant, i.e., is in $O(1)$.

□

Now we give a second, completely different, proof that *heapify* takes linear time.

Proof. Consider a recursive version of *heapify* shown in Algorithm 2.9. One easily convinces oneself that this indeed does exactly the same as *heapify* (and hence has asymptotically the same run-time). Namely, the *fix-down* commands are still executed at the bottom-most level first and then higher up, though the order in which they are executed is different (we do them in post-order, rather than reverse level-order).

Let $T(n)$ be the run-time to do *heapify-recursive*(A, n, root). Let us assume that the heap has all levels full (if this is not the case, then we can artificially pad the heap with very small

Algorithm 2.9: *heapify-recursive*($A, i \leftarrow \text{root}(), n \leftarrow A.\text{size}$)

```

1 if  $i$  has a left child, say  $i_L$  then heapify-recursive( $A, i_L, n$ )
2 if  $i$  has a right child, say  $i_R$  then heapify-recursive( $A, i_R, n$ )
3 fix-down( $A, i, n$ )

```

items until the level is full and only double the run-time). Then there are $(n - 1)/2$ nodes in each of the two sub-heaps. We know that *fix-down* takes $O(\log n)$ time. Therefore the run-time $T(n)$ satisfies the recursion $T(1) \leq c$ and

$$T(n) \leq 2T\left(\frac{n-1}{2}\right) + c \cdot \log n \leq 2T\left(\frac{n}{2}\right) + c \cdot \log n \quad \text{for } n > 1,$$

where c is a constant. This resolves to $O(n)$ (specifically, $T(n) \leq 3cn - c \cdot \log n - 2c$). \square

Summarizing, executing *heapify* rather than building the heap with multiple calls to *insert* will bring down the run-time for building the heap from $O(n \log n)$ to $O(n)$. This does not improve the worst-case complexity of the sorting algorithm, because we still have to do the repeated deletion of the maximum (and this takes $\Theta(n \log n)$ time in the worst case), but it will be faster in practice. Algorithm 2.7 gives the final complete pseudocode for *heap-sort*, the version of *priority-queue-sort* that uses binary heaps and the above improvements.

As mentioned, heap-sort is non-recursive and performs quite well in practice. It has some disadvantages as well. First, it's much less intuitive, especially if you execute it directly on the array. Second, it is not *stable*, i.e., if you have two items with equal keys, then there is no guarantee which of them will end up earlier in the final sorted output. This is something that we will be concerned with later.

2.4 More PQ operations (cs240e)

Binary heaps were primarily invented as an implementation of priority queues, but can easily support other operations:

- *findMax*: This operation only finds the maximum element, without removing it. In a binary heap, this can be done in $\Theta(1)$ time, since the maximum is always at the root.
- *decreaseKey*: This operation has as parameter a reference i to the location of one item of the heap. (We call such a reference a *handle*: it refers to one particular part of the structure, and in particular is implementation-dependent.) In a binary heap implemented as an array, this handle would be an index, while if it were implemented as a tree, it would be one node of the tree.

As a second parameter, *decreaseKey* has a key k_{new} . It is supposed to decrease the key of item i to become k_{new} (it needs to do nothing if the key of i was already smaller than k_{new}). This can easily be done in a binary heap: If we decrease a key, then the only place

where the heap-order property could be violated is from item i to its children; one call to $fix_down(i)$ will restore the heap properties.

- *increaseKey* symmetrically should replace the key at a given item by a new one. In a binary heap, one call to $fix_up(i)$ will restore the heap properties in $O(\log n)$ time.
- *delete*: Generally, deleting an item i in a binary heap is difficult, because we would have to find i first, and there is no routine for searching for an item in a binary heap. But if we assume that we are given a handle to i , then deletion is easy: First increase the value of the key at i to ∞ (or to $findMax() + 1$, which is sufficient), and then call $deleteMax$. This takes $O(\log n)$ time in a binary heap.

You will see some applications for these operations in later classes, for example in cs341 (min-oriented) priority queues that support *decreaseKey* are used for finding paths of smallest weight in graphs.

We finally turn to one more operation, which can not easily be supported by binary heaps, but which other implementations of ADT Priority Queue can support. This is the operation

merge: Given two priority queues P_1, P_2 , create a new priority queue P that contains exactly the elements in P_1 and P_2 .

If some element exists in both P_1 and P_2 , then in the result this element should exist twice. (Recall that priority queues are allowed to have duplicates.)

If binary heaps are stored as arrays, then merging means that at least one of the two arrays has to be copied over into another array, and so this requires $\Omega(\min\{n_1, n_2\})$ time (where n_1, n_2 are the sizes of P_1 or P_2). If $n_1 \approx n/2 \approx n_2$ then merging hence must take linear time, which is too slow. (Recall that we could build the entire heap from scratch in linear time.) So if we want sub-linear run-time for merging, we must assume that the heap is stored as a tree. Merging two binary heaps then *is* possible in $O(\log^3(n_1 + n_2))$ time, but it is quite complicated and the details will not be given here. Instead we give two variants of binary heaps (both are tree-based) where merging is comparatively easy.

2.4.1 Meldable heap

We first study *meldable heaps*, which are the same as binary heaps except that they drop the structural property. Thus a meldable heap is a binary tree where for every node v the key at v is the maximum among all keys of descendants of v .

The standard priority-queue operations can be implemented on a meldable heap by using *merge* as a sub-routine.

- *insert*: Create a 1-node meldable heap P' with the new key-value pair that we want to add. Then call $merge(P, P')$ with the existing and the newly-created heap. See Figure 2.7.
- *deleteMax*: The maximum is at the root. Remove it, but rather than trying to fill the resulting hole (as we did with binary heaps), instead take the two sub-heaps P_L and P_R at the two children. Then apply $merge(P_L, P_R)$ to obtain the new heap.

1669 The run-time for both is hence proportional to the run-time for *merge* since all other steps take
 1670 constant time.⁵

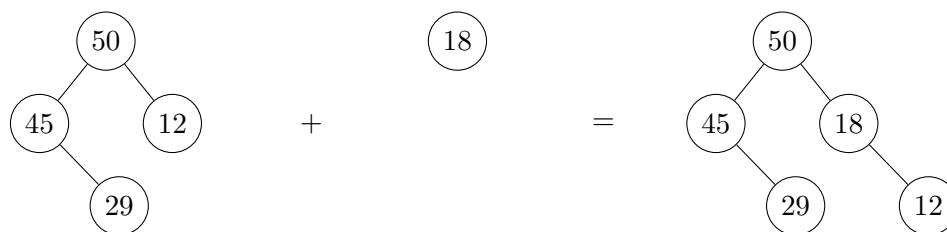


Figure 2.7: A meldable heap, and doing *insert* via *merge*.

1671 Now we explain how to do *merge* on two meldable heaps P_1, P_2 . After possible renaming
 1672 we may assume that the key at $P_1.root$ is no smaller than the key at $P_2.root$. To maintain the
 1673 heap-order property, therefore $P_2.root$ should become a descendant of $P_1.root$. So the idea is to
 1674 merge P_2 into one of the two sub-heaps at the children of $P_1.root$. But which one?

1675 Here is the crucial idea for meldable heaps: *Randomly* choose which child c of $P_1.root$ is used
 1676 for merging, with equal probability for both children. Then consider the sub-heap P_c rooted
 1677 at this child, and recursively merge P_c and P_2 . The process ends if we reach a child that is an
 1678 empty sub-tree; here we simply place the heap that needs to be merged in. Algorithm 2.10 gives
 1679 the pseudo-code and Figure 2.8 illustrates an example.

Algorithm 2.10: *meldableHeap::merge*(r_1, r_2)

Input : r_1, r_2 : roots of two meldable heaps (r_1 is not NIL)

Output: returns root of merged heap

```

1 if  $r_2$  is NIL then return  $r_1$ 
2 if  $r_1.key < r_2.key$  then swap  $r_1$  and  $r_2$ 
3 randomly pick one child  $c$  of  $r_1$ 
4 replace  $c$  by result of merge( $r_2, c$ )
5 return  $r_1$ 

```

1680 What is the run-time of merging two heaps? The worst-case run-time could be quite bad,
 1681 because a meldable heap could have height $\Theta(n)$, and if we are really unlucky then the random
 1682 choices lead us to merge exactly at the lowest leaf of that heap. But this is very unlikely, since
 1683 we choose the child at which to merge randomly. As such, we should be analyzing the expected
 1684 run-time. You may want to re-read the section on randomized algorithms on Page 45 for details
 1685 of this definition.

⁵This is an example of a *reduction*: To solve one problem (here: how to do *insert* or *deleteMax*), do a few transformation steps (the actual reduction) and then appeal to the solution of a different problem (here: *merge*). Reductions will be a vital tool in cs360 and cs341.

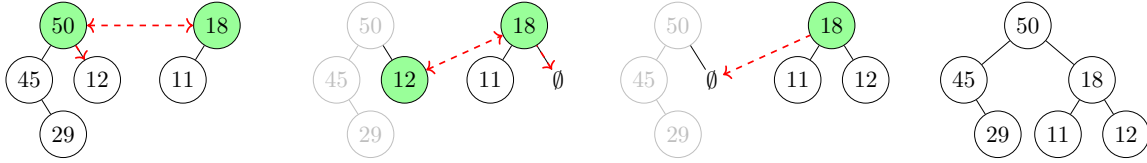


Figure 2.8: Merging two meldable heaps. Since $18 < 50$, we need to merge 18's heap into one child of 50, and we pick 12. Since $12 < 18$, we need to merge 12's heap into one child of 18 and pick the empty subtree. This performs the merge, and we undo the recursion steps to obtain the final meldable heap.

To do so, we first need a helper-result. Let a *random downward walk* in a binary tree be the following: start at the root, randomly (with equal probability) choose one of the two children to go to, and repeat from there until you reach an empty subtree.

Lemma 2.3. *In a binary tree with n nodes, the expected length of a random downward walk is $O(\log n)$.*

Before proving this, let us note that the worst-case length of such a walk is $\Theta(n)$: the meldable heap could have height $\Theta(n)$, and we could walk exactly along the path to the lowest leaf. The intuition why in expectation the length is short is that if the top $\log n$ levels were full, then no nodes are left for lower levels, so the height would be $\log n$. Reversing the argument, having large height means that there are lots empty spots in the top $\log n$ levels, and we have a good chance of hitting one (and hence have a short downward walk). While this is a good intuition of why the result might hold, the actual proof proceeds quite differently.

Proof. Write $T(H, R)$ for length of a random downward walk in a binary tree H if the random coin-flips are R . Here “length” is measured by how many nodes there are on the path. We prove by induction on n that $T^{\text{exp}}(n) = \max_H \sum_R P(R) T(H, R) \leq \log(n+1)$. In the base case, $n = 1$, so H is a single node. Then there are no random flips and the downward walk has length $1 = \log(2) = \log(n+1)$ as desired.

Now assume $n > 1$, let I be a tree on n nodes, and let n_L and n_R be the size of the left and right subtree of H . We know $n_L + n_R = n - 1$, but we know nothing about their relative sizes. With probability $\frac{1}{2}$, we go into the left subtree which has size n_L . Likewise with probability $\frac{1}{2}$ we go into the right subtree which has size n_R . This suggests the formula

$$T^{\text{exp}}(I) = \sum_R P(R) \cdot T(I, R) \leq 1 + \frac{1}{2} T^{\text{exp}}(n_L) + \frac{1}{2} T^{\text{exp}}(n_R).$$

(As for previous examples of randomized algorithms, the upper bound does not hold for one individual sequence R of outcomes, but holds if we take the weighted sum over all R . This can

1705 be shown similarly as in Lemma 1.7.) Therefore

$$\begin{aligned}
 T^{\text{exp}}(n) &= \max_I \sum_R P(R) \cdot T(I, R) \\
 &\leq \max_{n_L + n_R = n-1} \left\{ 1 + \frac{1}{2} T^{\text{exp}}(n_L) + \frac{1}{2} T^{\text{exp}}(n_R) \right\} \\
 &\leq 1 + \max_{n_L + n_R = n-1} \left\{ \frac{1}{2} \log(n_L+1) + \frac{1}{2} \log(n_R+1) \right\}
 \end{aligned}$$

by induction. Now we need a trick from calculus. Remember that $\log(x)$ is a *concave* function, which says that the straight-line segment between two points on its curve lies *below* the curve. Put in mathematical terms, for any positive a, b , and any $0 \leq \lambda \leq 1$, we have

$$\lambda \log(a) + (1 - \lambda) \log(b) \leq \log(\lambda a + (1 - \lambda)b)$$

1706 See also Figure 2.9.

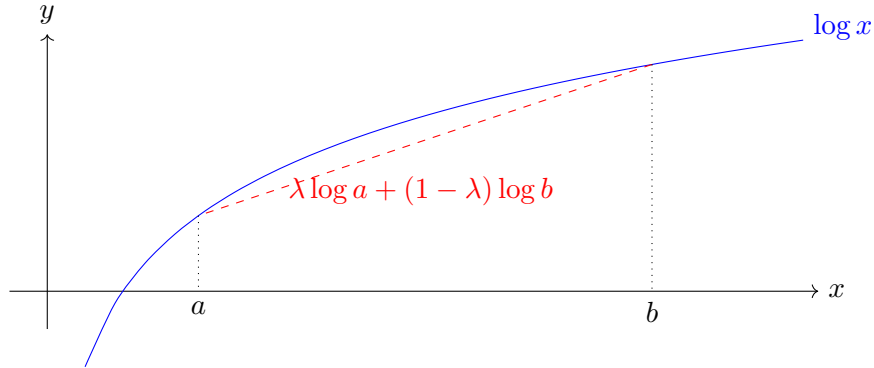


Figure 2.9: The function $\log(x)$ is concave.

1707 We specifically use $a = n_L+1$ and $b = n_R+1$ and $\lambda = \frac{1}{2}$ here, and therefore have

$$\begin{aligned}
 T(n) &\leq 1 + \max_{n_L + n_R = n-1} \left\{ \frac{1}{2} \log(n_L+1) + \frac{1}{2} \log(n_R+1) \right\} \\
 &\leq 1 + \max_{n_L + n_R = n-1} \left\{ \log\left(\frac{1}{2}(n_L+1) + \frac{1}{2}(n_R+1)\right) \right\} \quad \text{by concavity of } \log \\
 &\leq 1 + \max_{n_L + n_R = n-1} \left\{ \log\left(\frac{1}{2}(n+1)\right) \right\} \quad \text{by } n_L + n_R = n-1 \\
 &= 1 + \log\left(\frac{1}{2}(n+1)\right) = 1 + \log(n+1) - 1 = \log(n+1)
 \end{aligned}$$

1708 as desired. □

1709 If you prefer to avoid concavity, here is an alternate (and perhaps easier) proof that gives a
 1710 slightly weaker constant.

Proof. We show that $T^{\text{exp}}(n) \leq 2 \log(n+1)$ by induction on n . Clearly this holds in the base case. In the induction step, we can argue the formula for T^{exp} as above and then use induction to get

$$\begin{aligned} T^{\text{exp}}(n) &\leq \max_{n_L+n_r=n-1} \left\{ 1 + \frac{1}{2}T^{\text{exp}}(n_L) + \frac{1}{2}T^{\text{exp}}(n_R) \right\} \\ &\leq 1 + \max_{n_L+n_r=n-1} \left\{ \frac{1}{2}2 \log(n_L+1) + \frac{1}{2}2 \log(n_R+1) \right\} \\ &= 1 + \max_{n_L+n_r=n-1} \left\{ \log(n_L+1) + \log(n_R+1) \right\}. \end{aligned}$$

We know that one of n_L and n_R is at most $(n-1)/2$ while the other one is at most n . Therefore

$$\begin{aligned} T^{\text{exp}}(n) &\leq 1 + \max_{n_L+n_r=n-1} \left\{ \log\left(\frac{n-1}{2}+1\right) + \log(n+1) \right\} \\ &\leq 1 + \log\left(\frac{n+1}{2}\right) + \log(n+1) = 1 + \log(n+1) - 1 + \log(n+1) = 2 \log(n+1) \end{aligned}$$

1711 as desired. □

1712 Now let us come back to the run-time of *merge* in meldable heaps. We can view *merge* as
 1713 doing two random downward walks, one in each of the heaps P_1 and P_2 . One of them indeed
 1714 needs to walk all the way down until it reaches an empty subtree, the other one may actually
 1715 stop earlier. Clearly the run-time of *merge* is proportional to the lengths of these two walks,
 1716 and so the expected run-time of *merge* is $O(\log n_1 + \log n_2)$, where n_1 and n_2 are the sizes of
 1717 the heaps to be merged.

1718 Summarizing meldable heaps: If we are willing to trust our luck (and to implement heaps
 1719 as trees rather than arrays), then a simple randomized algorithm permits us to merge priority
 1720 queues and perform all other priority queue operations in $O(\log n)$ expected time.

1721 2.4.2 Binomial heaps

1722 We next study *binomial heaps*, which have a lot more changes compared to binary heaps, but
 1723 in exchange permit to merge two priority queues in $O(\log n)$ *worst-case* time (as opposed to
 1724 expected time for meldable heaps).

1725 Both the structural and the order-property are quite different for binomial heaps compared
 1726 to binary heaps. To define it, we first need the concept of a *flagged tree* (the name was chosen
 1727 because it looks a bit like a conifer with a flag attached at its top). This is defined to be a binary
 1728 tree where every level is full, except that the root only has a left child; see also Figure 2.10.
 1729 Note that a flagged tree of height h has 2^h nodes; this will be crucial when merging two such
 1730 trees into one new one.

1731 **Definition 2.3.** A binomial heap is a list L of binary trees with the following properties:

- 1732 • **Binomial-heap structural property:** Any tree in L is a flagged tree.

We say that a binomial heap is proper if no two flagged trees in L have the same height.

See Figure 2.10 for an example.⁶ Note that there are no restrictions on the relationship of keys between two different flagged trees. Also note that keys in the right subtree of a node are specifically *allowed* to have bigger keys—see for example node 14 in the example, where the right child has key 15. However, it is still true that the maximum key within any flagged tree is at the root, because all other keys in that tree are in the left subtree of the root, hence required to be no bigger. It follows that we can easily find the maximum in the entire binomial heap, by scanning list L and finding the maximum among the keys at the roots.

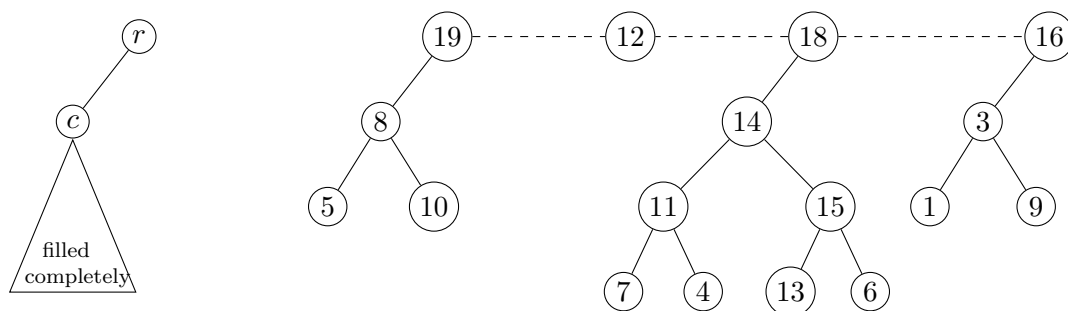


Figure 2.10: A flagged tree, and a binomial heap. (It is not proper because it has two flagged trees of height 2.)

In general the length $|L|$ of the list could be in $\Theta(n)$ (each element could be a single-node tree), but not if the binomial heap is proper.

Observation 2.2. *A proper binomial heap of size n contains at most $\log(n) + 1$ flagged trees.*

Proof. Among the flagged trees in the binomial heap, let T be the one that has the maximum height, say h . Then T has 2^h nodes, so $2^h \leq n$ and $h \leq \log n$. Since the trees in L all have distinct heights, we have at most one tree for each possible height $0, \dots, h$, and so at most $h + 1 \leq \log n + 1$ trees. \square

The key idea for making operations in binomial heaps efficient is hence to make the heap proper after every operation, and we study here first how to do this.

⁶The definition of binomial heap given here is *not* what you would find in many other textbooks on data structures. The other textbooks typically use *multi-way trees*, i.e., nodes to have more than two children. This means that the structure property is quite different but the order-property is then the same as for heaps. The two definitions are actually equivalent if one knows how to map multi-way trees to binary trees, but for our approach via flagged trees it will be more obvious to see why the run-times are logarithmic.

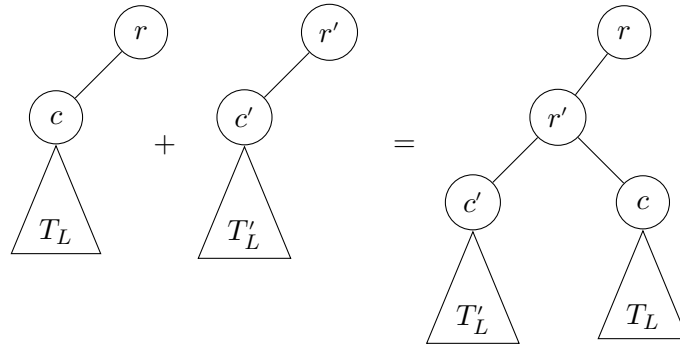


Figure 2.11: Combining two flagged trees of equal height if $r.key \geq r'.key$.

1752 **Making a binomial heap proper:** The idea for making a binomial heap proper is very
 1753 simple: If there are two flagged trees T, T' of the same height h , then they both have 2^h nodes.
 1754 We want to combine them into one flagged tree of height $h + 1$. To do so, compare the keys at
 1755 the roots r, r' , say the key at r is bigger. Make r' the left child of r , and move the former left
 1756 subtree T_L of T to become the right subtree of r' . See Figure 2.11. One easily verifies that the
 1757 order property is satisfied since r has the largest key of both trees. (Here it is crucial that keys
 1758 in T_L are *not* required to be upper-bounded by $r'.key$.)

1759 The only difficulty now is how to test efficiently whether there are trees of equal height in
 1760 list L . But since the heights are integers and of small values, we can do so by storing the flagged
 1761 trees in an array indexed by the height; if two flagged trees both want the same array-slot then
 1762 they have the same height, and so we immediately combine them. (We will see very similar ideas
 1763 when we do bucket-sort in Section 3.4.1 and direct addressing in Section 7.1.) Algorithm 2.11
 1764 gives the pseudo-code and Figure 2.12 gives an example.

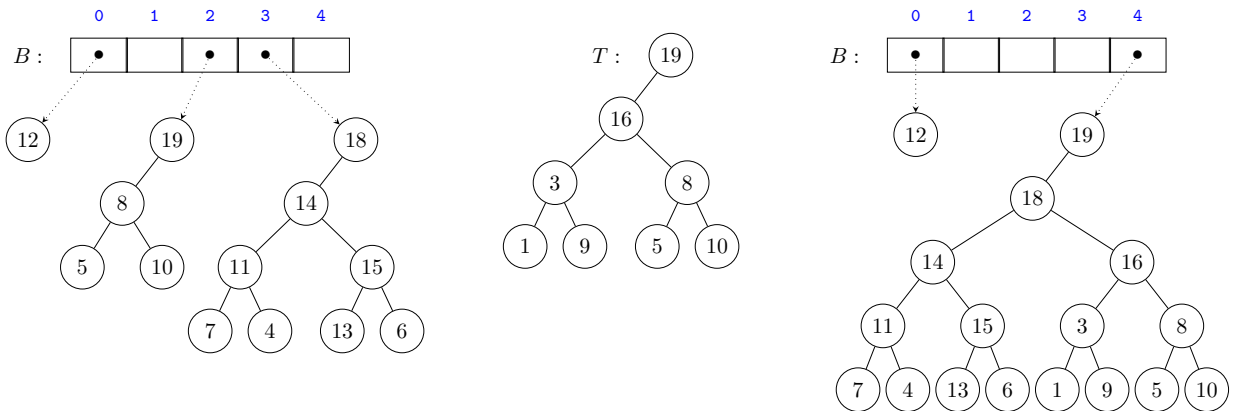


Figure 2.12: Making the binomial heap of Figure 2.10 proper. We show the status after adding the first three trees to B , the result of merging $B[2]$ and the fourth tree, and the final array.

Algorithm 2.11: *binomialHeap::makeProper()*

```

1  $n \leftarrow$  size of the binomial heap
2 for ( $\ell = 0; n > 1; n \leftarrow \lfloor n/2 \rfloor$ ) do  $\ell++$                                 // compute  $\lfloor \log n \rfloor$ 
3  $B \leftarrow$  array of size  $\ell + 1$ , initialized all-NIL
4  $L \leftarrow$  list of flagged trees
5 while  $L$  is non-empty do
6    $T \leftarrow L.pop()$ ,  $h \leftarrow T.height$ 
7   while  $T' \leftarrow B[h]$  is not NIL do
8     if  $T.root.key < T'.root.key$  then swap  $T$  and  $T'$ 
9      $T'.right \leftarrow T.left$ ,  $T.left \leftarrow T'$ ,  $T.height \leftarrow h+1$            // merge  $T$  with  $T'$ 
10     $B[h] \leftarrow$  NIL,  $h++$ 
11   $B[h] \leftarrow T$ 
12 for ( $h = 0; h \leq \ell; h++$ ) do                                           // copy  $B$  back to list
13  if  $B[h] \neq$  NIL then  $L.append(B[h])$ 

```

1765 The run-time for *merge* is $O(\log n + |L|)$: We spend $O(\log n)$ time for initializing the array
 1766 and then parse each element in L . Combining two trees takes $\Theta(1)$ time and reduces the total
 1767 number of trees, so the total time for all combining is $\Theta(|L|)$.

1768 **Implementing priority-queue operations:** With *makeProper* in place, all other priority-
 1769 queue operations can easily be performed in $O(\log n)$ time. We maintain that after each opera-
 1770 tion the binomial heap is proper, hence $|L| \in O(\log n)$ throughout.

- 1771 • *merge*(P_1, P_2): Simply concatenate the two lists of P_1 and P_2 into one. Then call *makeProper*.
 1772 This takes time $O(\log n_1 + \log n_2) \subseteq O(\log n)$.
- 1773 • *insert*(k, v): This is handled as for meldable heaps: Create a single-node binomial heap,
 1774 and then call *merge*. This takes time $O(\log n)$.
- 1775 • *findMax*(): Scan through list L and compare the keys of the roots. The largest among
 1776 them is the maximum. This takes time $O(|L|) \subseteq O(\log n)$.
- 1777 • *deleteMax*(): Assume we found the maximum at the root x_0 of flagged tree T , and let
 1778 h be the height of T . (We assume here that trees know their height, but if not, we can
 1779 also compute this in $O(\log n)$ time since $h \leq \log n$.) Remove T from list L and split it
 1780 as follows. Let x_1 be the left child of x_0 . For $1 \leq i \leq h$, let x_{i+1} be the right child of
 1781 x_i . Define T_i to be the tree consisting of x_i and its left subtree; this is a flagged tree. See
 1782 Figure 2.13.

1783 Create a second binomial heap P' consisting of T_1, T_2, \dots, T_h , and note that this covers
 1784 all keys in T except x_0 . Now merge P' into what remains of the original binomial heap
 1785 P . Since P was proper, and the list of P' has length $h \leq \log n$, merging (and with it
 1786 *deleteMax*) has run-time $O(\log n)$.

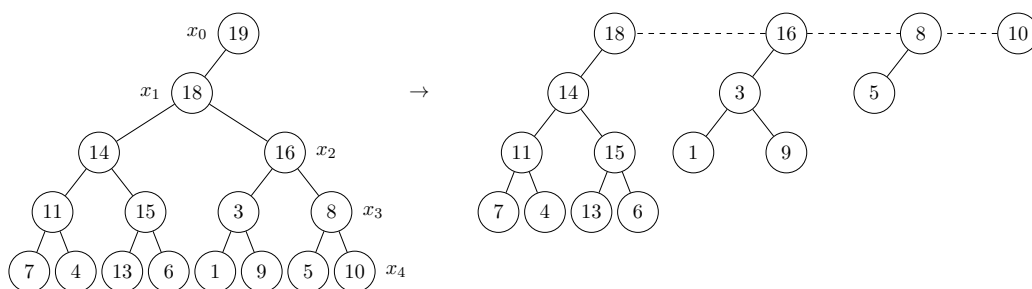


Figure 2.13: Removing key 19 results in a list of 4 flagged trees.

Summarizing, a binomial heap supports all priority-queue operations as well as merging in $O(\log n)$ worst-case time. (It also supports *decreaseKey*; details are left as an exercise.) It may be worth noting that some aspects of the implementation given here could be improved. In particular, since we are keeping the binomial heap proper at all times, there is no need to store L as a list—we might as well keep the array B for storing the flagged trees (by height) throughout, and save the copying back-and-forth when making the heap proper. This would also have the advantage that *insert* then has $O(1)$ *amortized run-time*, a concept that we will see later. Details are left as an exercise.

As far as the theoretical run-time is concerned, binomial heaps and binary heaps are equally good. However, in practice one would expect binary heaps to be faster, because they can store the key-value pairs in an array, and hence don't need the overhead of storing references to the left and right child at each node, as well as storing the list. So if *merge* is not a required operation, the binary heaps would be preferred, but if it is required then one should use binomial heaps.

2.5 Take-home messages

Let us summarize the most important insights from this chapter.

- A binary heap is a binary tree that stores key-value pairs and satisfies the heap-structure and the heap-order property.
- Heaps are one possible realization of priority queues. The worst-case run-time for *insert* and *deleteMax* is $O(\log n)$ if using heaps.
- A binary heap can be built in linear time.
- Combining the idea of priority-queue-sort with heaps, we can obtain a fast and easy-to-implement sorting algorithm *heap-sort*. This uses $O(1)$ auxiliary space and has $O(n \log n)$ worst-case run-time. It beats *merge-sort* in practice but is not stable.
- (cs240e) With some modifications (and after storing heaps as trees rather than arrays) we can also implement operation *merge* to merge priority queues. A minor modification gives $O(\log n)$ expected run-time, and a major modification gives $O(\log n)$ worst-case run-time for merging two priority queues of total size n .

It is worth mentioning that there are even more implementations of priority queues. None of them lead to a faster implementation of *priority-queue-sort*, but they offer other tradeoffs. In particular, a further change to binomial heaps gives *Fibonacci heaps*, which support operation *decreaseKey* with a run-time that is constant when averaged over all operations. But this is beyond the scope of cs240; see cs341 for application of finding shortest paths (which benefits from using Fibonacci-heaps) and cs466 for the details of how to implement and analyze them.

2.6 Historical remarks

The idea of binary heaps is from the 1960's, with Williams introducing the concept (as well as heapsort) [Wil64] and Floyd improving the run-time of building the heap to $O(n)$ [Flo64]. Meldable heaps implemented via randomization were analyzed by Gambien and Malinowski in 1998 [GM98], but the concept of merging heaps is much older: Binomial heaps were introduced by Vuillemin in 1978 [Vui78]. We briefly mentioned that binary heaps (if implemented as trees) can be used to achieve poly-logarithmic merging time; this was shown by Sack and Strothotte in 1985 [SS85]. There are numerous variants and extensions of priority queues; see for example Brodal's 2013 survey [Bro13] for further references.

Chapter 3

Sorting

Contents

3.1	The Selection problem	83
3.1.1	Partition	85
3.1.2	Algorithm <i>quick-select</i>	88
3.1.3	Average-case analysis and randomization	91
3.2	Algorithm <i>quick-sort</i>	97
3.2.1	Average-case analysis	99
3.2.2	Tips & Tricks for <i>quick-sort</i>	101
3.3	Lower Bound for Comparison-based Sorting	106
3.4	Sorting integers	109
3.4.1	Sorting by one digit	109
3.4.2	Sorting multi-digit numbers	111
3.5	Take-home messages	113
3.6	Historical notes	114

We have already seen the Sorting problem, and studied *merge-sort* and *heap-sort*, two algorithms to solve it efficiently. However, there are other sorting algorithms. You may have heard of *quick-sort*, which is generally reputed to be the fastest way to sort. We will study *quick-sort* in detail in this chapter, and analyze its running time. It turns out that while *quick-sort* is *not* the fastest algorithm in theory, one can argue that in average (over all possible input) it performs very well.

We will begin this exploration by first studying a closely related problem, Selection, which uses the same sub-routine but is easier to analyze; this will be a good warm-up.

3.1 The Selection problem

Definition 3.1. *The Selection problem is the following problem. Given an array $A[0..n-1]$ and an index k with $0 \leq k < n$, $\text{select}(A, k)$ should return the element in A that would be at index k*

1860 *if we sorted A.*

1861 So for example, if the array is as follows:

1862

0	1	2	3	4	5	6	7	8	9
30	60	10	0	50	80	90	10	40	70

1863 then *select*(3) should return 30, because the sorted array would contain the items 0, 10, 10,
 1864 30, 40, 50, 60, 70, 80, 90, and so the item at index 3 would be 30. Roughly speaking, the Selection
 1865 problem asks to return the *k*th smallest item in the array. We prefer to avoid describing it
 1866 this way for two reasons: First, most people would say that “2nd smallest” means that there is
 1867 exactly one item that is smaller. In contrast, since our arrays are indexed starting at 0, *select*(2)
 1868 returns an item where there are *two items* (at *A*[0] and *A*[1] if sorted) that are smaller than the
 1869 returned item (at *A*[2] if sorted). Second, it is not clear what “*k*th smallest” means if there are
 1870 duplicates (does the 2nd smallest of set {1, 1, 2, 2, 3, 3} have value 1 or value 2)? For this reason
 1871 we prefer the more convoluted expression “the item that would be at *A*[*k*] if we sorted *A*”.

1872 One special case of the Selection problem is the case $k = \lfloor n/2 \rfloor$, i.e., we want to find the
 1873 *median* of the given items. Executing *select*($\lfloor n/2 \rfloor$) is also called the *median-finding problem*,
 1874 with obvious applications in statistics.

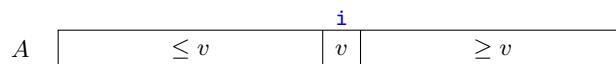
1875 There are some straightforward methods for implementing *select*(*A*, *k*):

- 1876 • Idea 1: Sort array *A*, then return *A*[*k*]. The run-time for this is $\Theta(n \log n)$ if we use
 1877 *heap-sort* or a similarly fast sorting-algorithm.
- 1878 • Idea 2: Build a heap *H* from *A* with *heapify*. Then delete the maximum of *H* $n - k + 1$
 1879 times. (Put differently, partially sort *A* by running the first $n - k$ iterations of *heap-sort*.)
 1880 This means that the $n - k$ items in *A*[$k \dots n-1$] are in sorted order, and the items in
 1881 *A*[$0 \dots k-1$] are no bigger than *A*[*k*]. Therefore we can return *A*[*k*] as the answer.
 1882 The run-time for this is $\Theta(n + (n - k) \log n)$, which is quite efficient if *k* is large. However,
 1883 for the median-finding problem where $k \approx n/2$, the run-time is $\Theta(n \log n)$, no better than
 1884 Idea 1.
- 1885 • Idea 3: This idea is symmetric to Idea 2, and works well if *k* is small. Build a *min-oriented*
 1886 heap *H* from *A*. Then delete the minimum of *H* $k + 1$ times. The last deleted item is the
 1887 answer.
 1888 Since deleting the minimum in a min-oriented heap can be done in $\Theta(\log n)$ time, the
 1889 run-time is $\Theta(n + k \log n)$. This is quite efficient when *k* is small, but no better than Idea
 1890 1 for median-finding.

1891 So we have multiple solutions, but none is better than $O(n \log n)$ in the worst case. We
 1892 are interested in finding algorithms that are *faster* (at least one average), and will do so in the
 1893 next few subsections. One should point out here that the improvement is not all that great (the
 1894 difference between $\Theta(n \log n)$ and $\Theta(n)$ is barely noticeably unless *n* is in the millions), but the
 1895 algorithm that we will develop is very simple, the sub-routine that it uses will be used again for
 1896 *quick-sort* later, and the analysis will be a good preparation for the analysis of *quick-sort* later.

3.1.1 Partition

We first study the sub-routine *partition*. This assumes that we are given a *pivot-value*, i.e., one element v of A which will be used to split the array. One very simple method of finding the pivot is to take the last element of A . (As it will turn out, this is *not* a good method, we will see better methods later, but it is good enough for explaining how partition works.) The sub-routine *partition* gets this pivot-value v and splits the array such that it has the following structure afterwards:



Put differently, it splits the items in A into those that are smaller than v , bigger than v , and equal to v . Then it rearranges A so that the smaller items go left v , the bigger ones go right of v , and the equal ones can go either way. There is *no* promise about the order within the left or the right part. Finally we return the index i where v got placed; this is called the *pivot-index* and will be vital when analyzing algorithms that use *partition* later. Note that if we have multiple copies of v in the input, then the pivot-index is not unique; in this case we accept any index where v could be as the pivot-index.

It is very easy to implement *partition* in $O(n)$ time, see Algorithm 3.1.

Algorithm 3.1: *partition*(A, p, n) // simple variant

Input : Array A , index p with $0 \leq p \leq n-1$

Output: The index i where $A[p]$ would be if A were sorted

```

1 Create empty lists smaller, equal and larger.
2  $v \leftarrow A[p]$  //  $v$  is the pivot-value.
3 for each element  $x$  in  $A$  do
4   if  $x < v$  then smaller.append( $x$ )
5   else if  $x > v$  then larger.append( $x$ )
6   else equal.append( $x$ )
7  $i \leftarrow \text{smaller.size}$ 
8  $j \leftarrow \text{equal.size}$ 
9 Overwrite  $A[0 \dots i-1]$  by elements in smaller
10 Overwrite  $A[i \dots i+j-1]$  by elements in equal
11 Overwrite  $A[i+j \dots n-1]$  by elements in larger
12 return  $i$ 
```

Hoare's partition routine: Clearly any implementation of *partition* must use $\Omega(n)$ time because we must look at every element of A . So our simple implementation of *partition* is as fast (asymptotically) as we can hope for. Nevertheless, there are implementations that work much

1916 better in practice. This is one of the places where the limitations of our method of analysis
 1917 (consider the worst-case run-time and bound it asymptotically) become apparent: In practice
 1918 it does make a difference how much space we are using and how often we compare or exchange
 1919 items.

1920 We explain here an implementation of *partition* that was developed by Hoare. There are
 1921 two reasons why it does much better than the naive implementation. First, it is *in-place*,
 1922 i.e., it uses only $O(1)$ auxiliary space. We saw the advantage of this already for *heap-sort*:
 1923 allocating and copying to new space does take significant time in practice and should be avoided
 1924 if possible. Secondly, Hoare's implementation handles items that are equal to v better. There
 1925 were no particular restrictions about where these items should go (left or right of v). Hoare's
 1926 implementation achieves that if all items are equal, then v will end up in the middle, something
 1927 that is helpful for the routines that use *partition*.

Algorithm 3.2: *partition*(A, p, n) // Hoare's more sophisticated approach

Input : Array A of size at least n , index p with $0 \leq p \leq n-1$
Output: The index i where $A[p]$ would be if A were sorted

```

1 swap  $A[n-1]$  and  $A[p]$            // stash pivot-value at the end of the array
2  $i \leftarrow -1, j \leftarrow n-1, v \leftarrow A[n-1]$ 
3 while () do
4   repeat
5      $i \leftarrow i + 1$ 
6   until  $A[i] \geq v$ 
7   repeat
8      $j \leftarrow j - 1$ 
9   until  $j < i$  or  $A[j] \leq v$ 
10  if  $i \geq j$  then
11    break (i.e., go to the final swap below)
12  else
13    swap  $A[i]$  and  $A[j]$ 
14 swap  $A[n-1]$  and  $A[i]$            // put pivot-value in its place
15 return  $i$ 
```

1928 The pseudocode for Hoare's partition-routine is in Algorithm 3.2, and an example is in
 1929 Figure 3.1. The disadvantage of Hoare's algorithm is that it is much harder to understand why
 1930 it works (and correspondingly it is very easy to make errors when coding it). The following is a
 1931 (very detailed) proof of correctness; you are encouraged to skip it and only look at the picture
 1932 of the invariant to understand the rough idea. But if you find yourself puzzled by some aspects
 1933 of the code (why do we initialize i with -1 , rather than 0 , for example?) then you might want
 1934 to study the proof in more detail.

$i=-1$	0	1	2	3	4	5	6	7	8	$j=9$
	30	60	10	0	50	80	90	20	40	$v=70$
	0	1	2	3	4	$i=5$	6	7	$j=8$	9
	30	60	10	0	50	80	90	20	40	$v=70$
	0	1	2	3	4	$i=5$	6	7	$j=8$	9
	30	60	10	0	50	40	90	20	80	$v=70$
	0	1	2	3	4	5	$i=6$	$j=7$	8	9
	30	60	10	0	50	40	90	20	80	$v=70$
	0	1	2	3	4	5	$i=6$	$j=7$	8	9
	30	60	10	0	50	40	20	90	80	$v=70$
	0	1	2	3	4	5	$j=6$	$i=7$	8	9
	30	60	10	0	50	40	20	90	80	$v=70$
	0	1	2	3	4	5	$j=6$	$i=7$	8	9
	30	60	10	0	50	40	20	70	80	90

Figure 3.1: Example of Hoare's partition routine.

Claim 3.1. *Hoare's partition-routine correctly partitions the array.*

Proof. (cs240e) We will prove this by giving a suitable while-loop-invariant, then proving that this holds in every execution of the loop and therefore implies the result when the loop terminates. The loop invariant is illustrated in the following picture:

	i	j	$n-1$	
A	$\leq v$?	$\geq v$	v

Put into words, all items in $A[0..i]$ are no bigger than v , and all items in $A[j..n-2]$ are no smaller than v . We also need to know that $i \leq n-1$ and $j \geq 0$. Furthermore, item v is at $A[n-1]$ and the array contains a permutation, i.e., exactly all elements of the original input. (These latter two conditions are obvious, so we will only prove the former ones.)

First observe that this loop invariant is true with the initial choice of $i = -1$ and $j = n-1$. Namely, $A[0..i] = A[0..-1]$ is empty, and so is $A[j..n-2] = A[n-1..n-2]$, so the claims hold vacuously.

Now we must show that the loop-invariant holds again the next time that we reach “while”. Let us study what happens in-between. The two repeat-loops increase i and decrease j until

1949 the first time that the until-condition is violated. Roughly speaking, we will have reached the
 1950 following situation:

1951

			i		j		$n-1$
A		$\leq v$	$\geq v$	$?$	$\leq v$	$\geq v$	v

1952 except that there are special cases when i gets too big or j too small. To put it into equations,
 1953 when we reach line 10, we know that $A[i] \geq v$ and either $j < i$ or $A[j] \leq v$. We also know that
 1954 $A[0..i-1] \leq v$ and $A[j+1..n-1] \geq v$ since this held in the loop invariant and continues to hold as
 1955 we change i and j since we explicitly check for it. Finally observe that $j \leq n-1$ throughout, since
 1956 this holds initially and we only decrease j . Also $i \geq 0$ as soon as we enter the first repeat-loop,
 1957 and so $i \geq 0$ throughout. We now have multiple possibilities:

- 1958 • $i < j$, and therefore $0 \leq i < j \leq n-1$. We reached this point because $A[i] \geq v \geq A[j]$. We
 1959 swap these two items, and therefore the loop invariant holds again.
- 1960 • $i \geq j$. Observe that $i \leq n-1$, because once i reaches $n-1$, we have $A[i] = v \geq v$ and the
 1961 repeat-loop for i stops. We have sub-cases:
 - 1962 – $j = i$. Since the repeat-loop for i stopped, we have $A[i] \geq v$. Since the repeat-loop
 1963 for j stopped with $j \geq i$, we have $A[j] \leq v$. By $i = j$ therefore $A[i] = A[j] = v$.
 1964 We also know $A[0..i-1] \leq v$ and $A[j+1..n-1] \geq v$, so the partition of A is correct
 1965 with i as breakpoint. We do one swap to bring the original pivot-element into place
 1966 $A[i]$. (This step would not be needed if we are only sorting integers, but there may
 1967 be values associated with the integers, so we should put the original pivot-item into
 1968 $A[i]$.) Then we have achieved the desired outcome.
 - 1969 – $j < i$. We know $A[0..i-1] \leq v$. We also know $A[j+1..n-1] \geq v$, so in particular
 1970 $A[i..n-1] \geq v$. Therefore A is partitioned correctly. Swapping $A[i]$ and $A[n-1]$ does
 1971 not change this (and brings the pivot-item to the correct place), so we have achieved
 1972 the desired outcome.

□

1974 3.1.2 Algorithm *quick-select*

1975 With the *partition* routine in place, let us return to the selection problem. Recall that we want
 1976 the item in A that would be at $A[k]$ if A were sorted; we call this item m . Finding m would be
 1977 very easy if we re-arranged the array so that it looks as follows

1978

			k
A	$\leq m$	m	$\geq m$

1979 because then we can simply return $A[k]$. This looks a *lot* like the outcome of partition

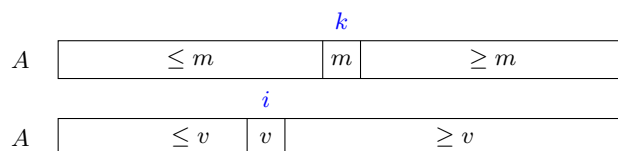
1980

A	$\leq v$	v	$\geq v$
-----	----------	-----	----------

except that *partition* gives the user no control over the resulting index i , and so applying it once is not enough to find the desired item m . But, once we have done *partition* (with some pivot-value—we will discuss this choice below), the array has enough structure that we can exclude lots of items from further search. Consider three cases:

- If $i = k$, then we were lucky and got exactly the right split: Return v as the solution.
- If $i > k$, then there are i items in A that are at most v (namely, the items in $A[0..i-1]$, i.e., to the left of the pivot-value). Since $i > k$, this means that the desired selection-value m is somewhere on the left side, and in particular $m \leq v$. So it suffices to search among the items on the left side for m .
- If $i < k$, then symmetrically we should recurse on the right side to find m .

The pseudo-code is given in Algorithm 3.3. A few things are worth pointing out. First, the algorithm assumes the existence of a routine *choose-pivot*, which returns the index p in $\{0, \dots, n-1\}$ that we use to find the pivot. For now we assume that *choose-pivot* simply returns $n-1$, i.e., the pivot-value is the last item in the array. (This turns out to be a bad choice; we will discuss better choices much later, in Section 3.2.2.) Second, we have chosen to keep the description simple and hence pass entire arrays into sub-routines. As discussed in Step 3 of Section 1.4.6, one should pass only the limits (left and right boundary) of the array in consideration, and refer to the original array when comparing values and swapping items. This makes the code much less easy to read (which is why we don't show this here), but is much faster in practice. Third, why do we pass " $k - (i+1)$ " as index to search for in the second recursion, rather than k ? Let us inspect the desired and the actual situation if $i < k$:



We know that the desired item m is in $A[i+1..n-1]$ of the array, and hence recurse in this sub-array. But this means that the $i + 1$ items in $A[0..i]$ are *not* passed into the recursion. So we must subtract $i + 1$ from the desired index k .

Algorithm *quick-select* has one accidental but useful outcome. We rearrange the array such that all items in $A[k+1..n-1]$ are no smaller than $A[k]$. Therefore, the $k + 1$ smallest items are exactly the items in $A[0..k]$. Therefore *quick-select* does not only find one item at a specific index, we can in the same run-time find the entire set of smallest items (where we can choose what size of set we want).

Analysis of *quick-select*

Let us first take a quick look at the auxiliary space used by *quick-select*. At first look, this appears to be $O(1)$, because we have only a constant number of variables, and both *choose-pivot* and *partition* use $O(1)$ auxiliary space. But the code hides another source of auxiliary space use: recursions. Every recursive call requires that the current status of variables gets stored

Algorithm 3.3: *quick-select*($A, k, n \leftarrow A.size$)

Input : Array A of size n , index k with $0 \leq k \leq n-1$

```

1  $p \leftarrow \text{choose-pivot}(A)$ 
2  $i \leftarrow \text{partition}(A, p)$ 
3 if  $i = k$  then
4   | return  $A[i]$ 
5 else
6   | if  $i > k$  then
7     | return quick-select( $A[0, 1, \dots, i-1], k$ )
8   | else
9     | return quick-select( $A[i+1, i+2, \dots, n-1], k-i-1$ )
```

until the recursion returns, and this takes $O(1)$ space per stacked recursive call. This could be as much as $\Theta(n)$ auxiliary space! It turns out that this can be reduced easily to $\Theta(1)$ by using *tail-recursion* (replace the recursion by a suitable while-loop). We will not give the details here (but see the discussion in Section 3.2.2 for similar techniques). So with a suitable variant of the code, the auxiliary space is $\Theta(1)$.

Now we analyze the run-time of *quick-select*. As in many earlier examples, we count the number of *key-comparisons* (“is $A[i] > v$?”), because this is a good proxy for run-time and it has the advantage that we can do an exact count, rather than needing to deal with constants to avoid doing arithmetic with asymptotic notation.

One easily sees that for $n \geq 2$ Hoare’s *partition* routine takes exactly n key-comparisons: For as long as $i < j$, we have compared each item in $A[0..i]$ and each item in $A[j..n-2]$ exactly once with pivot-value v . We exit the loop when $j = i$, and so $A[j] = A[i]$ was compared twice with v , giving n key-comparisons in total.

Let $T(A)$ be the number of key-comparisons used when running *quick-select* on an array A with n items. (Technically we should make $T(A)$ also dependent on the parameter k , but since we take upper bounds it will turn out that k is irrelevant.) We can express $T(A)$ as follows:

$$T(A) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + T(\text{size of sub-array that we recurse in}) & \text{if } n \geq 2 \end{cases}$$

Let us use A' to denote the sub-array that we recurse in. The size of A' could be as much as $n-1$ (if $i = 0$ or $i = n-1$), but not more. Hence the worst-case run-time satisfies the recurrence $T^{\text{worst}}(n) \leq T^{\text{worst}}(n-1) + n$. It is easy to see to show by induction that this is at most $\sum_{\ell=1}^n \ell = \frac{n(n+1)}{2} \in O(n^2)$. So the worst-case run-time is $O(n^2)$, and this is tight if the input A is sorted and we are using $k = 0$, because then the sub-array A' always has size $n-1$.

In the best case, the desired output value was at the last position in the array, hence chosen as pivot-value, and so after the first call to *partition* we have $i = k$ and immediately stop. So in the best case, the run-time is $\Theta(n)$.

3.1.3 Average-case analysis and randomization

We now want to analyze the average-case number of key-comparisons used by *quick-select* (the average-case run-time is proportional to this). Recall that this is defined as $\frac{1}{|\mathcal{I}_n|} \sum_I T(I)$ where $T(I)$ is the number of key-comparisons used for one instance I . For the Selection problem, each instance consists of an array A and the index k of the item that we are selecting. Note that there are infinitely many possible arrays (because there are infinitely many numbers), so we again use the trick of mapping instances to sorting permutations that we already encountered in Section 1.5.1. So instances are described as $\langle \pi, k \rangle$ where $k \in \{0, \dots, n-1\}$, and therefore the average-case run-time is defined as $T^{\text{avg}}(n) = \frac{1}{n!} \frac{1}{n} \sum_{\pi} \sum_k T(\langle \pi, k \rangle)$. As it turns out, there is no need to keep the parameter k separate. Specifically, we instead analyze

$$\max_k \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\langle \pi, k \rangle) =: T_{\text{max-}k}^{\text{avg}}(n)$$

(i.e., taking the maximum over all k but the average over all π), show that this is in $O(n)$, and then of course $T^{\text{avg}}(n) \leq T_{\text{max-}k}^{\text{avg}}(n) \in O(n)$ as well.

Let us study one particular permutation π , and observe that this determines the pivot-index i . (We assume throughout that we use the last item in A as pivot-value; hence i is the index of $A[n-1]$ if we sorted A , so $\pi(i) = n-1$, so $i = \pi^{-1}(n-1)$.) Let π' be the sorting permutation of the sub-array into which we recurse. We know that π' has size i or $n-i-1$, and so one would be tempted to claim that $T(\langle \pi, k \rangle) \leq n + \max\{T_{\text{max-}k}^{\text{avg}}(i), T_{\text{max-}k}^{\text{avg}}(n-i-1)\}$. Alas, as was the case for *avgCaseDemo* in Section 1.5.1, this formula is not correct because we do not know whether π' hits the average case. But again it is the case that when summing over all possible such permutations, we do reach the average. To state this precisely, define $\Pi_n(i)$ to be all those permutations π with $\pi(i) = n-1$.

Lemma 3.1. *If $n \geq 3$ then for any $0 \leq i, k \leq n-1$ we have*

$$\sum_{\pi \in \Pi_n(i)} T(\langle \pi, k \rangle) \leq |\Pi_n(i)| \left(n + \max\{T_{\text{max-}k}^{\text{avg}}(i), T_{\text{max-}k}^{\text{avg}}(n-i-1)\} \right).$$

We will not give a proof of this lemma; it is proved somewhat similar to Lemma 1.6, but much more complicated. If you believe this lemma, then the rest of the analysis is relatively straightforward. We will need to know the size of $\Pi_n(i)$.

Observation 3.1. *For any $0 \leq i, j < n-1$ we have $|\Pi_n(i)| = (n-1)!$*

Proof. Let $\pi \in \Pi_n(i)$. This means $\pi(i) = n-1$ and so one item of π is fixed. The remaining items are $\{0, \dots, n-2\}$ and their order is arbitrary, so there are $(n-1)!$ choices for π . \square

2056 Combining this lemma and observation, we get a recursive formula for $T_{\max-k}^{\text{avg}}(n)$:

$$\begin{aligned}
 T_{\max-k}^{\text{avg}}(n) &= \max_k \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\langle \pi, k \rangle) = \max_k \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{\pi \in \Pi_n(i)} T(\langle \pi, k \rangle) \\
 &\leq \max_k \frac{1}{n!} \sum_{i=0}^{n-1} |\Pi_n(i)| \left(n + \max \{ T_{\max-k}^{\text{avg}}(i), T_{\max-k}^{\text{avg}}(n-i-1) \} \right) \quad (\text{independent of } k) \\
 &= n + \frac{1}{n!} \sum_{i=0}^{n-1} (n-1)! \cdot \max \{ T_{\max-k}^{\text{avg}}(i), T_{\max-k}^{\text{avg}}(n-i-1) \} \\
 &= n + \frac{1}{n} \sum_{i=0}^{n-1} \max \{ T_{\max-k}^{\text{avg}}(i), T_{\max-k}^{\text{avg}}(n-i-1) \}.
 \end{aligned}$$

2057 We will show later (in Lemma 3.4) that this recursion resolves to $O(n)$. Therefore (assuming
 2058 you believe Lemma 3.1), the average-case run-time of *quick-select* is $O(n)$. This is tight since
 2059 *partition* requires n key-comparisons.

2060 **Randomizing *quick-select*:** If you're not comfortable with 'believing' Lemma 3.1, then we now
 2061 give a different way to analyze the average-case run-time, by doing a detour into a randomized
 2062 version of *quick-select*. (You may want to re-read Section 1.5.2 for some background.) As was
 2063 mentioned there, it is *not* obvious that a randomized version of *quick-select* can be used to bound
 2064 the average-case run-time; we will argue this carefully for the Selection problem (and a suitable
 2065 randomization) later.

2066 We first motivate why randomization is a useful strategy for *quick-select* in general (i.e.,
 2067 beyond the purpose of doing average-case analysis). There are some instances where *quick-select*
 2068 performs badly, and (many) others where it performs well. If you find that an algorithm typically
 2069 performs well (as is the case for *quick-select*) then one good idea is to change the input instance
 2070 randomly so that all instances become equally likely. (Of course we must do this in such a way
 2071 that the end-result stays the same.) If we can do this, then we shift the bad case from having
 2072 bad instances (over which we have no control) to having bad luck.

2073 So how can we randomize *quick-select*? One idea is to *shuffle* the input. Recall that the
 2074 input is characterized via its sorting-permutation. If we permute the input via a randomly
 2075 picked permutation, then the sorting-permutation of the resulting array will be random, and
 2076 so all instances are equally good. Algorithm 3.4 shows the *inside-out method* of shuffling an
 2077 array randomly. (It is not trivial that the resulting array is equally likely to have any of the $n!$
 2078 permutations as a sorting permutations; we will leave the details as an exercise.)

2079 While this approach works, there is a simpler way to achieve the same result. Observe that
 2080 for as the index of the for-loop of *inside-out-shuffle* is $i \leq n-2$, item $A[n-1]$ will not get
 2081 swapped. In the last execution of the for-loop, we hence swap the original $A[n-1]$ with some
 2082 randomly chosen item of A (possibly itself). Then later in *partition*, we use this randomly chosen
 2083 item of A as the pivot-value. We might as well avoid calling *inside-out-shuffle* at all, and instead

Algorithm 3.4: *inside-out-shuffle*($A, n \leftarrow A.size$)**Input** : Array A of size n **Output:** The sorting permutation of A is equally likely to be any of the $n!$ permutations

```

1 for  $i \leftarrow 1$  to  $n-1$  do      // Pick one previous item (or itself) to switch with
2    $\lfloor$  swap(  $A[i]$ ,  $A[random(i+1)]$  )

```

2084 choose the pivot-value directly and randomly among the items. In other words, we choose the
 2085 index p that we use for looking up the pivot-value randomly in $\{0, \dots, n-1\}$. See Algorithm 3.5.

Algorithm 3.5: *randomized-quick-select*($A, k, n \leftarrow A.size$)**Input** : Array A of size n , index k with $0 \leq k \leq n-1$

```

1  $p \leftarrow random(n)$ 
2  $i \leftarrow partition(A, p)$ 
3 if  $i = k$  then
4    $\lfloor$  return  $A[i]$ 
5 else
6   if  $i > k$  then
7      $\lfloor$  return randomized-quick-select( $A[0, 1, \dots, i-1], k$ )
8   else
9      $\lfloor$  return randomized-quick-select( $A[i+1, i+2, \dots, n-1], k-i-1$ )

```

Analyzing the expected run-time: Let us develop a recursive formula for the expected number of key-comparisons taken by *randomized-quick-select*; we denote this by $T^{\text{exp}}(n)$. Since we choose the index for the pivot randomly, the pivot-value v is randomly chosen among the n items in A , and the pivot-index i is equally likely to be any of $\{0, \dots, n-1\}$. In other words,

$$P(\text{pivot-index is } i) = \frac{1}{n}.$$

For *randomized-quick-select*, an instance consists of $\langle A, k \rangle$, where A is the input array and k is the index of the item that we want to select. We need to analyze $T(\langle A, k \rangle, R)$, where R is a sequence of random outcomes. For *randomized-quick-select*, R can be described as $\langle i, R' \rangle$, where i is the pivot-index while R' is the sequence of random outcomes for the recursions. With this, we can give a formula for the number of key-comparisons:

$$T(\langle A, k \rangle, \langle i, R' \rangle) = \begin{cases} n & \text{if } k = i \\ n + T(\langle A_\ell, k \rangle, R') & \text{if } k < i \\ n + T(\langle A_r, k-i-1 \rangle, R') & \text{if } k > i \end{cases}$$

where $A_\ell := A[0 \dots i-1]$ and $A_r := A[i+1 \dots n-1]$ are the left and right sub-array of A after the partition. Note that A_ℓ and A_r have size i and $n-i-1$, respectively so again it is tempting to replace $T(\cdot)$ in the right-hand side of this equation by $T^{\text{exp}}(i)$ and $T^{\text{exp}}(n-i-1)$, respectively. This is incorrect when looking at one particular set of A, k, i, R' , but correct if we consider all of them.

Lemma 3.2. *If $n \geq 2$, then*

$$T^{\text{exp}}(n) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{\text{exp}}(i), T^{\text{exp}}(n-i-1)\}.$$

Proof. The proof is a lengthy (but straightforward) manipulation of the definition of $T^{\text{exp}}(n)$. The crucial ingredient is that that definition of $T^{\text{exp}}(\cdot)$ uses the *maximum* over all possible instances; therefore the time for the recursion (e.g. $T(\langle A_\ell, k \rangle)$) can be upper-bounded by the maximum needed for some A' of the same size and some k' . To make this precise, let us write \mathcal{A}_n for the (infinite) set of arrays with n items. Then

$$\begin{aligned} T^{\text{exp}}(n) &= \max_{\text{instances } \langle A, k \rangle} \sum_{\text{outcomes } R=\langle i, R' \rangle} P(R) \cdot T(\langle A, k \rangle, R) \\ &= \max_{A \in \mathcal{A}_n} \max_k \sum_{i=0}^{n-1} \sum_{R'} P(\text{pivot-index } i) \cdot P(R') \cdot T(\langle A, k \rangle, \langle i, R' \rangle) \\ &= \max_{A \in \mathcal{A}_n} \max_k \left\{ \sum_{i=0}^{k-1} \sum_{R'} \frac{1}{n} \cdot P(R') \cdot \left(n + T(\langle A_r, k-i-1 \rangle, R') \right) \right. \\ &\quad \left. + \sum_{i=k+1}^{n-1} \sum_{R'} \frac{1}{n} \cdot P(R') \cdot \left(n + T(\langle A_\ell, k \rangle, R') \right) \right\} \\ &\leq \max_{A \in \mathcal{A}_n} \max_k \left\{ n + \frac{1}{n} \left(\sum_{i=0}^{k-1} \max_{A' \in \mathcal{A}_{n-i-1}} \max_{k'} \sum_{R'} P(R') \cdot T(\langle A', k' \rangle, R') \right) \right. \\ &\quad \left. + \sum_{i=k+1}^{n-1} \max_{A' \in \mathcal{A}_i} \max_{k'} \sum_{R'} P(R') \cdot T(\langle A', k' \rangle, R') \right\} \\ &= \max_{A \in \mathcal{A}_n} \max_k \left\{ n + \frac{1}{n} \left(\sum_{i=0}^{k-1} T^{\text{exp}}(n-i-1) + \sum_{i=k+1}^{n-1} T^{\text{exp}}(i) \right) \right\} \\ &\leq \max_{A \in \mathcal{A}_n} \max_k \left\{ n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{\text{exp}}(i), T^{\text{exp}}(n-i-1)\} \right\} \end{aligned}$$

which gives the result since the term inside the curly brackets depends on neither A nor k . \square

As we will argue below (in Lemma 3.4), this recursion resolves to $O(n)$, and therefore the expected run-time of *randomized-quick-select* is $O(n)$. This is tight since *partition* uses n comparisons.

If we are really unlucky in our random draws, then we could still have run-time $\Theta(n^2)$. (For example consider the situation where we searching for the smallest element, but the randomly chosen pivot-value always happens to be the largest element.) However, this is very unlikely (and independent of the input instance).

From expected to average-case (for Sorting and Selection): Recall that what we really wanted to analyze was the *average-case* run-time for *quick-select*, i.e., without randomization. As we now show, this is the same as the expected run-time of *randomized-quick-select*, and in particular therefore, is in $O(n)$. We prove a slightly more general statement here, because it will be useful later for *quick-sort* as well.

Lemma 3.3. *Let \mathcal{A} be a (non-randomized) algorithm that solves Selection or Sorting. Let \mathcal{B} be the following randomized algorithm:*

- *Given: instance I (an array)*
- *Apply inside-out-shuffle to convert I into I'*
- *Call algorithm \mathcal{A} on I*

Then \mathcal{B} solves the same problem as \mathcal{A} and $T_{\mathcal{A}}^{\text{avg}}(n) = T_{\mathcal{B}}^{\text{exp}}(n)$, presuming $T(\cdot)$ counts key-comparisons.

Proof. (cs240e) Correctness of \mathcal{B} is obvious, because for Selection and Sorting, the order of items in the input-array is irrelevant. So let us analyze the run-times. Recall that instance I is characterized via its sorting-permutation; call it π . Let us use σ to denote the permutation that was applied during *inside-out-shuffle*, i.e., $I' = \sigma(I)$. Hence I' has sorting permutation $\pi \circ \sigma^{-1}$. Now

$$T_{\mathcal{B}}^{\text{exp}}(n) = \max_{\pi} \sum_{\sigma \in \Pi_n} P(\text{inside-out-shuffle creates } \sigma) \cdot T_{\mathcal{B}}(\sigma, \pi) = \max_{\pi} \sum_{\sigma \in \Pi_n} \frac{1}{n!} \cdot T_{\mathcal{A}}(\pi \circ \sigma^{-1})$$

since once we have determined the random outcome σ , the rest of algorithm \mathcal{B} is deterministic and its run-time is the run-time of \mathcal{A} for the new sorting permutation. Now observe that σ goes over all permutations while (inside the maximum) π is one fixed permutation. In consequence $\pi \circ \sigma^{-1}$ also goes over all permutations, and we can do a change of variables with $\tau = \pi \circ \sigma^{-1}$:

$$T_{\mathcal{B}}^{\text{exp}}(n) = \max_{\pi} \sum_{\sigma \in \Pi_n} \frac{1}{n!} T_{\mathcal{A}}(\pi \circ \sigma^{-1}) = \max_{\pi} \frac{1}{n!} \sum_{\tau \in \Pi_n} T_{\mathcal{A}}(\tau) = \max_{\pi} T_{\mathcal{A}}^{\text{avg}}(n) = T_{\mathcal{A}}^{\text{avg}}(n)$$

since $T_{\mathcal{A}}^{\text{avg}}(n)$ is independent of π . □

Because we defined *randomized-quick-select* carefully so that it mimics the effect of *inside-out-shuffle*, the average-case run-time of *quick-select* equals the expected run-time of *randomized-quick-select*, and hence is in $\Theta(n)$.

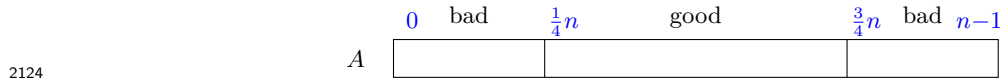
2116 **Resolving the recursion** Whether you believed Lemma 3.1 or went via randomization, in
 2117 both cases we ended with the same recursive formula. It remains to show that this formula
 2118 resolves to $O(n)$.

Lemma 3.4. *The recursive function $T(\cdot)$ defined by $T(1) = 1$ and*

$$T(n) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\}$$

2119 *is in $O(n)$.*

2120 *Proof.* We will show by induction that $T(n) \leq 8n$. Clearly this holds for $n = 1$ so assume $n > 1$.
 2121 For ease of writing, let us assume that n is divisible by 4 (the proof works similarly by applying
 2122 appropriate rounding brackets otherwise). Call an index i in $\{0, \dots, n-1\}$ *good* if it falls within
 2123 the range $[\frac{1}{4}n, \frac{3}{4}n)$ and *bad* otherwise, and note that half of the indices are good.



2125 If i is good, then $\max\{i, n-i-1\} \leq \frac{3}{4}n$. If i is bad then we use as (weak) upper bound
 2126 $\max\{i, n-i-1\} \leq n$. Therefore the recursion can be upper-bounded by

$$\begin{aligned}
 T(n) &\leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\} \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{8i, 8(n-i-1)\} \\
 &= n + \frac{1}{n} \left(\sum_{i:i \text{ is good}} 8 \cdot \frac{3}{4}n + \sum_{i:i \text{ is bad}} 8n \right) \\
 &= n + \frac{1}{n} (\#\{\text{good indices}\} \cdot 6n + \#\{\text{bad indices}\} \cdot 8n) \\
 &\leq n + \frac{1}{n} \left(\frac{n}{2} \cdot 6n + \frac{n}{2} \cdot 8n \right) = n + 3n + 4n = 8n
 \end{aligned}$$

2127 as desired. □

2128 This proof gives a bit of the intuition of why $T(n)$ is linear. For half of the pivot-indices, the
 2129 situation is *good* in the sense that the recursive array is a constant fraction smaller (which leads
 2130 to an overall linear time bound). One could drive this further: for 90% of the pivot-indices, the
 2131 situation is such that the recursive array has size at most $\frac{95}{100}n$, which *still* leads to an overall
 2132 linear time bound (with a much bigger constant). So the number of pivot-indices that lead to
 2133 quadratic run-time is vanishingly small.

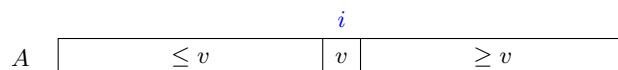
Summarizing *quick-select*: So in summary, we have now shown that *randomized-quick-select* has expected run-time $O(n)$, and also shown that this implies that *quick-select* has average-case run-time $O(n)$. Both bounds are tight, because we do at least one call to *partition*, which takes $\Omega(n)$ time.

Remember that in the worst case, *quick-select* is not fast (it could have run-time $\Theta(n^2)$). Whether this worst case happens depends much on how we select the pivot, which we will study in Section 3.2.2. The implementation that is fastest in practice seems to be to pick the pivot randomly, i.e., to use *randomized-quick-select*.

It is worth mentioning that there are ways to modify *quick-select* so that even the worst-case run-time becomes $O(n)$. These are, however, quite complicated and much much slower in practice. You may encounter this in cs341 (where it is studied mostly out of academic interest and for seeing unusual recursive algorithm), but for cs240 we treat *quick-select* as an algorithm where the worst-case run-time is $\Omega(n^2)$.

3.2 Algorithm *quick-sort*

Now we get to the main topic of this section, the sorting-algorithm *quick-sort*. This also uses *partition*, and is based on the following insight: After we have executed *partition*, we have re-arranged the array as follows:



This means that v has been put into the place where it should be when A is sorted. (If there are multiple copies of v then there are multiple places that v could be, but the one that we put it into is definitely correct.) So to sort the entire array, it suffices to recurse on *both* sides of the pivot-index. This is in contrast to *quick-select*, where we recursed on only one side. Algorithm 3.6 gives the pseudocode for *quick-sort*. (This is a ‘quick-and-dirty’ version; we will see a refined version in Section 3.2.2.)

Algorithm 3.6: *quick-sort*($A, n \leftarrow A.size$)

Input : array A of size n

1 **if** $n \leq 1$ **then** return

2 $p \leftarrow \text{choose-pivot}(A)$

3 $i \leftarrow \text{partition}(A, p)$

4 *quick-sort*($A[0, 1, \dots, i-1]$)

5 *quick-sort*($A[i+1, \dots, n-1]$)

To analyze the run-time for *quick-sort*, we again count the number of key-comparisons. As for *quick-select*, this depends on the pivot-index i returned by *quick-select*; we recurse in a sub-array of size i and another of size $n-i-1$.

Sometimes it is helpful to visualize the execution of *quick-sort* via its *recursion tree*. This is a binary tree defined by the subproblems that we are trying to solve; we label each node with the size of the sub-problem and create children for each sub-problem that is called recursively. See Figure 3.2. The height of this tree is the same as the *recursion depth*, i.e., the maximum number of recursions that are stacked inside each other at some point of the execution of *quick-sort*.

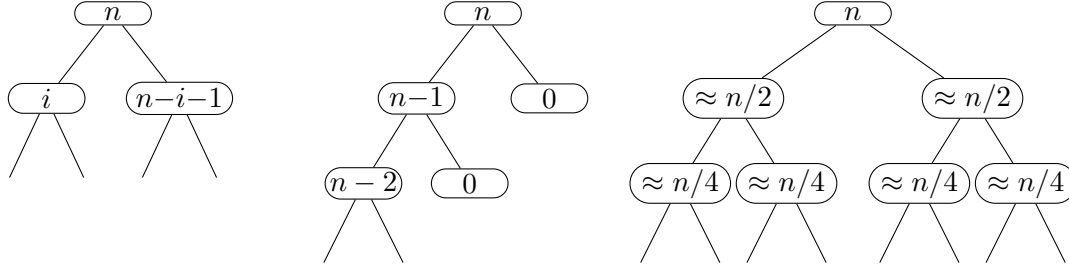


Figure 3.2: The topmost level of the recursion tree for *quick-sort*, and the recursion trees in the worst and best case.

On each level of the recursion tree we use at most n key-comparisons, since there are at most n items in total. Thus the run-time for *quick-sort* is at most the height of the recursion-tree, times n . With this we can easily prove upper bounds on the run-times, and it is not hard to show that they are tight.

Lemma 3.5. *The worst-case run-time for quick-sort is in $\Theta(n^2)$.*

Proof. The height of the recursion-tree is at most n , and we use at most n key-comparisons on each level, so the worst-case run-time is $O(n^2)$. This is tight if the array is sorted, because then already *quick-select* uses $\Omega(n^2)$ comparisons, and *quick-sort* can be no faster. \square

Lemma 3.6. *The best-case run-time for quick-sort is in $\Theta(n \log n)$.*

Proof. Consider the instances where the pivot-index is always roughly in the middle. Then the size of the array gets cut in half with each recursion, which means that the recursion tree has height $\approx \log n$ and therefore the run-time for this instance is in $O(n \log n)$.

(cs240e) Roughly speaking, this is tight because the recursion tree has height $\Omega(\log n)$ and uses roughly n key-comparisons on each level. But we need to be a bit more precise here because ‘ n ’ is an upper bound, while we need to prove a lower bound here. So consider the recursion tree. The subproblems on the first level of the recursion tree have $n-1$ items in total (in some unknown split). On the second level of recursion, each subproblem “sheds” one item and splits into at most two more problems, so there are at least $n-3$ items and at most 4 subproblems. Iterating the argument, one sees that on level ℓ of the recursion tree, there are at least $n - (2^\ell - 1)$ items in total, split among at most 2^ℓ subproblems. Doing *partition* in all these subproblems takes at least $n - 2^\ell + 1$ key-comparisons in total. Setting $\ell := \lfloor \log n \rfloor - 1$, we have $2^\ell \leq n/2$,

and therefore at least $n - 2^\ell + 1 \geq n/2 + 1$ key-comparisons are done on each of layer $0, 1, \dots, \ell$. So there are $\Omega(n)$ key-comparisons on each of $\Omega(\log n)$ levels, and in total we used $\Omega(n \log n)$ key-comparisons. \square

3.2.1 Average-case analysis

Algorithm *quick-sort* is very popular (and works very well in practice), even though its worst-case run-time is asymptotically worse than *heap-sort* or *merge-sort*. We can justify its popularity with a theoretical result: its average-case run-time is good.

Theorem 3.1. *Algorithm quick-sort has average-case run-time $\Theta(n \log n)$.*

We already saw that its best-case run-time is on $\Omega(n \log n)$, and the average-case run-time can be no better, so we only need to show the upper bound. As for *quick-select*, trying to analyze the average-case run-time directly will lead to some problems, because it is not obvious (though correct) that for the sub-arrays again the average case applies. Therefore we will again do the detour via randomization. Formally, let *randomized-quick-sort* be the same as *quick-sort* except that the index p for the pivot-value is picked to be *random*(n). In what follows, we will analyze the expected run-time of *randomized-quick-sort*, and by Lemma 3.3 the bound then also holds for the average-case run-time of *quick-sort*.

Let us develop a recursive formula for $T^{\text{exp}}(n)$, the expected number of key-comparisons for *randomized-quick-sort* in a size- n array. As for *randomized-quick-select*, we know that the probability that the pivot-index has value i is $\frac{1}{n}$ for any $0 \leq i \leq n-1$. For a given pivot-index i , we spent n key-comparisons on *partition* and then recurse in sub-arrays of size i and $n-i-1$, respectively. Similarly to what we did in Lemmas 1.7 and 3.2, a straightforward but tedious analysis shows that

$$T^{\text{exp}}(n) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} (T^{\text{exp}}(i) + T^{\text{exp}}(n-i-1)).$$

(We leave the details to the reader.) Therefore

$$\begin{aligned} T^{\text{exp}}(n) &\leq n + \frac{1}{n} \sum_{i=0}^{n-1} (T^{\text{exp}}(i) + T^{\text{exp}}(n-i-1)) \\ &= n + \frac{2}{n} \sum_{i=0}^{n-1} T^{\text{exp}}(i) = n + \frac{2}{n} \sum_{i=2}^{n-1} T^{\text{exp}}(i) \quad (\text{since } T(0) = T(1) = 0) \end{aligned}$$

Theorem 3.1 holds if we show the following:

Lemma 3.7. *The recursive function $T(\cdot)$ defined as $T(0) = T(1) = 0$ and $T(n) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} T(i)$ satisfies $T(n) \leq 2n \ln n \approx 1.37n \log n \in O(n \log n)$.*

We give two proofs of this; the first one is perhaps simpler to understand but the second one gives a tighter bound.

Proof. We show the claim only for n divisible by 4; the other cases are similar with appropriate rounding. We claim that

$$T(n) \leq 2n \cdot \log_{4/3} n \text{ for all } n \geq 1$$

and prove this by induction on n . The base case ($n = 1$) holds since $\log_{4/3} 1 = 0$. For the induction step, call an index i in the summation *small* if $i \leq \frac{3}{4}n$ and *large* otherwise. The contribution of a small index i to $T(n)$ is hence (by induction)

$$T(i) \leq 2i \cdot \log_{4/3} i \leq 2i \cdot \log_{4/3} (\frac{3}{4}n) = 2i \cdot \log_{4/3} n - 2i.$$

The contribution of a large index i to $T(n)$ is at most $2i \cdot \log_{4/3} n$. So the recursive formula becomes

$$T(n) \leq n + \frac{2}{n} \sum_{i \text{ small}} 2i \cdot \log_{4/3} n - \frac{2}{n} \sum_{i \text{ small}} 2i + \frac{2}{n} \sum_{i \text{ large}} 2i \cdot \log_{4/3} n \leq n + \frac{4}{n} \sum_{i=2}^{n-1} i \log_{4/3} n - \frac{4}{n} \sum_{i=2}^{\frac{3}{4}n} i$$

Observe that

$$\frac{4}{n} \sum_{i=2}^{n-1} i \leq \frac{4}{n} \cdot \frac{(n-1)n}{2} = 2(n-1) \quad \text{and} \quad \frac{4}{n} \sum_{i=2}^{\frac{3}{4}n} i = \frac{4}{n} \cdot \frac{1}{2} \cdot \frac{3}{4}n \left(\frac{3}{4}n + 1 \right) - \frac{4}{n} \geq n \quad \text{by } n \geq 4.$$

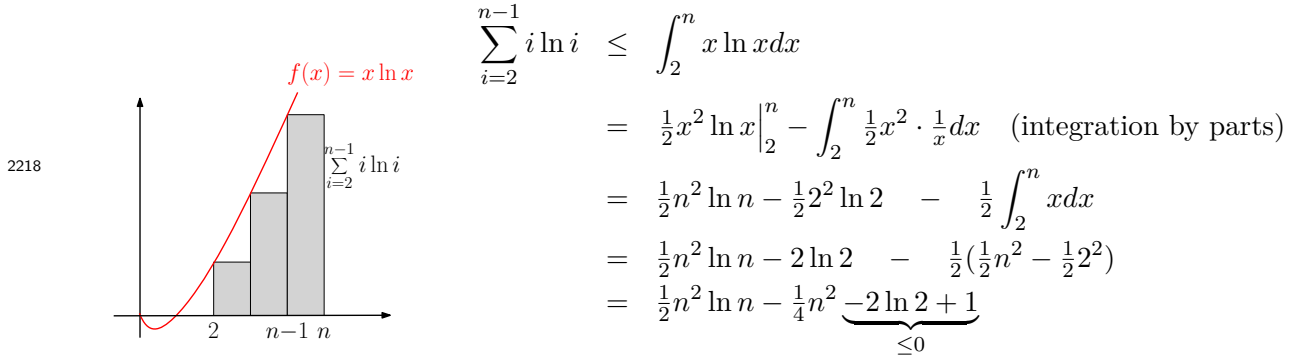
2209 Therefore $T(n) \leq n + 2(n-1) \log_{4/3} n - n \leq 2n \log_{4/3} n$ as desired. \square

2210 This proof shows that in the average case we use at most $2 \log_{4/3} n \log n \approx 4.8n \log n$ key-
 2211 comparisons (recall that all logs are base-2 unless said otherwise). Because *quick-sort* is *so*
 2212 important in practice, researchers have tried to find proofs that the constant is much smaller.
 2213 The following proof of Lemma 3.7 needs more advanced mathematical tools, but gives a much
 2214 tighter constant (in fact, one can argue that this is the best constant one can achieve).

Proof. Recall that $\ln n$ denotes the natural log (i.e., base e), and that its derivative is $\frac{1}{n}$. The proof is by induction and the base case ($n = 1$) clearly holds since $\ln 1 = 0$. Now consider $n > 1$. We have by induction

$$T(n) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} T(i) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} (2 \cdot i \ln i) = n + \frac{4}{n} \sum_{i=2}^{n-1} i \ln i.$$

2215 Observe that the function $f(x) = x \ln x$ is monotonically increasing. Therefore the sum can
 2216 be upper-bounded by the integral (because the sum describes the unit-step-function below the
 2217 function-curve). In other words



Therefore

$$T(n) \leq n + \frac{4}{n} \sum_{i=2}^{n-1} i \ln i \leq n + \frac{4}{n} \left(\frac{1}{2} n^2 \ln n - \frac{1}{4} n^2 \right) = 2n \ln n$$

2219 as desired. □

2220 **Summarizing *quick-sort*:** So in summary, we have now shown that *randomized-quick-sort* has
 2221 expected run-time $O(n \log n)$, which implies that *quick-sort* has average-case run-time $O(n \log n)$.
 2222 But remember that in the worst case, *quick-sort* is not fast (it could have run-time $\Theta(n^2)$).
 2223 Whether this worst case happens depends much on how we select the pivot, which we will study
 2224 in the next section.

2225 3.2.2 Tips & Tricks for *quick-sort*

2226 There are numerous ways to make *quick-sort* a bit faster in practice. (None of them affect
 2227 the asymptotic bounds.) We will list a few of them here. Contrast Algorithm 3.6 (the short
 2228 and intuitive version of *quick-sort* that we had earlier) with Algorithm 3.7; we will explain the
 2229 changes one-by-one below.

Algorithm 3.6: *quick-sort*($A, n \leftarrow A.size$) // simple version repeated

Input : array A of size n

```

1 if  $n \leq 1$  then return
2  $p \leftarrow \text{choose-pivot}(A)$ 
3  $i \leftarrow \text{partition}(A, p)$ 
4 quick-sort( $A[0, 1, \dots, i-1]$ )
5 quick-sort( $A[i+1, \dots, n-1]$ )

```

Algorithm 3.7: *quick-sort*($A, n \leftarrow A.size$) // improved version

```

1 Initialize a stack  $S$  of index-pairs with  $\{(0, n-1)\}$ 
2 while  $S$  is not empty do
3    $(\ell, r) \leftarrow S.pop()$ 
   // We want to sort  $A[\ell..r]$ 
4   while  $(r - \ell + 1 > 10)$  do
5      $p \leftarrow \text{choose-pivot}(A, \ell, r)$ 
6      $i \leftarrow \text{partition}(A, \ell, r, p)$ 
     // Need to sort subproblems  $A[\ell..i-1]$  and  $A[i+1..r]$ .
     // Stash one on stack, and update boundaries for the other.
7     if  $(i - \ell > r - i)$  then
8        $S.push((\ell, i-1))$ 
9        $\ell \leftarrow i+1$ 
10    else
11       $S.push((i+1, r))$ 
12       $r \leftarrow i-1$ 
13 insertion-sort( $A$ )

```

2230 **Do not pass sub-arrays**

2231 Algorithm 3.6 does the recursion using an array as parameter. In general this is a bad idea. As
 2232 discussed with *merge-sort* (Algorithm 1.4), passing an array as a parameter means that much
 2233 time is spent on allocating arrays and copying elements. This should be avoided when possible.

2234 In case of *quick-sort*, any subproblem that we need to solve can be uniquely described by
 2235 giving left and right boundaries ℓ, r of the range that we wish to sort. The sorting routine for
 2236 $A[\ell..r]$ never uses array-entries outside this range and so can safely re-use the same array. As
 2237 such, we describe subproblems via index-pairs (ℓ, r) only.

2238 **Stop recursion early**

2239 Experiments have shown that *quick-sort* spends a *lot* of time on the subproblems that have very
 2240 small size. It stops when the size is 0 or 1, but even on a problem of size 3 it will do 2 recursive
 2241 calls before putting the items in order. Since we only need 2 key-comparisons to sort 3 items,
 2242 this is a lot of overhead for very little sorting.

2243 Therefore an idea is to stop the recursions early, when the subproblem has fairly small size.
 2244 (Experiments have shown that cutting off at size 10 seems to offer the best tradeoffs.) This
 2245 creates an array A that is not necessarily sorted, but it is *almost-sorted*: Every item in the array
 2246 is within distance 10 of its correct place (where distance is measured by difference of indices). To
 2247 see this, consider one item x , and the last subproblem (ℓ, r) that contains x and into which we

recursed. We reached this subproblem because $A[\ell-1]$ and $A[r+1]$ were in their correct place. (Let us assume that these items actually exist; otherwise we are at the boundary of the array and the argument is similar.) Since $A[\ell-1] \leq x \leq A[r+1]$ by *partition*, therefore the correct place for x is somewhere within indices ℓ, \dots, r , i.e., at most distance 10 away.

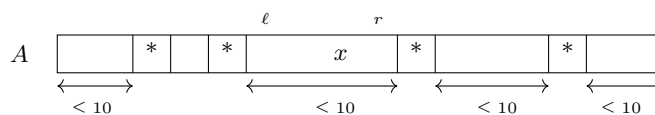


Figure 3.3: An almost-sorted array. Entries marked * are at their correct index.

To sort an almost-sorted array, the method of choice is *insertion-sort*. Recall that this has $\Theta(n^2)$ worst-case run-time, but $\Theta(n)$ best-case run-time if the array is sorted. In fact, the time spent can be expressed as the sum of the time spent on putting each item x into its correct place. If the array is almost-sorted, then we spend $O(1)$ time on each item x and hence $O(n)$ time total.

Avoid recursions

It is a general principle in software development that if there is a way to avoid recursion, then one should avoid recursion. (There is a lot of internal overhead for the recursive calls which tends to slow down programs.) Often one can re-write a recursion as a while-loop instead. (This is also called *tail-recursion*, and smart compilers automatically try to convert recursive code into tail-recursions, but doing it explicitly is safer.)

In *quick-sort*, we have two recursions, but we can re-write one as a while-loop. The other recursion cannot really be avoided: since we branch into two subproblems, but can work on only one at a time, one of the subproblems has to be “stashed” somewhere until we can return to it later. However, we can at least avoid storing quite so much overhead with the recursion. The subproblem that we need to store can be characterized with only two numbers: The left and right index of the sub-array that still needs to be sorted. So rather than calling recursion, we should store this minimal description (a pair of numbers) on a stack.

Reduce auxiliary space

What is the auxiliary space used by *quick-sort*? As for *quick-select*, we must include here the auxiliary space used by the recursions. Using a stack makes this space-usage explicit: We need $|S|$ auxiliary space, where $|S|$ denotes the number of items on stack S .

If we choose the subproblem to put onto the stack arbitrarily, then $|S|$ is $\Theta(n)$ in the worst case. Consider the example where the pivot-index is always the 20^{th} item from the right. So the right subproblem is quite small, but not so small that we are done with it. See Figure 3.4(left). If we always put the right subproblem on the stack, then the size of the stack is roughly $n/20 \in \Omega(n)$ when we reach the leftmost leaf.

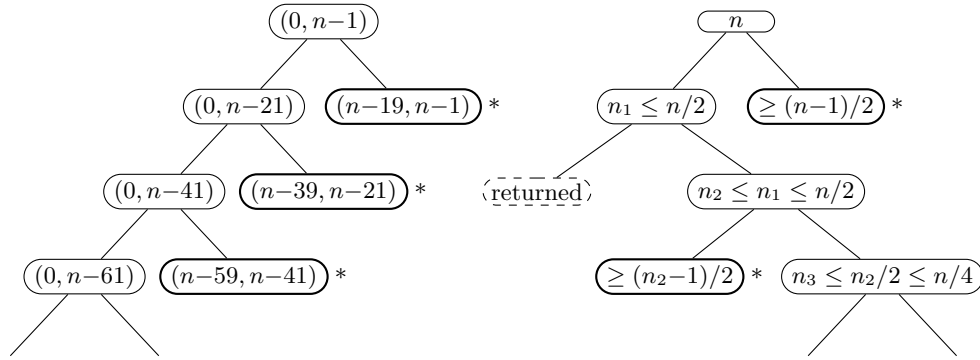


Figure 3.4: Which subproblem should be on the stack? (Starred thick nodes indicate subproblems on the stack.)

To avoid this, the code puts the *bigger* subproblem on the stack (breaking ties arbitrarily). With this, the auxiliary space is $O(\log n)$ due to the following.

Lemma 3.8. *At any time of the execution, $|S| \leq \log n$.*

Proof. (cs240e) Consider the subproblems P_0, P_1, \dots, P_k that are currently active, where P_0 is the outer-most recursion (of size n). Write n_i for the size of P_i . Now there are two possibilities for $i = 1, \dots, k$:

- P_i was the smaller subproblem of P_{i-1} . Then we added the sibling of P_i to the stack, but we also know that $n_i \leq (n_{i-1} - 1)/2 \leq n_{i-1}/2$.
- P_i was the larger subproblem of P_{i-1} . This means that we had already worked on the smaller subproblem of P_{i-1} and returned from it. We only know $n_i \leq n_{i-1} - 1$, but we also know that the sibling of P_i is not on the stack.

See also the right tree of Figure 3.4. The stack-size equals the number of problems on the stack, and all such problems must be siblings of one of the problems P_1, \dots, P_k . Since having a sibling on the stack implies having size at most half the size of the parent, we hence have

$$n_k \leq n_0 \cdot \left(\frac{1}{2}\right)^{|S|} \quad \text{or equivalently} \quad 2^{|S|} \leq \frac{n_0}{n_k}.$$

Since $n_0 = n$ and $n_k = 1$, therefore $2^{|S|} \leq n$ or $|S| \leq \log n$ as desired. \square

Choosing the pivot-index

So far, we have put no restriction on how to choose the pivot-index p . The simplest approach is to choose the rightmost item as pivot, i.e., to set $p = n-1$. In practice this is a *terrible* idea. Recall that we discussed that arrays for sorting are often close to being sorted. If we use the rightmost item as pivot, then this will give us the worst-possible split if the array is sorted, which means that *quick-sort* will take time $\Omega(n^2)$.

So if one has to do a simple approach that fixes the pivot-index directly, one should at least use $p = \lfloor n/2 \rfloor$, so that if the array is sorted, the best-case time run-is achieved. However, a small trick called *median-of-3* works even better. This works as follows. Consider the three items $A[0]$, $A[\lfloor n/2 \rfloor]$, $A[n-1]$, and use the median of these three as the pivot-value. We use three extra key-comparisons to determine the pivot, but in exchange have taken a small sample of the array and should likely get a pivot that is towards the middle. Of course one can drive this further (take bigger samples, e.g. median-of-7, and perhaps take the sample randomly), but in practice median-of-3 seems to offer the best tradeoff between extra time spent on finding the pivot and improvement in the run-time since the pivot is (usually) good.

Median-of-3 works well in practice, but one can construct examples where the worst-case run-time is still $\Theta(n^2)$. But we can also do a similar trick as for *quick-select* and choose the pivot randomly. The resulting algorithm (called *randomized quick-sort*) performs extremely well in practice. Also, just like for randomized *quick-select*, one argues that its expected run-time equals the average-case run-time of *quick-sort*, i.e., it is $\Theta(n \log n)$.

Here is one pivot-choice strategy that one *shouldn't* do. We mentioned briefly earlier that there exists an algorithm to do *select* in worst-case linear time. In particular therefore we can find the median of array A in linear time. We could use this median as the pivot and achieve asymptotically the best-case run-time for *quick-sort*. However, the run-time for finding the median has such a large constant that this algorithm is not at all competitive (you would be much faster with *heap-sort* or *merge-sort*). For cs240 we hence treat *quick-sort* as an algorithm where the worst-case run-time is $\Omega(n^2)$.

Other improvement ideas

Sorting is such an important topic that researchers continue to work and improve it. And even though *quick-sort* is decades old, improvements continue to be made. We only mention a few ideas here to give a glimpse of what one could do; you are not expected to know any details of these topics.

One observation is that Hoare's *partition* routine currently is non-specific about items that are equal to the pivot-value v . One could instead split the array three-ways, keeping all items equal to v in the middle and recursing on sub-arrays that are much smaller if there are many duplicate items.

A:

$< v$	$= v$	$> v$
-------	-------	-------

This *can* be done, even while being in-place, but the code gets quite complicated and will not be given here.

A very recent improvement (from 2009) is to use more than one pivot. For example, if we use three pivots (say $v \leq w \leq x$ are the pivot-values), then the array gets split 4-ways.

A:

$\leq v$	v	$\geq v, \leq w$	w	$\geq w, \leq x$	x	$\geq x$
----------	-----	------------------	-----	------------------	-----	----------

The idea is obvious (and probably much older); the real improvement occurs only if one chooses v, w, x carefully, especially when combining with randomization.

3.3 Lower Bound for Comparison-based Sorting

We have now seen many sorting algorithms, some of which have run-time $\Theta(n^2)$ while others have run-time $\Theta(n \log n)$ time. Can we do better? The answer to this question depends on what kind of algorithms we allow. We will argue that for a general-purpose sorting algorithm (where we know nothing about the keys other than that they can be compared) one cannot do better. Later sections will then study specific types of keys and argue that we *can* do better if the keys have a suitable structure.

So in this section we study only *comparison-based algorithms*, recall that this means algorithms that use no information about the input except the outcome of key-comparisons. All algorithms we have seen so far are comparison-based.

Theorem 3.2. *Any comparison-based sorting algorithm \mathcal{A} uses $\Omega(n \log n)$ key-comparisons in the worst case.*

Before proving this theorem, we should think about how one could even prove such a thing. If someone gives you an algorithm, then it is (usually) not too hard to find an example where it does badly. But how can we prove a lower bound on an algorithm that we do not know?

The main tool that we need here is the *decision tree*, see Figure 3.5. This is a tree that describes succinctly the decisions that were taken during an algorithm and the resulting outcome. Such a structure can be created for any comparison-based algorithm, and is a binary tree with the following properties:

- Any interior node corresponds to a key-comparison, i.e., the question whether x_i is smaller than x_j or not, for some i, j . (We assume that the instance consists of keys x_0, \dots, x_{n-1} .)
- At the root, we write down the first key-comparison that would be done when executing the algorithm on n keys.
- The key-comparison at the root has two possible outcomes. Depending on how the comparison was set up, these outcomes are either $<$ and \geq , or \leq and $>$.¹
- We give the root two children, and label the links to these children with the outcomes.
- For each child, we build the decision tree in a similar manner, by asking what the next key-comparison would be, given the outcome of the key-comparison at the root.
- If at some point the algorithm makes no more key-comparisons, then we create a leaf of the decision tree and label it with the result returned by the algorithm. (In case of a sorting algorithm, this result would be the sorting permutation.)

Figure 3.5 illustrates a decision tree for sorting three keys. Following the steps in the definition, any comparison-based sorting-algorithm can be converted into such a decision tree. Vice

¹Some references use *three* outcomes $<$, $=$ or $>$. But deciding among them actually requires *two* key-comparisons in a computer, so we will stick with two outcomes.

2368 versa, we can read a sorting-algorithm from a decision tree. To see this, consider the input
 2369 $\langle 4, 2, 7 \rangle$, i.e., $x_0 = 4$, $x_1 = 2$, $x_2 = 7$. At the root, the algorithm compares $x_0 : x_1$, i.e., 4 with
 2370 2, and since $4 \geq 2$, it goes right. There it compares $x_1 : x_2$, i.e., 2 with 7, and since $2 < 7$ it
 2371 goes left. Then it compares $x_0 : x_2$, i.e., 4 with 7, and since $4 < 7$ it goes left. This brings
 2372 us to a leaf, and the algorithm returns $\langle 1, 0, 2 \rangle$ as the sorting permutation. In other words, the
 2373 algorithm determines $x_1 \leq x_0 \leq x_2$ as sorted order, which is correct since $2 \leq 4 \leq 7$.

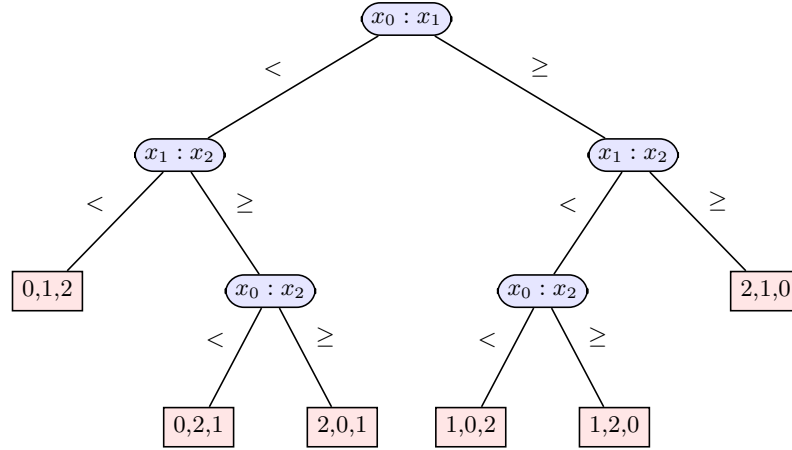


Figure 3.5: The decision tree for an algorithm to sort 3 elements x_0, x_1, x_2 with at most 3 key-comparisons.

2374 Now we are ready for the proof of Theorem 3.2.

2375 *Proof.* Fix an arbitrary comparison-based sorting-algorithm \mathcal{A} , and consider how it sorts an
 2376 instance of size n . Because \mathcal{A} uses only key-comparisons, we can express it as a decision tree \mathcal{T} .

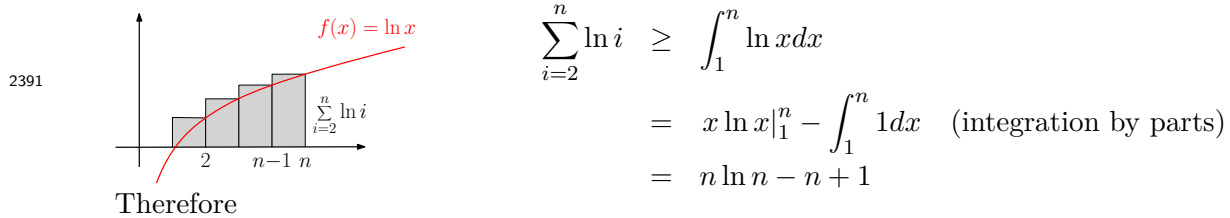
2377 For each sorting permutation π , let I_π be an instance that has distinct items and sorting
 2378 permutation π (e.g. $I_\pi = \pi^{-1}$). Executing \mathcal{A} on I_π must lead to a leaf that stores π . No two
 2379 sorting permutations can lead to the same leaf, otherwise the output would be incorrect for
 2380 one of them since items are distinct. We have $n!$ sorting permutations, hence at least $n!$ leaves
 2381 that are reached for some I_π . (There could be even more leaves in \mathcal{T} if the algorithm has some
 2382 branches that are never reached, but we will not consider these.)

2383 Let h be the largest layer-number of a leaf reached by some I_π . Since \mathcal{T} is binary, for any
 2384 $0 \leq \ell \leq h$ there are at most 2^ℓ leaves in layers $0, \dots, \ell$. So necessarily $2^h \geq n!$ or

$$\begin{aligned}
h \geq \log(n!) &= \log(n \cdot (n-1) \cdot \dots \cdot 1) \\
&= \log n + \log(n-1) + \dots + \log 1 \\
&\geq \log n + \log(n-1) + \dots + \log(\lceil n/2 \rceil) \quad (\text{delete a few terms}) \\
&\geq \log(n/2) + \log(n/2) + \dots + \log(n/2) \quad (\geq \lceil n/2 \rceil \text{ terms remain}) \\
&\geq n/2 \log(n/2) = (n/2) \log n - (n/2) \in \Omega(n \log n).
\end{aligned}$$

2385 Consider the sorting permutation π that has its leaf on layer h . Executing algorithm \mathcal{A} on
 2386 I_π will hence take $h \in \Omega(n \log n)$ key-comparisons, so the worst-case bound holds. \square

2387 **More and better lower bounds: (cs240e)** We can make this lower bound slightly stronger.
 2388 First, the lower bound on $\log(n!)$ was rather loose. We can make this tighter by using a trick
 2389 similar to the one used for Lemma 3.7: Lower-bound a sum by an integral. Specifically, we know
 2390 that



$$\log(n!) = \frac{1}{\ln 2} \ln(n!) = \frac{1}{\ln 2} \sum_{i=2}^n \ln i \geq \frac{1}{\ln 2} (n \ln n - n) = n \log n - \Theta(n)$$

2392 which means that in the worst case, any sorting algorithm uses $n \log n$ key-comparisons (minus
 2393 lower-order terms).

2394 As a second improvement, we can give a lower bound even on the average-case run-time.

2395 **Theorem 3.3.** *Any comparison-based sorting algorithm \mathcal{A} uses $\Omega(n \log n)$ comparisons in the*
 2396 *average case.*

Proof. Fix again the decision tree \mathcal{T} corresponding to \mathcal{A} . For each sorting permutation π , define $I_\pi = \pi^{-1}$ and let ℓ_π be the level-number of the leaf that we reach when executing \mathcal{A} on input I_π . Write Π_n^{high} for those sorting permutations where $\ell_\pi \leq \lfloor \log(n!) \rfloor - 1$ and Π_n^{low} for the remaining ones. Within layers $0, 1, \dots, \lfloor \log(n!) \rfloor - 1$ we have no more than

$$2^{\lfloor \log(n!) \rfloor - 1} \leq \frac{1}{2} 2^{\log(n!)} = \frac{n!}{2}$$

leaves, so at least half of the sorting permutations are in Π_n^{low} . If we use $T_{\mathcal{A}}^{\text{avg}}(\cdot)$ to denote the average-case number of key-comparisons taken by \mathcal{A} , we hence have

$$\begin{aligned} T_{\mathcal{A}}^{\text{avg}}(n) &= \frac{1}{n!} \sum_{\pi \in \Pi_n} \ell_{\pi} = \frac{1}{n!} \left(\sum_{\pi \in \Pi_n^{\text{high}}} \underbrace{\ell_{\pi}}_{\geq 0} + \sum_{\pi \in \Pi_n^{\text{low}}} \underbrace{\ell_{\pi}}_{\geq \lfloor \log(n!) \rfloor} \right) \\ &\geq \frac{1}{n!} |\Pi_n^{\text{low}}| \lfloor \log(n!) \rfloor \geq \frac{|\Pi_n^{\text{low}}|}{n!} (\log(n!) - 1) \geq \frac{1}{2} \log(n!) - \frac{1}{2} \in \Omega(n \log n) \end{aligned}$$

as desired. □

3.4 Sorting integers

Now we turn to the situation where the keys are known to have a special structure, for example they could be integers within a small range. So for this section, we assume that we are given n numbers, and we know that they are all integers that have exactly m digits, where each digit falls into the range $\{0, \dots, R-1\}$. The value R is called the *radix* and typically we have $R = 10$ (for humans) or $R = 2$ or $R = 16$ (for computers). But principally R could be anything (even non-constant). In our example, we will use $R = 4$ since this is big enough to be interesting and small enough to keep the examples manageable.

How would a human sort such numbers? For example, if you were handed a pile of student exams that you need to sort by the student ID number (an integer with 8 digits in the range $\{0, \dots, 9\}$), how would you do this? Most likely you would look first at the first significant digit, and break the pile of exams into 10 piles according to the first digit. Then you would work on each pile and sort it by the second digit, and so on. This is indeed one of the methods that we will see below (it is called *MSD-radix-sort*), but as it turns out, there is another method that usually is even faster. We need a bit of background before we can explain them both.

3.4.1 Sorting by one digit

Let us study first how we can sort radix- R numbers by one of their digits. There is a very obvious approach: We have R possibilities for what the digit can be, so create R lists B_0, \dots, B_{R-1} and put each number in list B_i if it has digit i . Then put the items back in order by taking first the items in B_0 , then the ones of B_1 , etc. This is *bucket-sort*; the code is in Algorithm 3.8 and an example is in Figure 3.6.

The pseudo-code given here for *bucket-sort* is specifically for m -digit radix- R numbers, but the idea of bucket-sort works more generally for any set of numbers within a bounded range: If we have an array $A[0..n-1]$ where all numbers are in $\{0, \dots, R-1\}$, then the same code can be used to sort them after replacing “ d^{th} digit of $A[i]$ ” by “ $A[i]$ ”.

The run-time of bucket-sort is $O(n + R)$, and the auxiliary space is $\Theta(n + R)$, because we use $\Theta(R)$ space for array B and $\Theta(1)$ auxiliary space per item when creating the lists.

Algorithm 3.8: *bucket-sort*($A, n \leftarrow A.size, \ell \leftarrow 0, r \leftarrow n-1, d$)

Input : Array A of size at least n contains numbers with m digits in $\{0, \dots, R-1\}$.
 $1 \leq d \leq m$

Output: $A[\ell..r]$ is sorted by d^{th} digit.

```

1 Initialize an array  $B[0..R-1]$  of empty lists
2 for  $i \leftarrow \ell$  to  $r$  do                                // Move array-items to buckets
3    $\lfloor$  append  $A[i]$  at the end of  $B[d^{\text{th}}$  digit of  $A[i]$ 
4  $i \leftarrow \ell$ 
5 for  $j \leftarrow 0$  to  $R-1$  do                                // Move bucket-items to array
6   while  $B[j]$  is non-empty do
7      $\lfloor$  move first element of  $B[j]$  to  $A[i++]$ 

```

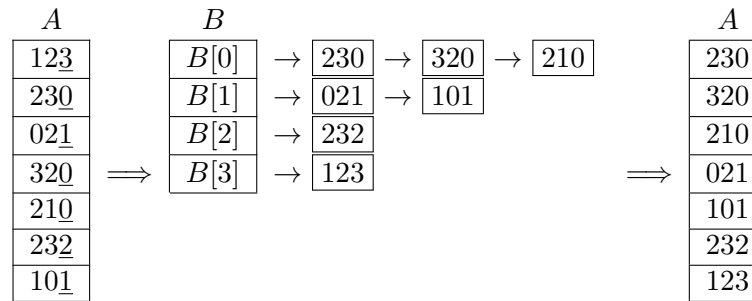


Figure 3.6: An example of bucket-sort to sort base-4 numbers by their last digit.

2427 We claim that *bucket-sort* is stable. Recall that this means that if you have two items that
 2428 are the same (with respect to the key by which we sort), then in the output the items stay in the
 2429 same relative order. Consider for example the numbers 230 and 320 in Figure 3.6. We handle
 2430 230 first, and so place it in $B[0]$ first. When we come to 320, we append it at the end of $B[0]$,
 2431 so it comes later. Since we extract items from $B[0]$ in order to put them back into the array,
 2432 therefore 230 comes before 320 in the final output.

2433 It is worth mentioning that one can avoid the lists in *bucket-sort* and use instead only two
 2434 arrays: One array of size R to *count* how many items would be in each bucket, and one array
 2435 of size n into which we can then (based on the count) directly copy array A in sorted order.
 2436 This method is called *count-sort*. We will leave the details as an exercise, but only note that
 2437 while *count-sort* would be somewhat faster in practice, its theoretical bounds on run-time and
 2438 auxiliary space are the same as for *bucket-sort*.

3.4.2 Sorting multi-digit numbers

Now we sort multi-digit numbers. Recall that we made the assumptions that they all have the same number of digits, so if they do not then we need to pad them with leading 0s. The first algorithm to sort such numbers follows the idea introduced earlier: Split the numbers into groups based on the leading digit, then each group based on the next digit, etc. The resulting recursive algorithm is called *MSD-radix-sort*; its pseudocode is in Algorithm 3.9 and an example is in Figure 3.7.

Algorithm 3.9: *MSD-radix-sort*($A, n \leftarrow A.size, \ell \leftarrow 0, r \leftarrow n-1, d \leftarrow 1$)

Input : Array A of size at least n , contains numbers with m digits in $\{0, \dots, R-1\}$

‘where m and R are global variables.

Optional integers ℓ, r indicate the range $A[\ell..r]$ that should get sorted.

Optional index d indicates the digit by which we wish to sort

```

1 if  $\ell < r$  then
2   bucket-sort( $A, n, \ell, r, d$ )
3   if  $d < m$  then
4     // Find sub-arrays that have same  $d$ th digit and recurse
5     while  $\ell' < r$  do
6        $\ell' \leftarrow \ell$ 
7       while  $r' < r$  and  $d$ th digit of  $A[r' + 1]$  equals  $d$ th digit of  $A[\ell']$  do  $r'++$ 
8       MSD-radix-sort( $A, n, \ell', r', d+1$ )
9        $\ell' \leftarrow r' + 1$ 

```

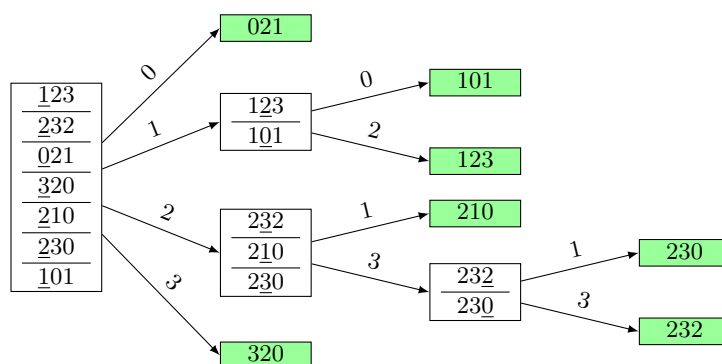


Figure 3.7: An example of MSD radix sort for base-4 numbers.

What is the run-time? Each call to *bucket-sort* takes $O(n_i + R)$ time, where $n_i = r - \ell + 1$

is the size of the sub-array that we need to sort. We also need $\Theta(n_i)$ time to parse the sub-array and recurse. Within each level of the recursions, the subproblem-sizes add up to $\sum_i n_i \leq n$. There may be $\Omega(n)$ subproblems on the level, so the worst-case run-time on each level is $O(n + R \cdot \{\text{subproblems on level}\}) \subseteq O(n + Rn) = O(Rn)$. Since the recursion depth is at most m , the total run-time is $O(mRn)$ (and one can show that this is tight). The auxiliary space is $\Theta(n + R + m)$ since *bucket-sort* uses $\Theta(n + R)$ auxiliary space, and the recursion-stack may use $\Theta(m)$ space.

Algorithm *MSD-radix-sort* will be fairly slow in practice, because it uses lots of recursions. There are numerous little tricks that can be applied to improve *MSD-radix-sort*, many are similar to what we did for *quick-sort*. For example, we could replace the recursions with a stack (but the stack would get quite big), we could do smallest recursions first (so that the stack does not get too full), we could stop the recursions once the subproblem is quite small (and use *insertion-sort* once at the end instead). But none of these get around the fundamental problem that many recursions are needed explicitly or implicitly. Therefore *least significant digit radix sort*, which achieves the same run-time and has no recursion, is frequently preferred. See Algorithm 3.10 for the pseudocode and Figure 3.8 for an example of *LSD-radix-sort*.²

Algorithm 3.10: *LSD-radix-sort*($A, n \leftarrow A.size$)

Input : Array A array of size n , contains m -digit radix- R numbers, where m and R are global variables.

1 **for** $d \leftarrow m$ **down to** 1 **do** *bucket-sort*(A, d)

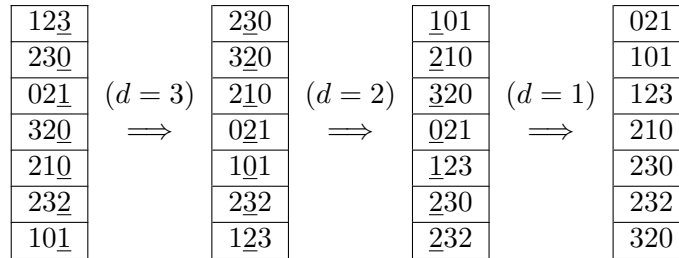


Figure 3.8: An example of *LSD-radix-sort*.

This is one place where correctness is really not all that obvious, until we state the appropriate loop invariant:

The input array would be sorted if only digits $d + 1, \dots, m$ of each number are considered.

²Any allusions to hallucinogenic drugs are pure acronym coincidences.

In the example of Figure 3.8, after round $d=2$ the array restricted to only digits 2, 3 would be $\langle 01, 10, 20, 21, 23, 30, 32 \rangle$, which is in sorted order. The invariant certainly holds initially, when $d = m$ and so we are not looking at any digits and the array is vacuously sorted. When executing the loop for value d , we sort by the d th digit. Here it is crucial that the sorting method that we use is *stable*: equal entries are kept in the same relative order. So the output of *bucket-sort* is sorted by the d th digit, and within any sub-array where the d th digit is the same everywhere, ties are broken by how the numbers were sorted before. Since they were sorted with respect to digits $d+1, \dots, m$, each sub-array is hence in sorted order by digits d, \dots, m , and so is all of A .

LSD-radix sort obviously has run-time $\Theta(m(n+R))$, and uses auxiliary space $\Theta(n+R)$ inside the call to *bucket-sort*. So its run-time bound is better than for *MSD-radix-sort*, and it has the far simpler code and no recursions. However, it has one drawback: it looks at *all* digits of *all* input numbers, whereas *MSD-radix-sort* only looks at those digits that are actually required to put the numbers into the sorting order. So if there are many different digits, and few numbers that have the same leading digits, then *MSD-radix-sort* might be faster.

Note that the radix R is usually a constant. If m (the number of digits) is also a constant, as would be expected in a 32-bit or 64-bit computers where numbers fit into one memory cell, then the run-time of these radix-sorts becomes $O(n)$. At first sight this seems a contradiction to the lower bound of $\Omega(n \log n)$ for sorting. But this is not a contradiction, because the lower bound only holds for *comparison-based* sorting, and here we used an algorithm that is not comparison-based. To see exactly where the proof of the lower bound fails: If we only consider 64-bit numbers, then there are at most 2^{64} distinct numbers, which means that there are at most $(2^{64})^n$ sorting permutations that could be needed. This is a large number, but smaller than $n!$ for sufficiently large n , which is why the lower bound cannot be carried over to sorting finite-length numbers.

3.5 Take-home messages

We have now seen numerous algorithms for sorting—it is an important and *very* well-studied problem. Table 3.9 summarizes the sorting methods that we have seen.

Summarizing, we can sort n items in $\Theta(n \log n)$, but we cannot sort them in $o(n \log n)$ time unless we know more about the input. Of the algorithms listed above, *quick-sort* is fastest in practice if implemented carefully, but *not* the fastest in the worst-case. Algorithm *heap-sort* performs quite well in practice, uses only $O(1)$ auxiliary space, and is much easier to implement to be fast.

Another interesting concept that we applied in this chapter are randomized algorithms. These can eliminate bad cases by shifting “bad” from “bad input” to “bad luck” so that the expected run-time of the randomized algorithm equals the average-case run-time of the non-randomized algorithm.

Sort	Running time	Analysis	Comments
<i>insertion-sort</i>	$\Theta(n^2)$	worst case	good if mostly sorted, stable
<i>merge-sort</i>	$\Theta(n \log n)$	worst case	flexible, merge-routine useful, stable
<i>heap-sort</i>	$\Theta(n \log n)$	worst case	clean short code, in-place
<i>quick-sort</i>	$\Theta(n^2)$ $\Theta(n \log n)$	worst case average case	in-place, fastest in practice
<i>randomized quick-sort</i>	$\Theta(n^2)$ $\Theta(n \log n)$	worst case expected	
comparison-based sort	$\Omega(n \log n)$	worst case	
<i>bucket-sort</i>	$\Theta(n + R)$	worst case	stable, needs integers from $[0, R)$
<i>radix-sort</i>	$\Theta(m(n + R))$	worst case	stable, needs m -digit radix- R numbers

Table 3.9: Sorting summary

3.6 Historical notes

As the name of the *partition* routine suggests, *quick-select* and *quick-sort* were both developed by Tony Hoare [Hoa61, Hoa62]. *quick-sort* quickly became widely adopted and implemented in standard libraries, for example the *C* standard library contains `qsort`. Numerous variations and properties have been studied subsequently, see for example the books by Bentley [Ben86] and Sedgewick [Sed98] for extensive discussions. The newest developments using multiple pivots were proposed Yaroslavsky and further analyzed by Wild et al. [WNM16].

The lower bound for comparison-based sorting follows easily from Shannon's entropy lower bound [Sha48]; one of the earliest papers to point it out explicitly for sorting appears to be by Bell [Bel58]. Knuth's book [Knu98] gives a good overview of the history of sorting up to the 1990s. Bucket-sort and radix-sort are extremely old, and have been used to implement mechanical sorting machines in the 19th century, beginning with Hollerith's tabulation machine.

Chapter 4

Dictionaries and balanced binary search trees

Contents

4.1	ADT Dictionary	115
4.1.1	Implementations you have seen	116
4.1.2	Outlook	119
4.1.3	Lazy deletion (cs240e)	119
4.2	AVL trees	120
4.2.1	AVL-tree definition	120
4.2.2	Height of an AVL-tree	122
4.2.3	AVL tree operations	123
4.3	Other balanced binary search trees	131
4.3.1	Scapegoat trees (cs240e)	133
4.4	Take-home messages	138
4.5	Historical remarks	139

4.1 ADT Dictionary

In this chapter (as well as quite a few of the followings ones) we discuss the abstract data type Dictionary. You should have seen this in predecessor courses already (though perhaps not under this name; it is also known as symbol-table, relation, or map). ADT Dictionary stores *key-value pairs* and supports the following three operations:

- *insert*(k, v): insert the key-value pair (k, v) .
- *search*(k): return the key-value pair that matches the given key k , or report that there is no such key.
- *delete*(k): delete the key-value pair that matches the given key k .

In real-life applications, ADT Dictionary is part of so-called *no-SQL-storage systems*. (You may

be familiar with SQL, the query-language for data bases, which can be used only if the data is extremely structured. ‘no-SQL’ stands for ‘not only SQL’ and is used for storing arbitrary kinds of data.)

A few assumptions are usually made around ADT Dictionary.

- All keys are distinct.

This is a fair assumption—usually keys are something that uniquely identifies the entry, such as an ID number.

- We will not call *insert* or *delete* when not appropriate: *insert*(k, v) will not be called if key k already exists in the dictionary. *delete*(k) will only be called if key k already exists in the dictionary.

Of course a real-life implementation should buffer these functions so that they can handle erroneous calls with a suitable error-message. However, doing this should be straightforward, and so to keep the pseudo-code simple we omit the error-handling.

- Keys are something that can be compared and that fits in $O(1)$ space.

You might think of keys as integers, but most of our realizations work for any comparable items. We will later (in Section 6.2 when we study dictionaries for words) discuss keys that do not fit into $O(1)$ space.

- Comparing two keys takes $O(1)$ time.

This assumption is justified when keys fit into $O(1)$ space. It does not hold for dictionaries for words (see Section 6.2).

- Values fit into $O(1)$ space, but we make no other assumptions, in particular values need not be comparable and may appear multiple times.

We will generally ignore the values and, as for heaps, not even show them in the pictures—it is silently assumed that when we show a key in a picture, this is really a reference to some item that stores both the key and the value. One exception will be made for dictionaries in external memory (Section 11.3), where the choice of where to store values can make a difference.

Also, unless explicitly said otherwise, n denotes the current size of the dictionary, i.e., the number of key-value pairs that are stored.

4.1.1 Implementations you have seen

We first review three implementations of dictionaries that you should be familiar with from previous courses.

Unsorted list or array: Store the key-value pairs in a list or array in no particular order.

We can then insert in $\Theta(1)$ time (assuming, as usual, that arrays are dynamic arrays and expand as needed to have space without this counting against our run-time). To search for a key, we must spend $\Theta(n)$ time because we have no guidance where a particular key might be stored and so must look at all items. Once we found the key, we can delete it in constant time

(in an array this involves exchanging it with the last item). So the deletion-time is dominated by the time for searching, and is also $\Theta(n)$.

Sorted array: Store the key-value pairs in an array in increasing order.

This has the advantage that we can use *binary search* to find a key more efficiently in $\Theta(\log n)$ time. You should have seen binary search in previous courses; its main idea is to look at the item in the middle and the recursively search either to the left or the right. (We will briefly revisit binary search in Section 6.1, and then see some improvements that apply if ADT Dictionary stores integers in a sorted array.)

The price to pay is that insertion is slower, because the new key must be placed correctly in the sorted order, and so other items must “move over” to make a space free in an array. So insertion takes $\Theta(n)$ time, and by the same argument so does deletion.



Figure 4.1: Deleting 60 in a sorted array means moving over the items on the right.

One could theoretically also consider a sorted list for a realization of ADT Dictionary. But here binary search takes linear time, because we cannot get to the middle element directly but must traverse the list to get to it. Therefore sorted lists are not usually used for ADT Dictionary. (But see Chapter 5 for a realization of ADT Dictionary that is based on sorted lists.)

Binary search trees: Let us briefly recall the definition of a *binary search tree* (BST): It is a binary tree where every node stores a key-value pair and traversing the nodes in in-order yields them in increasing order. Put different, it needs to satisfy the *BST-order-property*: If the root of T stores key k , then all nodes in subtree $T.left$ store keys that are smaller than k , and all nodes in subtree $T.right$ store keys that are bigger than k . Figure 4.2 shows a binary search tree. Note that (as before) we only show the keys, not the associated values. In this figure we show the empty subtrees to remind us that they exist, but they will usually be omitted in the figures.

Searching is very easy in a binary search tree, since the BST-tree property tells us where to look. So to search for key k in binary search tree T , compare k with the root-key. If we do not find k there, then recurse in either $T.left$ or $T.right$, depending on whether k is smaller or bigger. Repeat until we either found k or reached an empty subtree.

Insertion is similarly easy. Perform the same steps as for searching, except that we know that we will end in an empty subtree. (Recall that *insert(k)* comes with the precondition that k is not yet in the tree.) Replace this empty subtree with a 1-node subtree that stores the key-value-pair that we want to insert.

Deletion is a bit more complicated. First search for the key k that we are looking for, and assume that we find it at node x . (The precondition ensures that k is actually in the tree.) Then we have three cases:

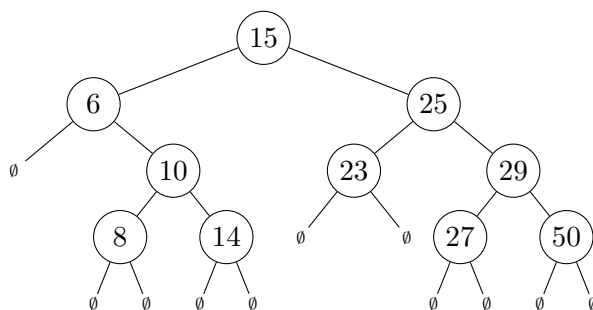


Figure 4.2: A binary search tree.

1. If x is a leaf, then we can simply delete it.
2. If x has exactly one non-empty subtree, then delete x and move the unique child c of x into its place.
3. If x has two non-empty subtrees, then we swap the key-value pairs of x and either its predecessor y or its successor z . These are the nodes such that $y.key < x.key < z.key$, and there are no other keys stored in the tree in-between. The choice between predecessor and successor is arbitrary; both work, and to keep the tree more balanced it is recommend to alternate between the two when deleting.

Let us assume we want to use the successor z . We can find z by going into $T.right$ and from there always to the left subtree until we reach an empty subtree; z is the last node before this. In particular, z has at least one empty subtree. Swap the key-value-pairs stored at x and z and then delete node z ; this can be done with one of the above two cases. Figure 4.3 shows an example of this operation.

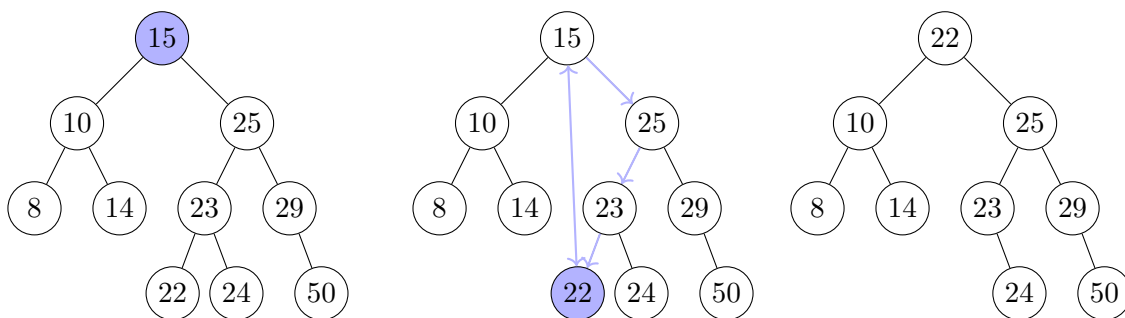


Figure 4.3: Deleting key 15 by exchanging it with its successor 22.

For all three operations, the worst-case run-time is $\Theta(h)$, where h is the height of the binary search tree. We know that $h \in \Omega(\log n)$, but unfortunately there are binary search trees of height $\Omega(n)$, so in terms of n the worst-case run-time is $\Theta(n)$.

4.1.2 Outlook

One can argue that $\Omega(\log n)$ is a lower-bound for searching in the key-comparison model, we will see this in Section 6.1.1. Binary search trees achieve this in the best case, so we are now aiming to find data structures that achieve this in more scenarios. There are numerous methods for doing so. We will see here two: AVL-trees achieve $O(\log n)$ worst-case time (but are slow in practice), and Skip-lists achieve $O(\log n)$ expected time (and are faster than AVL-trees in practice if implemented correctly). The enriched section will cover quite a few other realizations.

4.1.3 Lazy deletion (cs240e)

Of the three operations supported by ADT Dictionary, operation *delete* is perhaps the least important in the age of nearly infinite and nearly free memory. There exists a strategy called *lazy deletion*, which proceeds as follows:

- When *delete*(k) is called, first perform *find*(k) to find the key-value pair.
 - Mark the found key-value pair as “deleted”. We assume here that each key-value pair has a flag, which was set to “present” when the item was insert, and is switched to “deleted” now.
 - We can delete the value (to save space) but we *keep* the key because it may be needed for future searches.
 - When doing *search* and *insert*, we search among *all* keys in the data structure, regardless of whether they have been marked as deleted or not. (But of course *search* only returns an item if it is not marked ‘deleted’.)
 - Sometimes, *insert* may be able to re-use a place where we have a “deleted” key-value pair, but this depends on the requirements on the locations of keys. If in doubt, it is safer not to re-use such spots.
 - We keep track of how many items are marked “deleted”. If this counter reaches n (the number of items that are “present”), then we *rebuild*. This means re-creating the *entire* data structure from scratch.
- We can do a *rebuild* simply by initializing a new data structure and calling *insert* n times, but sometimes it can be done faster overall since we have all items available immediately.

Let us see one example: sorted arrays. Here we can *search* for a key in $O(\log N)$ time, where N is the items that are stored in the array, including the items marked “deleted”. But since we ensure that $N \leq 2n$ at all times, this is still $O(\log n)$. So if we do not need to rebuild then *delete* now takes $O(\log n)$ time (in contrast to the $\Theta(n)$ time with regular deletion).

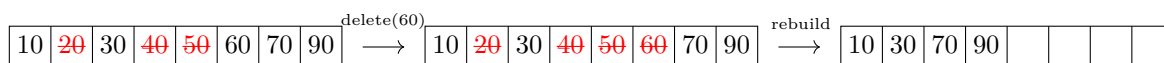


Figure 4.4: Lazy deletion in a sorted array usually means simply “striking out” the item, but occasionally we need to re-build entirely.

Roughly as for dynamic arrays, one can argue that on average over all operations, the run-time for lazy deletion is $O(\text{search} + \text{insert})$. This holds because re-building the structure can be done in $O(n * \text{insert})$ time, and was preceded by n cheap deletions that took $O(\text{search})$ time each. (Recall that this kind of analysis was called *amortized analysis*; details about it can be found in Section 1.5.3. One could formalize the above argument by using the number of deleted items as the potential function.)

In the case of sorted arrays, lazy deletion is even faster than $O(\text{search} + \text{insert})$, because when rebuilding we can extract the items in sorted order in $O(n)$ time, insert them in the new array in this order, and this takes only $O(n)$ time total. Therefore the amortized time for lazy deletion in sorted arrays is actually $O(\log n)$.

Summarizing, if one can live with a slight waste of space, and the occasional call to *delete* that is extremely slow (because the entire data structure gets re-built), one can save oneself the hassle of implementing the structural changes required for *delete*.

4.2 AVL trees

A *balanced binary search tree* is a class of binary search trees with some restriction on the structure that guarantees that the height is $\Theta(\log n)$. Many types of balanced binary search trees have been developed. We will see here one of the oldest and simplest, the AVL-tree. There are many others; we will discuss a few of them later.

For all balanced binary search trees, we need to discuss two major questions. First we need to describe the structural property, and argue that this implies that the height is $\Theta(\log n)$. This makes search fast. But then we must argue how we do insertion and deletion while maintaining the structural property. Usually the standard BST insertion and deletion is applied; the problem is then how to restore the structural property (if it becomes violated in the process) in $O(\log n)$ time.

4.2.1 AVL-tree definition

AVL-trees have the following structural property called *AVL-tree property*:

At any node of the tree, the heights of the left and right subtree differ by at most 1.

(It is worth recalling here that the height of a single-node tree is 0 and the height of an empty subtree is -1 .) The AVL-tree property means that for a tree of height h , there are only very few possibilities for the heights of the subtrees: At least one subtree must have height $h - 1$, and the other must have height $h - 1$ or $h - 2$.

For future reference, we rephrase the AVL-tree property in a different way. Define the *balance* of a node v to be the height-difference between right and left subtree, so

$$\text{balance}(v) = \text{height}(v.\text{right}) - \text{height}(v.\text{left}).$$

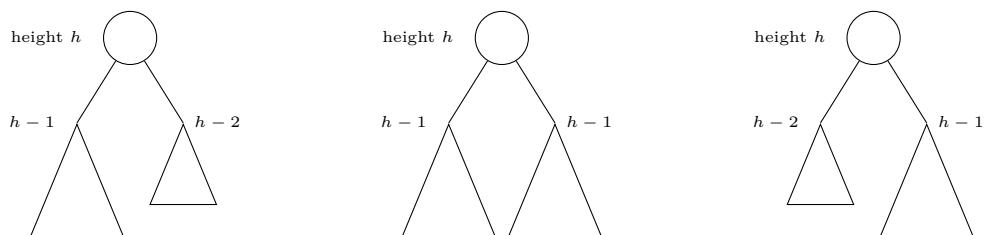


Figure 4.5: The shape of AVL-trees.

2696 Then a tree T is an AVL-tree if and only if every node v is *balanced*, i.e., $\text{balance}(v) \in \{-1, 0, +1\}$.
 2697 Figure 4.6 shows an example of an AVL-tree.

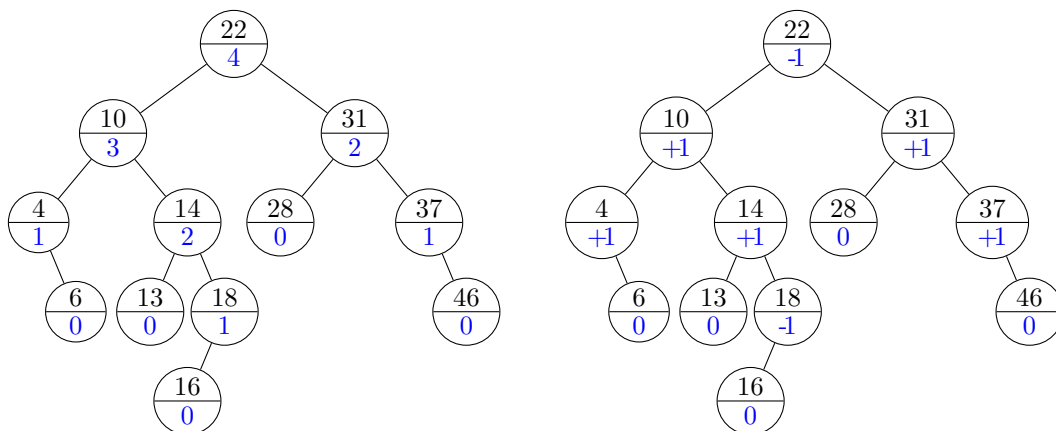


Figure 4.6: An AVL-tree, stored with heights or with balances.

2698 As outlined earlier, insertion and deletion in an AVL-tree will be done by doing the corre-
 2699 sponding BST-operation and then restoring balances. To be able to do this, we need to store
 2700 information that allows us to determine what the balances are. The simplest way to do this is to
 2701 store with each subtree the height of the subtree. There are also ways to do it if we only stored
 2702 the balance of each node; using the latter would save some space (we only need 2 bits per node,
 2703 as opposed to an integer) but significantly complicates the code. Therefore we will assume that
 2704 each subtree knows its height, and update these height-entries for relevant subtrees whenever
 2705 we insert or delete. Note that a node v has height $1 + \max\{v.\text{left.height}, v.\text{right.height}\}$, and so
 2706 we can re-compute its height in constant time, presuming the heights of its children have been
 2707 updated already.

4.2.2 Height of an AVL-tree

We now show that AVL-trees are indeed balanced binary search trees, i.e., have height $\Theta(\log n)$. We only need to show that the height is $O(\log n)$, because *any* binary tree has height $\Omega(\log n)$.

Lemma 4.1. *Any AVL-tree with n nodes has height $O(\log n)$.*

We prove this in a bit of an unusual way, by turning the perspective around. Normally one would prove such a statement by fixing an AVL-tree with n nodes, and wondering what the maximum possible height h could be. Instead, we will fix here an AVL-tree with height h , and ask what the minimum possible number of nodes could be. So define

$$N(h) \leftarrow \text{smallest number of nodes in an AVL-tree of height } h.$$

We will show:

Claim 4.1. $N(h) \geq c^h$ for some constant c .

This immediately proves Lemma 4.1, for any AVL-tree with n nodes and height h then satisfies $n \geq N(h) \geq c^h$, and therefore $h \leq \log_c n \in O(\log n)$. We will give two proofs of Claim 4.1, one that is simple and one that gives the tightest possible bound.

Proof. We prove that $N(h) \geq 2^{h/2} = \sqrt{2}^h$ by induction on h . This clearly holds for $h = 0$ (where the AVL-tree has one node) and $h = 1$ (where the AVL-tree needs at least two nodes to have height 1). For $h \geq 2$, we know that in any AVL-tree of height h both subtrees have height at least $h - 2$. So both subtrees contain at least $N(h - 2)$ nodes, and therefore

$$N(h) \geq 2N(h - 2) \geq 2 \cdot 2^{(h-2)/2} = 2^{h/2}$$

as desired. □

The second proof will give a bound where c is as big as possible.

Proof. Let us study the AVL-trees that have the minimum possible number $N(h)$ of nodes for a given height h . (For reasons that will be clear below, these are called *Fibonacci-trees*.) There is only one tree of height 0 (a singleton-node), so $N(0) = 1$. When the height is 1 (so the root has some children, but no grandchildren), then there are 2 or 3 nodes. But we want the smallest possible numbers of nodes, so $N(1) = 2$.

Consider a Fibonacci-tree T of height $h \geq 2$. There is only a limited number of possibilities for the heights of the subtrees, see Figure 4.5. We can exclude the possibility that both subtrees have height $h - 1$, since the other possibilities use fewer nodes. So one subtree has height $h - 1$ and the other has height $h - 2$. Also, both subtrees should be Fibonacci-trees, otherwise T did not have the minimum number of nodes. With this we know exactly the structure of Fibonacci-trees (except for the choices between left and right subtree), see Figure 4.7.

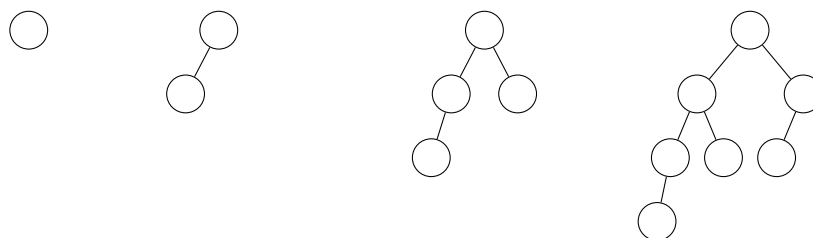


Figure 4.7: Fibonacci-trees of height 0, 1, 2, 3.

This also gives us a recursive formula for $N(h)$: Take the sizes of the subtrees and add one for the root. Therefore

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ 1 + N(h-1) + N(h-2) & \text{if } h \geq 2 \end{cases}$$

In particular, $N(h)$ obeys the following sequence:

$$N(h) = \langle 1, 2, 4, 7, 12, 20, \dots \rangle,$$

which may or may not look familiar. But look at the sequence for $N(h) + 1$:

$$N(h) + 1 = \langle 2, 3, 5, 8, 13, 21, \dots \rangle.$$

This should look familiar! This sequence is (up to an index-shift) the sequence of *Fibonacci numbers*, i.e., the sequence defined by the recursion $F(0) = 0$, $F(1) = 1$ and $F(i) = F(i-1) + F(i-2)$ for $i \geq 2$. Indeed, it is very easy to show by induction on h that $N(h) + 1 = F(h+3)$. With this, we can get a tight bound on $N(h)$, because a closed formula for $F(i)$ is known. Recall the *golden ratio* $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$. One can easily prove by induction that

$$F(i) = \frac{1}{\sqrt{5}}(\phi^i - (1-\phi)^i)$$

and therefore

$$N(h) = F(h+3) - 1 = \frac{1}{\sqrt{5}}(\phi^{h+3} - (1-\phi)^{h+3}) - 1 = \frac{\phi^{h+3}}{\sqrt{5}} - O(1).$$

2730 Notice that equality holds throughout, so we have obtained an exact (and exponential) bound
 2731 for $N(h)$ as desired. \square

2732 4.2.3 AVL tree operations

2733 We now turn to operations in AVL-trees. Operation *search* is exactly the same as in a binary
 2734 search tree, but its worst-case run-time is now $\Theta(\log n)$ since the AVL-tree has height $\Theta(\log n)$.
 2735 But *insert* and *delete* must restore the balances after the tree has been restructured.

2736 **Insert (part I).** To insert in an AVL-tree, we first call the routine *insert* as defined for binary
 2737 search trees (we use *BST::insert* to denote this routine as opposed to *AVL::insert* for the one in
 2738 AVL-trees). Recall that this inserts the key-value pair at a leaf that was newly added, and we
 2739 assume that *BST::insert* returns this leaf *z*.

2740 At any node *v* that is *not* an ancestor of *z*, both left and right subtrees are unchanged, so *v* is
 2741 balanced in the new tree because it was balanced in the old one. So we only need to rebalance (if
 2742 needed) the ancestors of *z*. To do so, we go up in the tree from *z*, and if we find an unbalanced
 2743 node, locally modify the tree to restore balances. We will discuss below in more detail how to
 2744 do this, but for now give the pseudo-code in Algorithm 4.1.

Algorithm 4.1: *AVL::insert(k, v)*

```

1  z ← BST::insert(k, v)           // z is the leaf where k is now stored
2  while z is not NIL do
3      if ( $|z.left.height - z.right.height| > 1$ ) then
4          Let y be the taller child of z
5          Let x be the taller child of y
6          z  $\stackrel{p}{\leftarrow}$  restructure(x, y, z)           // see later
7          break                               // we can argue that we are done
8      setHeightFromSubtrees(z)
9      z ← z.parent

```

2745 A few comments on this code. The command *setHeightFromSubtrees*(*z*) is a shortcut for
 2746 $z.height \leftarrow 1 + \max\{z.left.height, z.right.height\}$, i.e., it computes the height of *z* correctly,
 2747 presuming the height-entries at the left and right subtree are correct. (For this code we might
 2748 as well have written out the shortcut, but in later codes the shortcut will prevent some clutter.)

2749 Next, we did not give specifics how to break ties for *y* and *x*. Observe that for *y* there cannot
 2750 be a tie—we know that *z* is unbalanced, and therefore one subtree has larger height than the
 2751 other. It turns out that for *x* there *also* cannot be a tie, because the tree was changed by an
 2752 insertion. Details are left as an exercise.

2753 We also assume (in line 9) that every node of the tree stores its parent (and that this parent
 2754 has been initialized to NIL, i.e., we can test whether the parent of *v* exists by comparing *v.parent*
 2755 to NIL). It is possible to implement *AVL::insert* without parent-references (hence saving space).
 2756 This requires searching for *k* again (or modifying what *BST::insert* returns) to find the entire
 2757 path from the root to *z* and storing it appropriately (e.g. on a stack). As this is more complicated
 2758 (and no faster in the worst case), we will assume for the rest of this section that we have parent-
 2759 references. But for simplicity of the pseudocode we will assume that these are maintained
 2760 automatically, i.e., every time when we update a left child we also update the parent-reference
 2761 in this child, etc. (In the pseudo-code we will use $\stackrel{p}{\leftarrow}$ for an update where the parent-references
 2762 also need to be changed.)

2763 Finally, we must talk more about the subroutine *restructure*, and the three nodes x, y, z that
 2764 it uses. This is the operation that restores balances around the three nodes, but let us first look
 2765 at the example in Figure 4.8 to see what needs to be done. We have inserted a key-value pair
 2766 with key 8 using *BST::insert*, which places 8 at a leaf z . Note that with this the (stored) height-
 2767 information may be incorrect at the ancestors of z ; we have marked their height-information
 2768 with a ‘?’ to emphasize this. We now go back up from z and check balances and update heights.
 2769 The node that stores 8 is balanced, and so is node that stores 6 (where we update the height),
 2770 but the node that stores 4 is unbalanced.

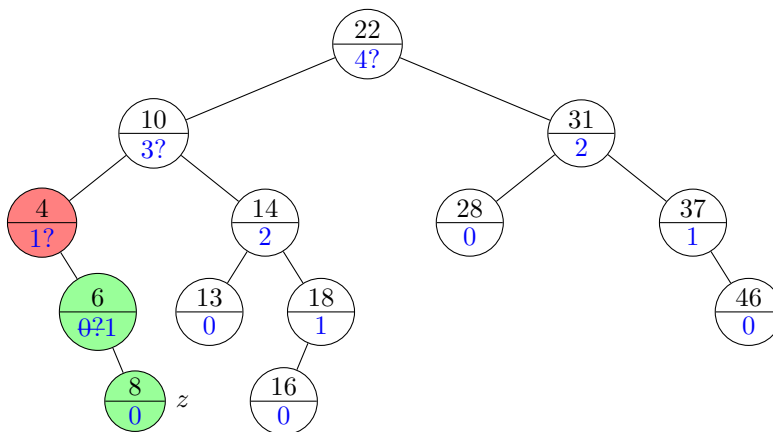


Figure 4.8: The beginning of insertion in an AVL-tree.

2771 The goal for restructuring is to change the subtree rooted at node 4, but leave the rest of
 2772 the tree untouched. Note that there are *many* different ways of storing the same set of keys as
 2773 a binary search tree; the trick will be how to choose what is stored where as to restore balance.
 2774 This is done using the so-called *rotations* that we define next.

2775 **Rotations:** Let T be a binary search tree with a node z that has a child y and a grandchild
 2776 x that is a child of y . A *rotation at z with respect to y and x* is a restructuring of tree T
 2777 such that the result is again a binary search tree, and subtree-references have been changed
 2778 only at x, y and z . There are many possible rotations. For restoring balances at AVL-trees,
 2779 we want the four rotations that make the median of x, y, z the new root of the subtree. For
 2780 example, if $x < y < z$ then we want y to become the new root, if $y < x < z$ then we want x
 2781 to become the new root, etc. Since we know that x, y belong to the same subtree of z (hence
 2782 either $\max\{x, y\} < z$ or $z < \min\{x, y\}$), we have four possible orderings among x, y, z , and
 2783 correspondingly four rotations:

- 2784 • *Right rotation:* $x < y < z$, which by the BST-order means that y is the left child of z and
 2785 x is the left child of y . We want y to become the root, and re-arrange the subtrees at x
 2786 and z correspondingly.

- 2787 • *Left rotation*: $z < y < x$, which by the BST-order means that y is the right child of z and
 2788 x is the right child of y . We want y to become the root.

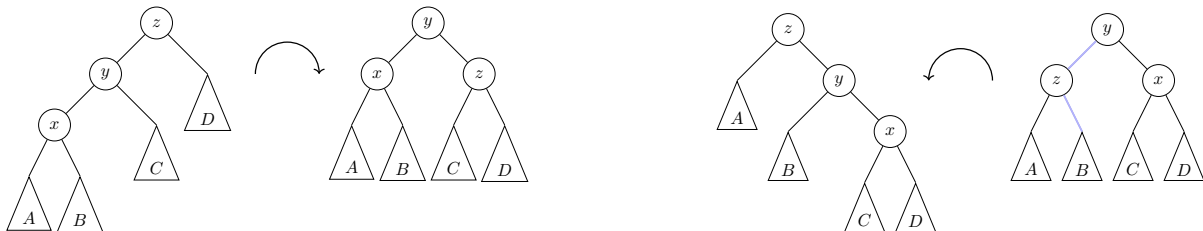


Figure 4.9: A right rotation and a left rotation.

- 2789 • *Double right rotation*: $y < x < z$, which by the BST-order means that y is the left child
 2790 of z and x is the right child of y . We want x to become the root.
 2791 • *Double left rotation*: $z < x < y$, which by the BST-order means that y is the right child of
 2792 z and x is the left child of y . We want x to become the root.

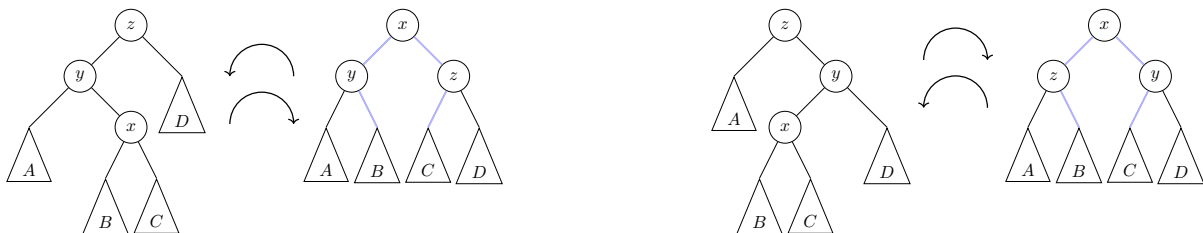


Figure 4.10: A double-right and a double-left rotation

2793 The name “right rotation” comes from the idea that we have pushed the root z towards
 2794 the right and down, hence rotated clockwise. This brings y up to the root, and the rest of
 2795 the subtrees get re-arranged in the unique way that respects the BST-order. The left rotation
 2796 is symmetric. Both rotations are sometimes called *single rotations* to distinguish them from
 2797 the double-rotations. The name “double-rotation” is used because these can be expressed by
 2798 applying two (single) rotations. For example, the double-right rotation can be achieved by doing
 2799 first a single-left rotation at y and then a single-right rotation at z .

2800 We will only give the code for the single-left rotation; the code for the single-right rotation is
 2801 symmetric and double-rotations are expressed via two single-rotations. See Algorithm 4.2 and
 2802 note that it returns the root of the rotated subtree.

2803 Deciding which rotation to apply is now done as explained above: bring the median of x, y, z
 2804 up to become the root of the subtree. See Algorithm 4.3, which is the subroutine that we called
 2805 in *AVL::insert*.

Algorithm 4.2: *rotate-left*(z)**Input :** Node z has a right child y **Output:** Rotate so that y becomes the root of the subtree and return it1 $y \leftarrow z.right$ 2 $z.right \xleftarrow{p} y.left$ 3 $y.left \xleftarrow{p} z$ 4 *setHeightFromSubtrees*(z)5 *setHeightFromSubtrees*(y)6 **return** y

2806 **Insert (part II):** Now that we have seen rotations and how to restructure, let us finish the
 2807 insertion example from Figure 4.8. Recall that we had found an imbalance at the node z with
 2808 key 4. We set y to be its taller child (the node with key 6) and x the taller child of y (the node
 2809 with key 8). Then *restructure* performs a single-left rotation, with the result in Figure 4.11.

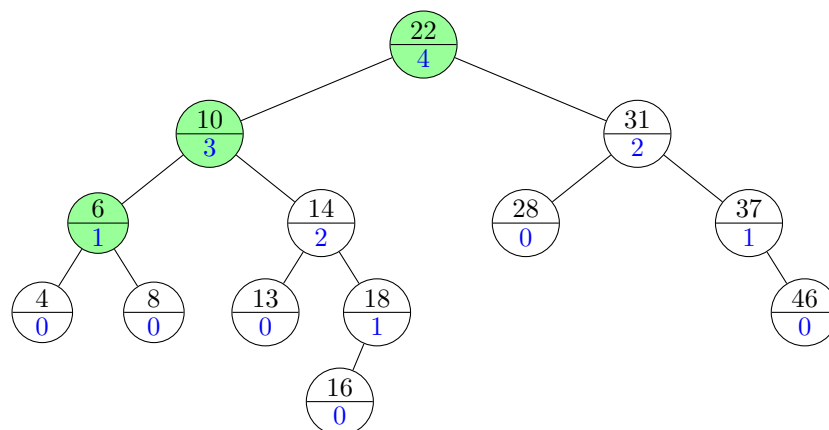



Figure 4.11: Restoring balance with a single-left rotation





2810 This rotation has two effects. First, all of x, y, z are now balanced. Second, the height of
 2811 the subtree with the rotation was reduced. In consequence, nodes 10 and 22 now have the same
 2812 height as they did before the insertion. Therefore both of them are now balanced since they
 2813 were balanced before.

2814 It turns out that the above is *always* true if we restructure the AVL-tree after an insertion:
 2815 The chosen rotation will balance the involved nodes, and will change the height to what it was
 2816 before the insertion. (Claim 4.2 below gives a proof.) So we do not have to check balances of
 2817 ancestors of the node where we rotated: as soon as *AVL::insert* has performed one rotation, the
 2818 AVL-tree property has been restored. This is why we can break out of the while-loop in line 7

Algorithm 4.3: *restructure*(x, y, z)

Input : Node x has parent y and grandparent z
Output: Rotate such that these nodes are in formation  and return new root

```

1 if : (i.e.,  $y = z.left$  and  $x = y.left$ ) then // Right rotation
2 |   return rotate-right( $z$ )
3 else if : (i.e.,  $y = z.left$  and  $x = y.right$ ) then // Double-right rotation
4 |    $z.left \stackrel{P}{\leftarrow}$  rotate-left( $y$ )
5 |   return rotate-right( $z$ );
6 else if : (i.e.,  $y = z.right$  and  $x = y.left$ ) then // Double-left rotation
7 |    $z.right \stackrel{P}{\leftarrow}$  rotate-right( $y$ )
8 |   return rotate-left( $z$ )
9 else // : (i.e.,  $y = z.right$  and  $x = y.right$ ) // Left-rotation
10 | return rotate-left( $z$ )

```

2819 of *AVL::insert*.

2820 **Delete:** Deletion in an AVL-tree is conceptually very similar to insertion. We perform the
 2821 corresponding deletion-routine in the BST-tree. Recall that this has three possible cases, and
 2822 one of them ends up removing a node that might be very far from the node that stored the key k
 2823 to be removed. We need to check for imbalances near the node z that was actually removed (only
 2824 at its ancestors could the balances have changed). So we assume that *BST::delete* returns the
 2825 parent of the node that was actually removed. (In the earlier example of Figure 4.3, algorithm
 2826 *BST::delete* would have returned node 23. In the example of Figure 4.12, algorithm *BST::delete*
 2827 would have returned node 31.)

2828 The rest of the routine is then very similar to *AVL::insert*. We go up from the node z where
 2829 the structural change happened, check its balance, and if needed, determine a suitable child
 2830 and grandchild for restructuring and rotating. However, in contrast to insertion, a rotation will
 2831 not restore the height of the subtree to what it was before. Therefore we do not know whether
 2832 ancestors of the node where we rotated are balanced afterwards; we have to continue upward
 2833 in the tree, check balances, and quite possibly rotate again and again. See Figure 4.12 for an
 2834 example.

2835 Also in contrast to insertion, it is not obvious where we must restructure. Say we have an
 2836 imbalance at node z , and y is its taller child (this is unique since we have an imbalance). If the
 2837 two children of y have different heights, then we use the taller child x for restructuring. However,

Algorithm 4.4: *AVL::delete(k)*

```

1  $z \leftarrow BST::delete(k)$            //  $z$  is the parent of the BST node that was removed
2 while  $z$  is not NIL do
3   if  $(|z.left.height - z.right.height| > 1)$  then
4     Let  $y$  be the taller child of  $z$ 
5     Let  $x$  be the taller child of  $y$  (break ties to prefer a single-rotation)
6      $z \xleftarrow{p} \text{restructure}(x, y, z)$ 
7     // do not break, continue up the path and rotate if needed.
7    $setHeightFromSubtrees(z)$ 
8    $z \leftarrow z.parent$ 

```

2838 if both children of y have the same height, then we *must* use the child of y for restructuring that
 2839 leads to a single rotation (as was done in the second rotation in Figure 4.12). If we used the
 2840 other child of y , then a rotation could lead to a tree that is not an AVL-tree, see Figure 4.13.

2841 **Run-time:** Both *AVL::insert* and *AVL::delete* have worst-case run-time $\Theta(\log n)$. The argu-
 2842 ment for this is the same for both. First observe that the call to *BST::insert* (respectively
 2843 *BST::delete*) takes $O(\log n)$ time because we know that an AVL-tree has height $O(\log n)$. The
 2844 remaining time spent consists of going up from the structural change and performing rotations
 2845 if needed. Since the height is $O(\log n)$, and rotations take $\Theta(1)$ time, the total time for this is
 2846 $O(\log n)$. This is tight if we insert/delete at the lowest level and never rotate.

2847 **Correctness:** The rotations clearly maintain BST-order, so the result is a binary search tree.
 2848 But we must argue that it is an AVL-tree, i.e., all nodes are again balanced. Informally, this
 2849 holds because we choose the grandchild x that has the largest height. Observe that all four
 2850 rotations bring the subtrees at x up by one level. Also note that the “other” subtree at z (i.e.,
 2851 the one not rooted at y) is brought down one level. But this subtree had much smaller height
 2852 (else z would not have been unbalanced), so moving it down should improve the balance at z .
 2853 The formal proof is a bit more complicated (and crucially requires that for *AVL::delete* node x
 2854 was chosen carefully in case of ties).

2855 **Claim 4.2.** *Assume we perform $\text{restructure}(x, y, z)$ during *AVL::insert* or *AVL::delete*. Then*
 2856 *all nodes in the returned subtree are balanced. Furthermore, if the operation was *AVL::insert**
 2857 *then the height of the subtree is restored to what it was before the insertion.*

2858 *Proof.* (cs240e) Consider the time when we are about to perform $\text{restructure}(x, y, z)$, i.e., we
 2859 have already done the structural change for the binary search tree, and perhaps rotated at some
 2860 descendants, but have not yet rotated at z . Consult Figures 4.14 and 4.15 for the following
 2861 definitions and observations. Let the height of the subtree at z at this time be $h + 1$. Since we

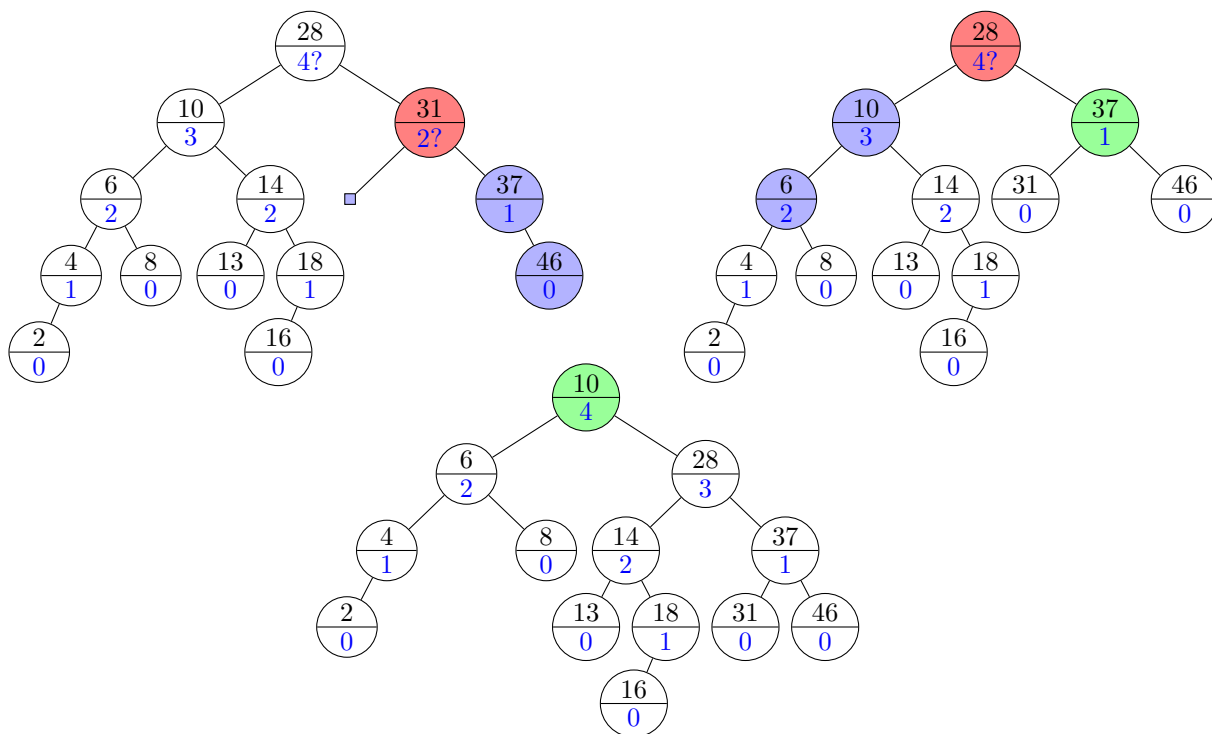


Figure 4.12: AVL-tree deletion may need multiple rotations. After deleting the child of 31, node 31 is unbalanced, so we perform a single-left rotation. This does not fix the imbalance at the root, so we perform a second rotation (a single-right rotation).

2862 chose y and x as tallest children, the subtree at y has height h and the one at x has height $h - 1$.
 2863 We may also, up to symmetry, assume that y is the left child. Let D be the right subtree of z ;
 2864 it has height at most $h - 2$ since z is imbalanced and y has height h . Actually, D has height
 2865 exactly $h - 2$ since z became imbalanced during the last operation, which can change heights of
 2866 subtrees by at most 1. We have two cases, depending on whether x is the left or right child of
 2867 y , i.e., depending on whether we do a single rotation or a double rotation.

2868 **Case 1:** Node x is the left child of y . Let C be the right subtree of y . Since y was balanced
 2869 and x has height $h - 1$, the height of C is $h - 1$ or $h - 2$. Now study the outcome of the rotation
 2870 (see Figure 4.14). Nodes x, y, z are now balanced, and all others in the subtree are balanced
 2871 because they were balanced before.

2872 We claim that if the operation was an insertion, then C actually has height $h - 2$. To see this,
 2873 observe that the insertion must have increased the heights of y and x , because z was previously
 2874 balanced. Thus the height of y was previously $h - 1$, which means that height of C (which is
 2875 unchanged by the insertion) is $h - 2$. With that, if the operation was an insertion then the new

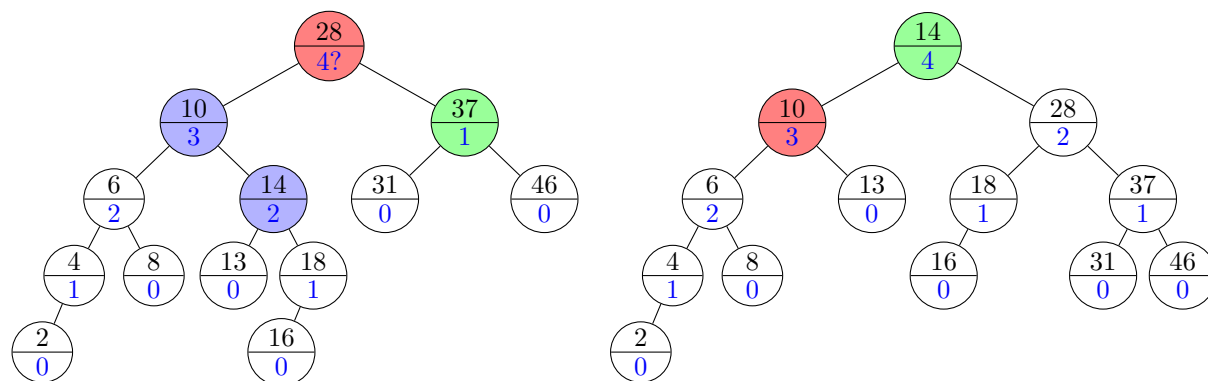


Figure 4.13: If (in the above example) we broke the tie differently, the result would *not* be an AVL-tree. Furthermore, the violation occurs in a subtree of the search-path and would not get corrected later.

subtree has height h , its height prior to the operation.

Case 2: Node x is the right child of y . Let A be the left subtree of y ; since y is balanced A has height $h-1$ or $h-2$. We crucially claim that the height of A is actually $h-2$. If the operation was an insertion, then this is argued the same way as we argued the height of C in the previous case. Now assume the operation was a deletion. Here the method of how we choose x becomes crucial. We know that the subtree at x has height $h-1$. If A had height $h-1$, then there would have been a tie in the choice of x , and so we would have preferred the root of A over x . So A has height $h-2$.

Let B and C be the two subtrees of x ; for each of them we know that the height is $h-2$ or $h-3$ since the height of x is $h-1$ and x is balanced. Again, looking at the rearranged tree, all nodes are balanced and the new subtree has height h , which proves the claim. \square

4.3 Other balanced binary search trees

While the AVL-tree realization of ADT Dictionary is asymptotically the best we can hope for, in practice it is rather slow. There are a number of reasons. One is that we use quite a bit of space-overhead, to store the height (or perhaps the balance) and to store a reference to the parent (or perhaps a stack that stores the insertion/deletion path). Experiments also show that AVL-trees use a lot of rotations (compared to some other realizations), making insertion and deletion slow.

For this reason, many other balanced binary search trees have been developed that improve on one or another of these aspects. The following lists some ideas:

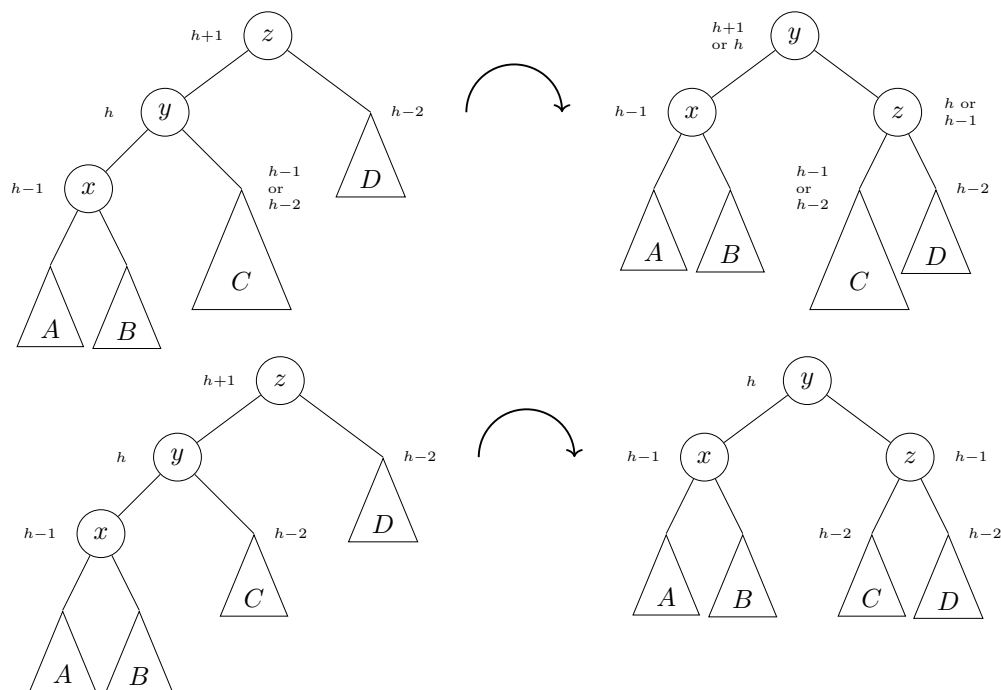


Figure 4.14: Height-changes during a single right rotation. (Bottom) The situation after an insertion.

- We can impose a different structural constraint, which permits bigger height, but still guarantees height $O(\log n)$ and leads to fewer rotations in practice. There are many ways of doing this; the enriched section will see *red-black trees* in Section 11.3.3 after we have studied the closely related 2-4-trees. (These 2-4-trees are also a possible realization of ADT Dictionary, though they are not binary trees.)
- We can relax the condition of $\Theta(\log n)$ time ‘in the worst case’ to $\Theta(\log n)$ time ‘on average’ (i.e., amortized over all operations). The enriched section will see two ways of doing this: *scapegoat trees* occasionally rebuild the entire tree, and *splay trees*, rotate a little bit after every operation. For both the amortized analysis is more complicated than what we did for dynamic arrays, and so we need better tools for doing amortized analysis.
- It turns out that assuming a random input-order, the expected height of a binary search tree is $O(\log n)$. While we cannot trust that the input is distributed enough to achieve this average, this suggests *randomization* to achieve a good expected run-time. There are different ways of doing this; we will see some in Chapter 5.

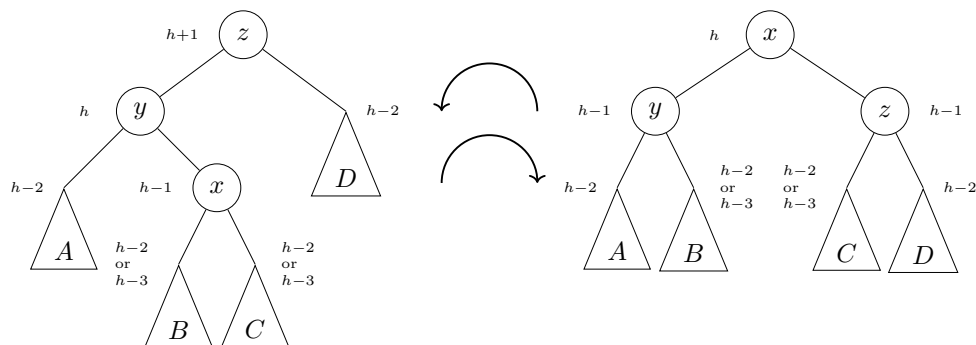


Figure 4.15: Height-changes during a double right rotation.

4.3.1 Scapegoat trees (cs240e)

We now give a different realization of ADT Dictionary, the *scapegoat trees*, which use no rotations and are quite easy to implement. We can show that the amortized run-time of operations is $O(\log n)$, even though the occasional insertion and deletion may have $\Theta(n)$ run-time.

The definition of a scapegoat tree depends on some number α with $\frac{1}{2} < \alpha < 1$. We can choose this number freely (within the range); in our examples we will use $\alpha = \frac{2}{3}$ which according to experiments offers the best run-time.

Definition 4.1. A scapegoat tree is a binary search tree where every node v stores the size of its subtree,¹ and $v.size \leq \alpha \cdot v.parent.size$ for all nodes that are not the root.

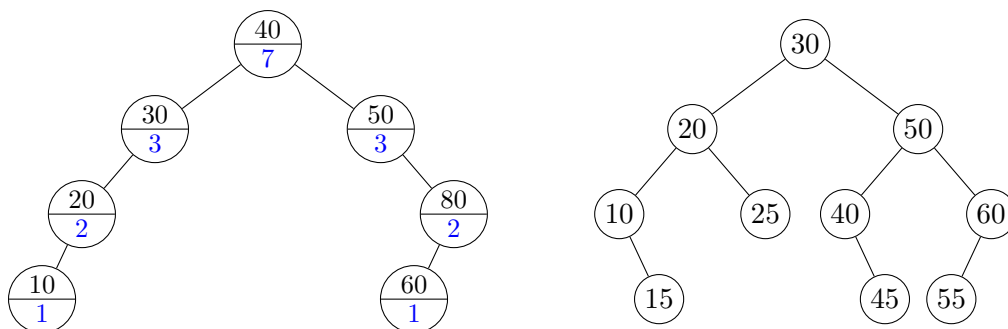


Figure 4.16: A scapegoat tree for $\alpha = \frac{2}{3}$. The lower entries denote the size of the subtree, and one verifies that $p.size \geq \frac{3}{2}v.size$ for any node v with parent p . We also show a perfectly balanced tree of size 10.

¹We assume here that nodes store sizes because this makes the description easier. It is possible to implement scapegoat trees without storing sizes, by checking a height-condition rather than a size-condition when updating. Details are omitted.

Lemma 4.2. *Any scapegoat tree has height $O(\log n)$.*

Proof. At any leaf ℓ (which has size 1), the parent has size at least $1/\alpha$ by definition of a scapegoat tree. Repeating the argument, the grand-parent has size at least $(1/\alpha)^2$, and generally the ancestor that is i levels higher has size at least $(1/\alpha)^i$. Since the root has size n , we hence must have $(1/\alpha)^d \leq n$, where d is the maximum depth of a leaf. Therefore $d \leq \log_{1/\alpha} n \in O(\log n)$. \square

So a scapegoat tree is a balanced binary search tree, and we can do *search*, *insert* and *delete* in logarithmic time. But we must discuss how to restore the structural property. To do so, go back up from where the structural change happened. The nodes on this path are the only ones where the size of the subtree has changed, so these are the only nodes that we need to check. If there is a violation, then let p be the parent at the *highest* such violation, and rebuild the subtree at p to become “perfectly balanced” (defined below). See Algorithm 4.5 for the pseudo-code for *insert* (the one for *delete* is identical except that *BST::delete* is called) and Figure 4.17 for an example.

Algorithm 4.5: *scapegoatTree::insert(k, v)*

```

1  $z \leftarrow \text{BST}::\text{insert}(k, v)$            //  $z$  is the leaf where  $k$  is now stored
2  $S \leftarrow$  stack initialized with  $z$ 
3 while  $p \leftarrow z.\text{parent} \neq \text{NIL}$  do           // get path and update sizes
4   |   increase  $p.\text{size}$ 
5   |    $S.\text{push}(p)$ 
6   |    $z \leftarrow p$ 
7 while  $S.\text{size} \geq 2$  do           // check condition going downward
8   |    $p \leftarrow S.\text{pop}()$ 
9   |   if  $p.\text{size} < \frac{1}{\alpha} \max\{p.\text{left}.\text{size}, p.\text{right}.\text{size}\}$  then
10  |   |   completely rebuild the subtree rooted at  $p$  as perfectly balanced tree
11  |   |   break (out of the while-loop)

```

A *perfectly balanced binary search tree* is a binary search tree where for every node v we have

$$|v.\text{left}.\text{size} - v.\text{right}.\text{size}| \leq 1,$$

i.e., the size-difference between the left and right is as small as possible. See Figure 4.16 for an example. The following is easy to show (details are left as an exercise):

Lemma 4.3. *Given an n -node binary search tree T , we can build a perfectly balanced binary search tree with the same key-value pairs in $\Theta(n)$ time.*

We claim that *scapegoatTree::insert* correctly restores a scapegoat tree. The size-condition could be violated only at ancestors of z . We searched for the highest possible node p where the size-condition is violated. Its entire subtree is rebuilt into a perfectly balanced tree. Such a tree is a scapegoat-tree by $\alpha > \frac{1}{2}$ and the following observation:

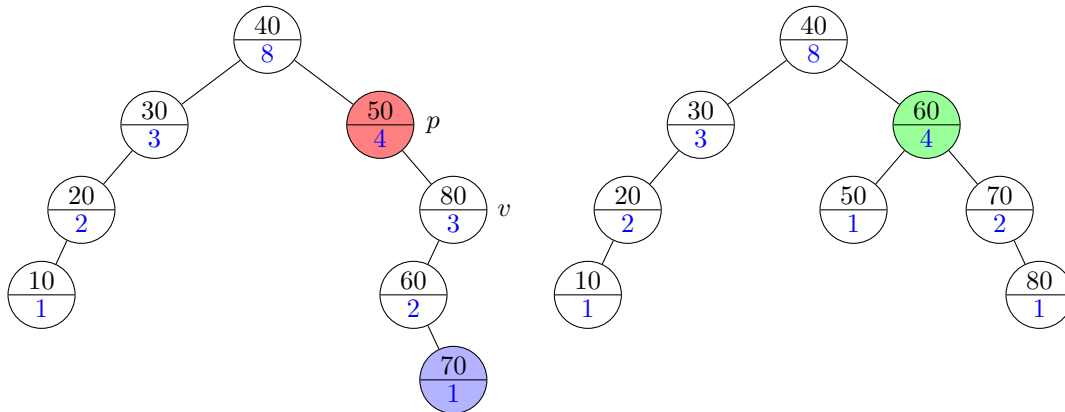


Figure 4.17: Inserting key 70 leads to a violation at the structural property: $p.size < \frac{3}{2}v.size$. We rebuild the entire subtree rooted at p .

2940 **Observation 4.1.** *In a perfectly balanced tree, we have $p.size \geq 2 \cdot v.size$ for any node v with*
 2941 *parent p .*

Proof. By symmetry assume that v is $p.left$. We have $p.right.size \geq p.left.size - 1$, and therefore

$$p.size = 1 + p.left.size + p.right.size \geq 1 + 2 \cdot p.left.size - 1 = 2 \cdot v.size$$

2942 as desired. □

2943 Therefore the size-condition holds everywhere within the subtree at p , and it holds every-
 2944 where else by choice of p .

2945 **Run-time analysis.** Scapegoat trees are very easy to implement, because they do not require
 2946 rotations, and the only helper-routine that we need is to build a perfectly balanced binary search
 2947 tree from stretch, which is easily done in linear time. (The fact that they do not use rotation
 2948 will be extremely useful in later applications in Chapter 8.) Also, most operations will take time
 2949 $O(\log n)$. However, the occasional insertion will cause a subtree to be rebuilt, which is slow.
 2950 Specifically, if we rebuild the subtree rooted at p , then this takes time $O(p.size)$. Since this
 2951 should be rare, we can prove a better amortized time bound. (You may want to review how to
 2952 do amortized analysis from Section 1.5.3.)

We will use the potential-function method for analyzing scapegoat trees. If one is simply given a potential function, then verifying the required conditions $\Phi(\cdot) \geq 0$ and $\Phi(0) = 0$ is usually straightforward, and all we have to do is to compute (and upper-bound) the value

$$T^{\text{amort}}(\mathcal{O}) := \underbrace{T^{\text{actual}}(\mathcal{O})}_{\text{actual run-time}} + \underbrace{\Phi_{\text{after}} - \Phi_{\text{before}}}_{\text{potential-difference } \Delta\Phi}$$

that gives us an amortized bound for operation \mathcal{O} . But how to come up with such a function? We will illustrate the process here (with some incorrect attempts first).

Let us first analyze the actual run-times. As for the amortized analysis of dynamic arrays, it helps to treat *rebuild* (i.e., re-building a subtree into a perfectly balanced binary search tree) as if it were a separate operation, so from now on *insert* or *delete* means “the part of the operation excluding *rebuild*”. The actual run-time of *rebuild* is $\Theta(n_p)$ (where n_p is the size of the sub-tree); assume time units are chosen such that this takes at most n_p time units. The actual run-time of *insert* and *delete* is $\Theta(\log n)$; assume time units are chosen such that they take at most $\log n$ time units.

The main goal is to have a potential function that *decreases* when we do an expensive operation. So look at how the expensive operation changes the structure, and put that change into the potential function. In the case of scapegoat trees, the expensive operation is *rebuild*; say it happens at node p . How does this affect the structure? By definition of perfectly balanced, we know that afterwards the right and left subtree of p have roughly the same size. We claim that before the rebuild, the sizes of these two subtrees was very different.

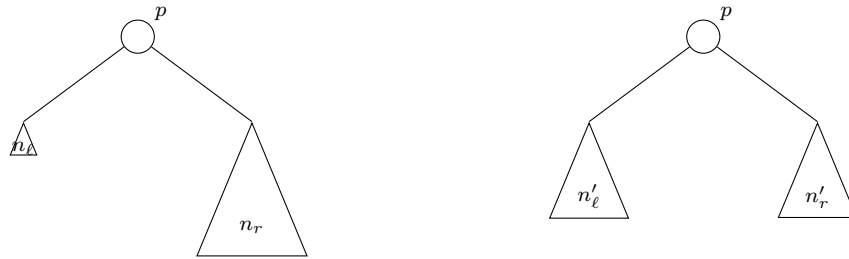


Figure 4.18: A scapegoat tree before and after rebuilding at p .

Claim 4.3. *If we rebuild the sub-tree at p , then $|n_r - n_\ell| \geq (2\alpha - 1)n_p + 1$, where n_p, n_ℓ, n_r denote the sizes of the subtrees at p and its left and right child before the rebuild.*

Proof. We may assume that $n_r \geq n_\ell$, the other case is symmetric. Since we are rebuilding at p , the larger of its children must violate the structural constraint, so $n_r > \alpha n_p$. We also know $n_p = 1 + n_\ell + n_r$. Therefore

$$n_r - n_\ell = n_r - (n_p - n_r - 1) = 2n_r - n_p + 1 > 2\alpha n_p - n_p + 1 = (2\alpha - 1)n_p + 1$$

as desired. □

This gives us an idea of how to define a potential function. Consider

$$\Phi_1(i) := \sum_{v \in S} |v.\text{left.size} - v.\text{right.size}|,$$

2971 where the sum goes over all nodes v in the scapegoat tree S . If we used this as our potential
 2972 function, then the term for p would *decrease* by $\Theta(n_p)$ when we rebuild, while the rebuild costs
 2973 n_p time units. So with an appropriate multiplicative factor this looks promising.

Unfortunately, there is a problem with Φ_1 . While the term at p decreases with a rebuild, the terms for a descendant v of p might *increase*. We know that $|v.left.size - v.right.size| \leq 1$ afterwards, but if this was 0 before and changes to 1, then this was an increase. So let us “make up” for this by decreasing the term for *every* node by 1 in the potential function.² In other words, try

$$\Phi_2(i) := \sum_{v \in S} (|v.left.size - v.right.size| - 1).$$

2974

Now for *every* node the term in the potential-function does not increase, and it still decreases a lot for p . So this looks promising. Alas, we have a different problem now. We need to have $\Phi(\cdot) \geq 0$ at all times. With Φ_2 this is not guaranteed—for example in a complete binary tree each node would contribute -1 . But we can “buffer” for this by making sure every term is non-negative and try

$$\Phi_3(i) := \sum_{v \in S} \max \{0, |v.left.size - v.right.size| - 1\}.$$

This is almost the correct potential function, except as suggested earlier we will need some multiplicative factor because the term at p only decreases by $\Theta(n_p)$. So let us use as final potential function

$$\Phi(i) := c \cdot \sum_{v \in S} \max \{0, |v.left.size - v.right.size| - 1\}$$

2975 for some constant $c > 0$ that we will determine from the analysis below. Whew, that was
 2976 complicated! But at least now you understand where each of the ingredients in this (rather
 2977 bizarre-looking) potential function comes from.

2978 As a convenient shortcut, define Φ^v to be the contribution of v to the sum in Φ , i.e., $\Phi^v =$
 2979 $\max\{0, |v.left.size - v.right.size| - 1\}$. The amortized run-time bounds are now obtained as
 2980 follows:

- Consider first *insert* (the analysis is the same for *delete*). The actual run-time is at most $\log n$ time units. Let z be the node that was added, and x be an ancestor of z . We have $\Phi_{\text{after}}^x \leq \Phi_{\text{before}}^x + 1$, because adding z may or may not have increased the size of the larger subtree at x . At all other nodes w the value of Φ^w is unchanged. Thus the amortized run-time is

$$T^{\text{amort}}(\text{insert}) = T^{\text{actual}}(\text{insert}) + \Phi_{\text{after}} - \Phi_{\text{before}} \leq \log n + c \cdot \#\{\text{ancestors of } z\} \in O(\log n)$$

2981 since the height is logarithmic and c is a constant.

²One could think of other ways of ‘fixing’ Φ_1 , e.g. by using $\lfloor \frac{1}{2}(v.left.size - v.right.size) \rfloor$ as the contribution of node v .

- Now consider a *rebuild* at vertex p . We have

$$\Phi_{\text{before}}^p = \max\{0, \underbrace{|p.\text{left.size} - p.\text{right.size}|}_{\geq (2\alpha - 1)n_p + 1 \text{ by claim}} - 1\} \geq (2\alpha - 1)n_p.$$

Also, since we make the subtree at p perfectly balanced, we have

$$\Phi_{\text{after}}^p = \max\{0, \underbrace{|p.\text{left.size} - p.\text{right.size}|}_{\leq 1 \text{ when perfectly balanced}} - 1\} = 0.$$

2982 Likewise $\Phi_{\text{after}}^v = 0$ for any descendant v of p , while $\Phi_{\text{before}}^w = \Phi_{\text{after}}^w$ for all other nodes w
 2983 (the size of subtrees of w does not change). So each node's contribution to the potential
 2984 function can only decrease, and at p it decreases a lot. Therefore

$$\begin{aligned} T^{\text{amort}}(\text{rebuild}) &= T^{\text{actual}}(\text{rebuild}) + \Phi_{\text{after}} - \Phi_{\text{before}} \\ &\leq n_p + c \underbrace{(\Phi_{\text{after}}^p - \Phi_{\text{before}}^p)}_{\leq -(2\alpha - 1)n_p} \leq (1 - c(2\alpha - 1))n_p \end{aligned}$$

2985 So set $c = \frac{1}{2\alpha - 1}$; this is positive since $\alpha > \frac{1}{2}$. Then $T^{\text{amort}}(\text{rebuild}) \leq 0$, and so the cost
 2986 for rebuilding is paid for by prior insertion and deletion operations.

2987 Summarizing scapegoat trees, most operations should take $O(\log n)$ time, the occasional
 2988 operation may take $\Theta(n)$ time, but the amortized run-time of all operations is $O(\log n)$.

2989 4.4 Take-home messages

- 2990 • ADT Dictionary is *the* data type of relevance when storing large amounts of data in which
 2991 we search by a key.
- 2992 • For a given set of keys, there are many binary search trees that could store them, and
 2993 through rotations we can convert them from one to the other.
- 2994 • Imposing a structural condition on a binary search tree makes search easier (presuming
 2995 we can argue a height-bound), but shifts the burden to *insert* and *delete* which must do
 2996 extra work to restore the condition.
- 2997 • AVL-trees are one possible realization of ADT Dictionary that achieves the asymptotically
 2998 best-possible worst-case run-time. But they are not best-possible in practice.
- 2999 • (cs240e) Scapegoat trees achieve $O(\log n)$ amortized run-time for all operations and never
 3000 use rotations.
- 3001 • (cs240e) Potential functions are a powerful tool for doing amortized analysis.

4.5 Historical remarks

The idea of a binary search tree is quite natural if one is familiar with binary search and writes down the comparisons that could be performed as a decision tree. As such, binary search trees appear to have been discovered independently repeatedly in the 1950s; an early description by Douglas attributes this to Wheeler and Berners-Lee³ [Dou59]. How to update binary search trees under insertions and deletions was apparently also discovered multiple times independently; one of the early descriptions is by Windley [Win60].

AVL-trees were introduced by Adel'son-Vel'skiĭ and Landis in 1962 (hence the name) [AVL62]. Scapegoat trees were introduced (under a different name) in 1972 by Nievergelt and Reingold [NR72]; the name scapegoat trees (and a variant that does not need to store sizes of subtrees) was coined in 1993 by Galperin and Rivest [GR93]. There are many (many!) other variants of balanced binary search trees where the run-time of operations is $\Theta(\log n)$ (at least in an amortized or expected sense); we will see some of them in Chapters 5 and 11 and references to many more can be found in Section 6.2.3 of Knuth's textbook [Knu98].

³No, not the Berners-Lee that you have likely heard about. His father.

Chapter 5

Other dictionary implementations

Contents

5.1	Randomized implementations of ADT Dictionary	141
5.1.1	Treaps (cs240e)	144
5.1.2	Skip lists	147
5.2	Biased search requests	156
5.2.1	Optimal static order in an unsorted list/array	157
5.2.2	Optimal static binary search trees (cs240e)	158
5.2.3	Self-adjusting lists/arrays	158
5.2.4	Self-adjusting binary search trees (cs240e)	162
5.3	Take-home messages	168
5.4	Historical notes	168

This chapter will study further implementations of ADT Dictionary, some of which are variants of binary search trees and some that return to linear implementations (but have other advantages). We will for this chapter still focus on general-purpose dictionaries (i.e., where we have no knowledge about the keys other than that they can be compared); the next chapters will deal with special keys.

5.1 Randomized implementations of ADT Dictionary

While binary search trees have $\Theta(n)$ worst-case run-time for dictionary operations, they are nevertheless quite popular, because they are quite easy to implement, and tend to work well in practice. We now give a theoretical justification of this.

Theorem 5.1. *The expected height of a randomly built binary search tree is in $O(\log n)$. Formally, if we build a binary search tree by inserting the items $\{0, \dots, n-1\}$ in a randomly chosen order, then the expected height of the tree is at most $3 \log(n+1)$.*

Before giving this proof, let us discuss what we learn from the theorem. If the items were

inserted in random order, then the height would be good and a binary search tree would perform well. However, it is too much to expect our insertions to ADT Dictionary to come in random order. Furthermore, the bound is no longer true if we also occasionally delete items. As such, trusting our luck with the instance (while using binary search trees) is questionable. Instead, similarly as we did for *quick-sort* and *quick-select*, we want to use *randomization* so that we have no more bad instances, only (maybe) bad luck. However, it is much less obvious how to do this for binary search trees. We will study this in Sections 5.1.1 and 5.1.2.

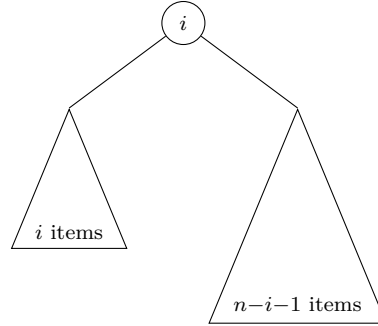
Now we give the proof of Theorem 5.1.

Proof. (cs240e) Let us write $H(\pi)$ for height of the binary search tree built when we use permutation $\pi \in \Pi_n$; here π is the random variable that we will be taking the expected value over. Crucially, we will not bound $H^{\text{exp}}(n) := E[H(\pi)]$ directly, but instead bound $Y^{\text{exp}}(n) := E[Y(\pi)]$, where $Y(\pi) := 2^{H(\pi)}$. Specifically, we will show that $Y^{\text{exp}}(n) \leq (n+1)^3$. With this we then have

$$H^{\text{exp}}(n) = E[H(\pi)] = E[\log(Y(\pi))] \leq \log E[Y(\pi)] \leq \log((n+1)^3) = 3 \log(n+1).$$

Here the first inequality is not at all obvious, but follows from Jensen's inequality (see Appendix A).

For $n = 1$ we have only one permutation π , and $H(\pi) = 0$, hence $Y^{\text{exp}}(1) = 2^{H(\pi)} = 1 \leq 2^3$. We will also recurse into the situation $n = 0$, where there is only the empty permutation, which gives rise to the empty tree which has height -1 . Hence $Y^{\text{exp}}(0) = \frac{1}{2} \leq 1^3$ and the claim holds for $n = 0, 1$. Now let us develop a recursive formula for $Y^{\text{exp}}(n)$ for $n \geq 2$. Fix one permutation $\pi \in \Pi_n$. If the first element of π is i , then i becomes the root of the binary search tree, and the order property dictates which subtrees contains which of the remaining items.



Specifically, π imposes the order $\pi_L \in \Pi_i$ in which $\{0, \dots, i-1\}$ are inserted into the left subtree. It also imposes the order in which $\{i+1, \dots, n-1\}$ are inserted into the right subtree, and after an index-shift, we can view this order as a permutation $\pi_R \in \Pi_{n-i-1}$. Hence for this specific permutation π we have

$$H(\pi) = 1 + \max\{H(\pi_L), H(\pi_R)\} \quad \text{for some permutations } \pi_L \in \Pi_i \text{ and } \pi_R \in \Pi_{n-i-1}.$$

It would now *very* tempting to conclude that $H^{\text{exp}}(n) \leq 1 + \frac{1}{n} \sum_i \max\{H^{\text{exp}}(i), H^{\text{exp}}(n-i-1)\}$, because one would think that taking the expected value on the left and pushing the probabilities

down into the recursive terms would give the formula on the right, similarly as done for example in Lemma 3.2. Alas, this would be correct only if we could interchange taking the sum (over all random outcomes) with the max (over the two height-bounds). Unfortunately this is not true in general, or at least not with the inequality that we need here. So we cannot use this recursive formula for $H^{\text{exp}}(n)$. (The reader is encouraged to go back to the proof of Lemma 3.2 and to verify that there the max was *not* directly exchanged with the sum; instead we handled it there by splitting the sum depending on which term achieves the max.)

So rather than developing a formula for $H^{\text{exp}}(n)$, we instead develop one for $Y^{\text{exp}}(n)$. We know that for the specific permutation π we have

$$\begin{aligned} Y(\pi) &= 2^{H(\pi)} = 2^{1+\max\{H(\pi_L), H(\pi_R)\}} = 2 \cdot 2^{\max\{H(\pi_L), H(\pi_R)\}} \\ &= 2 \cdot \max\{2^{H(\pi_L)}, 2^{H(\pi_R)}\} = 2 \cdot \max\{Y(\pi_L), Y(\pi_R)\} \\ &\leq 2(Y(\pi_L) + Y(\pi_R)) = 2Y(\pi_L) + 2Y(\pi_R). \end{aligned}$$

We have upper-bounded the max by the sum of the two terms (and due to the exponentiation, this will not do too much harm, though this is not obvious). Because taking the expected value can freely be interchanged with taking a sum and/or multiplying with constants, one shows (similarly as in Lemma 1.7) that we have

$$Y^{\text{exp}}(n) \leq \sum_{i=0}^{n-1} \underbrace{\frac{1}{n}}_{\text{probability that } \pi \text{ begins with } i} (2Y^{\text{exp}}(i) + 2Y^{\text{exp}}(n-i-1)) = \frac{4}{n} \sum_{i=0}^{n-1} Y^{\text{exp}}(i),$$

where the last equality holds after some re-arranging of terms. This is not a recursion that we have seen before, but it can easily be evaluated by induction:

$$Y^{\text{exp}}(n) \leq \frac{4}{n} \sum_{i=0}^{n-1} Y^{\text{exp}}(i) \leq \frac{4}{n} \sum_{i=0}^{n-1} (i+1)^3 = \frac{4}{n} \sum_{i=1}^n i^3 = \frac{4}{n} \cdot \frac{(n+1)^2 n^2}{4} = n(n+1)^2 \leq (n+1)^3.$$

This proves the bound for $Y^{\text{exp}}(n)$ and hence on the expected height. \square

Average-case vs. expected revisited (cs240e) Recall that for *quick-select* and *quick-sort*, the average case (for the number of comparisons) could be computed by randomizing and then computing the expected number instead. It may be tempting to think that this always works. Binary search trees will give us a good (and natural) example to show that it does *not*!

Specifically, Theorem 5.1 says that the expected height of a randomly built binary search tree is in $O(\log n)$. This in turn might make you think that ‘the average height of a binary search tree is in $O(\log n)$ ’. This is *false*! The average height of a binary search tree is actually in $\Theta(\sqrt{n})$. We will not prove this here (it is quite complicated, see [FO82]), but we will give here some idea of why this is false.

Call a binary search tree a *min-BST* if the root stores the minimum key. How many binary search trees are min-BSTs? For randomly built binary search trees, this is easy to estimate. The

3084 minimum is at the root if and only if it was the first item to be inserted, so the probability of
 3085 building a min-BST is $\frac{1}{n}$. Put differently, about an n th of the binary search trees that we build
 3086 randomly should be a min-BST.

Now what percentage of all binary search trees (on n items) are min-BSTs? For this, let us write $C(n)$ for the number of binary search trees with n items. It is easy to see that $C(0) = C(1) = 1$ (there is only one such tree) and that

$$C(n) = \sum_{i=0}^{n-1} C(i) \cdot C(n-i-1),$$

because any binary search tree is obtained by fixing the number i of items in the left subtree, taking one of the $C(i)$ possibilities for the left subtree, taking one of the $C(n-i-1)$ possibilities for the right subtree, and combining. The numbers that satisfy this recursion are called *Catalan numbers* and it is known that

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}.$$

With this we can easily determine the number of min-BSTs, because here we know that the right subtree of the root must contain $n-1$ items, and they can be in any arrangement. Therefore there are $C(n-1)$ min-BSTs, and the proportion of min-BSTs is

$$\frac{C(n-1)}{C(n)} = \frac{(2n-1)!}{n!(n-1)!} \cdot \frac{n!(n+1)!}{(2n)!} = \frac{n(n+1)}{2n(2n-1)} \xrightarrow{n \rightarrow \infty} \frac{1}{4}.$$

3087 So (at least for n large enough) roughly a quarter of the binary search trees are min-BSTs.
 3088 Notice the discrepancy to randomly built binary search trees! And also notice that min-BSTs
 3089 will not have a good height on average, because the entire left subtree of the root is empty. Since
 3090 a quarter of binary search trees are min-BST (and symmetrically another quarter are max-BSTs
 3091 where the root stores the maximum and the height is also bad), this gives you an idea of why
 3092 the average height of binary search trees is in $\omega(\log n)$ (i.e., asymptotically bigger).

3093 So what really went wrong here, why do we not have the same bounds for average-case and
 3094 expected? The real reason is that our method of randomly building a binary search tree does
 3095 not accurately reflect the distribution of binary search trees. If you wanted to randomly build
 3096 a binary search tree such that any of them is equally likely, then you would have to choose the
 3097 item i stored at the root with a non-uniform distribution. (Specifically, you would need that the
 3098 probability of using i is $C(i) \cdot C(n-i-1)/C(n)$.) What we analyzed in Theorem 5.1 is correct if
 3099 you think of a randomly (and uniformly chosen) *insertion order*, but not correct if you think of
 3100 a randomly (and uniformly chosen) binary search tree.

3101 5.1.1 Treaps (cs240e)

3102 The first randomized version of a binary search tree that we will see is a *treap* (also known as
 3103 priority search trees). This is a binary search tree, except that every node in a treap stores a
 3104 key-value-pair and *additionally* a *priority* p . The treap has two order-restrictions:

- With respect to key-value pairs, the treap acts like a binary search tree, i.e., if the root stores key k , then everyone in the left subtree has a smaller key, and everyone in the right subtree has a larger key.
- With respect to priorities, the treap acts like a heap, i.e., every node stores the item with the maximum priority among all nodes in its subtree.

This also explains the name “treap”, which is a combination of “tree” and “heap”, since a treap has aspects of both a (binary search) tree and a (binary) heap. See Figure 5.1 for an example.

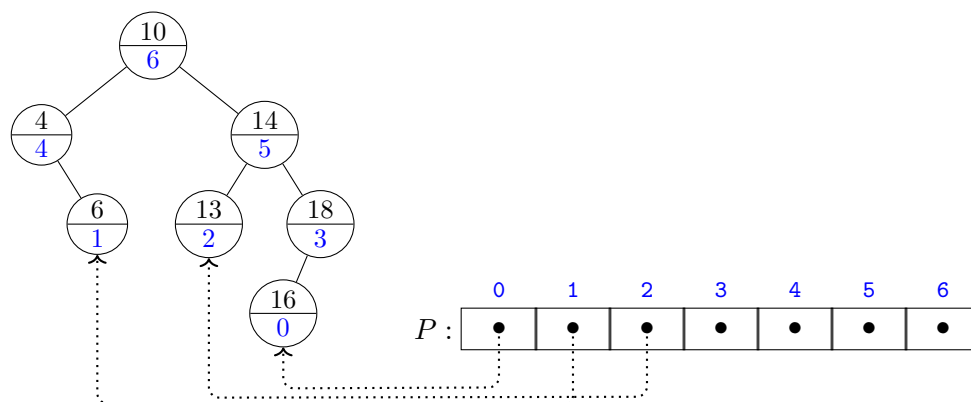


Figure 5.1: A treap. The top entry is the key, the bottom one is the priority. Not all links from P to the treap are shown.

The crucial insight is that the height of the treap is logarithmic if the priorities are chosen suitably.

Observation 5.1. *If the set of priorities in a treap is a permutation of $\{0, \dots, n-1\}$, and this permutation is chosen randomly and uniformly, then the expected height of the treap is $O(\log n)$.*

Proof. This essentially follows from Theorem 5.1, which states that a randomly built binary search tree has height $O(\log n)$. Specifically, the treap corresponds to the binary search tree that would have been built if we had first inserted the item with priority $n-1$, then the item with priority $n-2$, etc. Since this insertion order is chosen randomly and uniformly, the expected height is $O(\log n)$. \square

Operations in a treap

Since a treap is a binary search tree with respect to key-value pairs, *search* is done exactly as for binary search trees and takes time $O(\text{height})$. For *insert*, we also initially proceed as for binary search trees and add the new key-value pair at a leaf z . However, two problems arise: What priority should we assign to z , and how should we restore the heap-order property if this new priority violates it?

To restore the heap-order property, we need to do something like a *fix-up* in our treap. (We know that z is a leaf and that z' will get a bigger priority, so the heap-order property can only be violated upward, not downward.) However, we cannot do *fix-up* directly, since exchanging a node with its parent would violate the binary search tree property. Instead, we use single rotations to exchange the parent-child relationship, see Algorithm 5.1.

Input : Node z where priority may have gotten increased

We need to do *fix-up* once or twice, for the node z and z' . (We do it for z' first because its *fix-up* cannot possibly reach the leaf z and so works correctly even if the order at z has not been fixed yet.) Algorithm 5.2 gives the pseudo-code for *insert* and Figure 5.2 gives an example.

```

1  $n \leftarrow P.size$ 
2  $z \leftarrow BST::insert(k, v); n++$  //  $z$  is the leaf where  $k$  is now stored
3  $p \leftarrow random(n)$ 
4 if  $p < n - 1$  then // Change priority of other node
5    $z' \leftarrow P[p], z'.priority \leftarrow n - 1, P[n - 1] \leftarrow z'$ 
6    $fix\_up\_with\_rotations(z')$ 
7  $z.priority \leftarrow p, P[p] \leftarrow z$ 
8  $fix\_up\_with\_rotations(z)$ 

```

Deletion can be done similarly (but is even more complicated; to achieve a uniformly chosen permutation we must first trade the priority of the node z to be removed with some randomly chosen other node z' (possibly z itself), and then trade the priority of z with the node z'' that has priority $n - 1$ (because this is the priority that must be removed). Along the way, we must

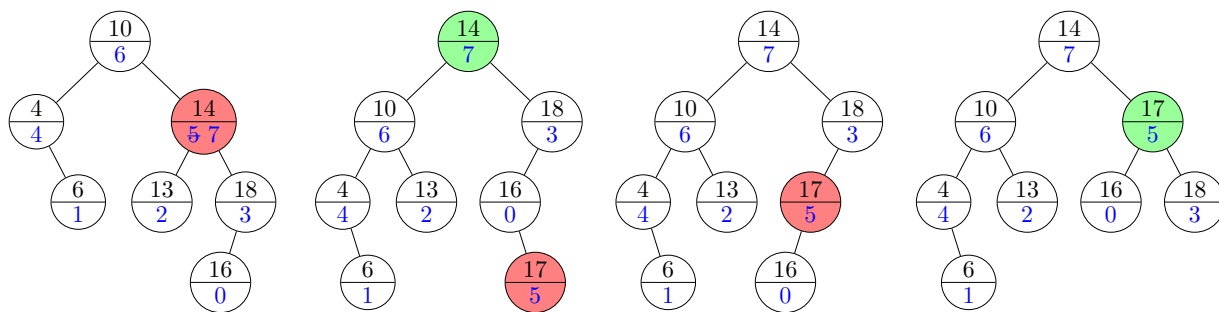


Figure 5.2: Assume that when inserting key 17 we randomly choose $p = 5$. This first rotates the node that previously had priority 5 to the root, and then rotates the new node z up sufficiently high.

restore the heap-order using suitable rotations. We leave the details as an exercise.

Summarizing treaps, they are randomly built binary search trees where the expected height (hence the run-time of all operations) is $O(\log n)$. They have a fairly large space-overhead, since we must store not only the binary search tree, but also parent-references and priorities at each node as well as the array P . As such, treaps are not the best choice of randomized binary search trees, but they are useful in other ways, see Section 8.5. There are other ways of building random binary search trees with slightly less overhead, but we will not cover them here because as it turns out, the best randomized implementation of ADT Dictionary is *not* based on a binary search but instead on an ordered list.

5.1.2 Skip lists

We now give a randomized implementation of ADT Dictionary that is not based on binary search trees but on a sorted list. But it shares with a binary search tree that one key-comparison typically eliminates a fraction of the keys, resulting therefore in logarithmic time to find an element.

Definition 5.1. A skip list is a hierarchy S of ordered linked lists (also called “levels” or “layers”) S_0, S_1, \dots, S_h such that:

- Each list S_i contains the special keys $-\infty$ and $+\infty$ (“sentinels”).
- List S_0 contains the key-value pairs of S in non-decreasing order. (The other lists store only keys.)
- Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$.
- List S_h contains only the sentinels.

A node of a skip list is a node in any of the linked lists in the hierarchy. We assume that each node $p \in S_i$ has two references *after* and *below*, which refer to the next node in S_i , and to the copy of p in list S_{i-1} . Usually there are more nodes than keys, because keys repeat. The *tower*

of a key k is the set of all nodes that contain k (including the node in S_0 that also stores the associated value). The *height* of the tower is the maximum index i such that the tower includes a node in S_i . (So the height counts the nodes that are *above* the node with key k in S_0 , because the node in S_0 always exists.) The *height* of the skip list is the maximum height of a tower, which (due to the sentinel-items) equals the index h of the topmost list. The *root* of the skip list is the leftmost node in the topmost list S_h ; any instance of a skip list has a reference to the root. See Figure 5.3.

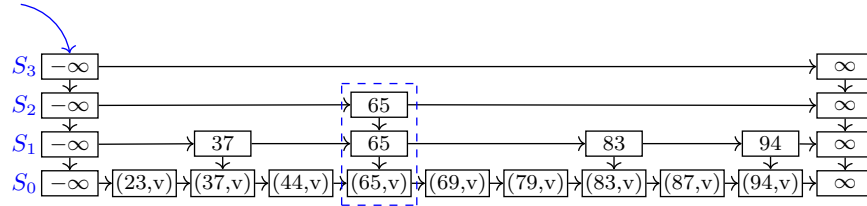


Figure 5.3: A skip list of height 3. The tower of key 65 (indicated) has height 2.

Search: Searching for a key-value pair with key k is straightforward in a skip list. Each list is ordered, so we search in it until we reach the place where k should be. If we searched in list S_i , for $i > 0$, then from there we now “drop down”, i.e., go to list S_{i-1} , because S_{i-1} refines S_i and we must search in its elements. But we do not need to search all of S_{i-1} ; the search in S_i excluded many items already, and so we only need to search the stretch of S_{i-1} between the last two items of S_i that we looked at.

The actual code for *search* will be a bit more complicated than this for two reasons. First, we only have references *after* and *below* in each node; we do not want to keep references to the previous node or the node above to save space. Second, *search* will also be used as subroutine for *insert* and *delete*, and we want to return information from the search that will be required for updates during the latter operations. So we develop a helper-routine *getPredecessors* used by all three operations and shown in Algorithm 5.3.

Algorithm 5.3: *getPredecessors*(k)

```

1  $p \leftarrow \text{root}$ 
2  $P \leftarrow$  stack of nodes, initially containing  $p$ 
3 while  $p.\text{below} \neq \text{NIL}$  do
4    $p \leftarrow p.\text{below}$                                      // drop down
5   while  $p.\text{after}.key < k$  do
6      $p \leftarrow p.\text{after}$                                    // step forward
7    $P.\text{push}(p)$ 
8 return  $P$ 

```

Figure 5.4 gives an example; nodes added to P are blue/dark gray. As the name suggests, *getPredecessors* gets those nodes in the skip list that are predecessors of k . Put differently, for each S_i ($i = 0, \dots, h$) it gets a node that stores a key k' with $k' < k$, and there is no node in S_i that stores a key between k' and k . It stores these predecessors in a stack, with the top item of the stack the predecessor in S_0 . Note that *getPredecessors* does not require that k actually exists in the skip list. Also note that due to the sentinels (and the list S_h that only contains sentinels) the code for *getPredecessors* is kept simple; we do not have to check whether we reach the end of some list because the ∞ -sentinel ensures that we would drop down before this happens.

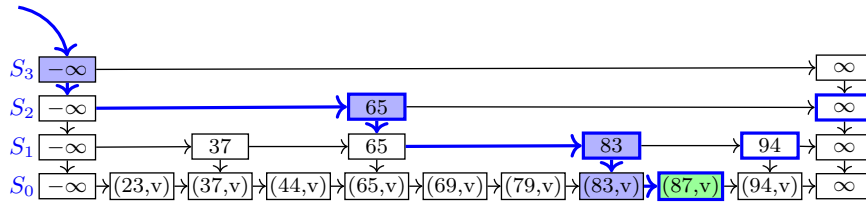


Figure 5.4: Searching in a skip list for key 87. Nodes where we did a comparison are thick, predecessors are blue, and the search-path is thick.

With the ability to find predecessors, the actual *search* routine now becomes very simple. We can look up the predecessor p_0 of the search-key k in S_0 with *getPredecessors*. Thus, either the key-value pair after p_0 stores k , or k is not in the list. See Algorithm 5.4. This also has the advantage that in case that k is not in the skip list, we can determine where it should have been; this will be useful for insertion later.

Algorithm 5.4: *skipList::search(k)*

```

1  $P \leftarrow \text{getPredecessors}(k)$ 
2  $p_0 \leftarrow P.\text{top}()$                                      // predecessor of  $k$  in  $S_0$ 
3 if  $p_0.\text{after.key} = k$  then
4   | return  $p_0.\text{after}$ 
5 else
6   | return “not found, but would be after  $p_0$ ”

```

Insert: To insert a key-value pair (k, v) in a skip list, we first call *getPredecessors*(k). Recall that this returns a stack of nodes that stores for S_0, \dots, S_h the predecessor of k , and so tells us where k would be in S_i for $i \geq 0$. But *should* k be in S_i , i.e., how tall should the tower of k be?

The crucial idea for skip lists is that we choose this tower height *randomly*. In particular, flip a coin until you get tails, and if you got head $i \geq 0$ times, then make the tower of k span the lists S_0, \dots, S_i . (In particular, we *always* insert in S_0 , but whether we insert higher or not

depends on random coin flips.) For future reference, we note that

$$P(\text{tower of key } k \text{ has height } i \text{ or more}) = \frac{1}{2^i},$$

3207 because this happens exactly if the first i coin-flips gave heads, and since we assumed an unbiased
 3208 coin the probability for this is $\left(\frac{1}{2}\right)^i$.

3209 Observe that the tower-height i that we have chosen may *exceed* the height h of the skip list.
 3210 Or it may be that $i = h$, but even then we need to act because we promised that the top-most
 3211 list contains no nodes other than the sentinels. If this happens, then we increase the height of
 3212 the skip list, by adding more lists at the top that contain only sentinels while increasing h , until
 3213 $h > i$. To do this, we need to know the height of the skip list, and it is not clear whether this
 3214 is stored explicitly (we do not need it anywhere else). But we can look up the height h in $O(h)$
 3215 time by dropping down repeatedly, and this does not affect the overall run-time since we use
 3216 $\Theta(h)$ time to get the predecessors later anyway.

3217 Once the skip list has sufficiently much height, the actual insertion is straight-forward: The
 3218 predecessors in the lists S_0, \dots, S_i are the top i items in the returned stack and so we add new
 3219 nodes after them and place k there. Algorithm 5.5 gives the pseudo-code and Figure 5.5 gives
 3220 an example.

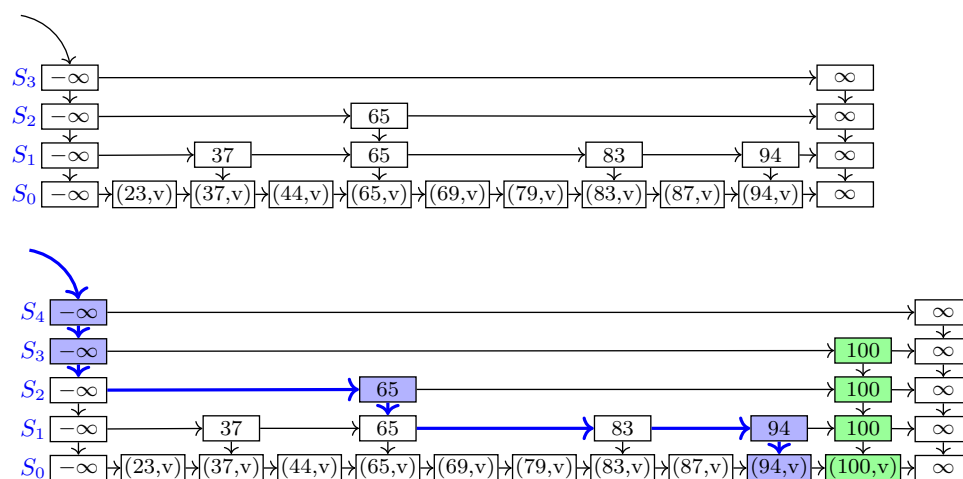
Algorithm 5.5: *skipList::insert(k, v)*

```

1  for ( $i \leftarrow 0$ ;  $\text{random}(2) = 1$ ;) do  $i++$                                 // random tower height
    // Do we need to increase the skip list height?
2  for ( $h \leftarrow 0$ ,  $p \leftarrow \text{root.below}$ ;  $p \neq \text{NIL}$ ;  $p \leftarrow p.\text{below}$ ) do  $h++$           // compute height
3  while ( $i \geq h$ ) do
4      create a new sentinel-only list and link it to the previous root-list appropriately
5       $\text{root} \leftarrow$  leftmost node of this new list
6       $h++$ 

    // Actual insertion
7   $P \leftarrow \text{getPredecessors}(k)$ 
8   $p \leftarrow P.\text{pop}()$                                                     // insert ( $k, v$ ) in  $S_0$ 
9   $z_{\text{below}} \leftarrow$  new node with  $(k, v)$ , inserted after  $p$ 
10 while  $i > 0$  do                                                            // insert  $k$  in  $S_1, \dots, S_i$ 
11      $p \leftarrow P.\text{pop}()$ 
12      $z \leftarrow$  new node with  $k$ , inserted after  $p$ 
13      $z.\text{below} \leftarrow z_{\text{below}}$ ;  $z_{\text{below}} \leftarrow z$ 
14      $i \leftarrow i - 1$ 

```



Delete: Deletion in a skip list is straightforward: find the key (which gives the stack of predecessors) and then remove it from all lists that it was in. We should also clean up in the sense that we do not want unnecessarily many lists, so if deleting a key results in multiple lists that store only sentinels, then delete all but one of them. Algorithm 5.6 gives the code and Figure 5.6 gives an example.

Algorithm 5.6: *skipList::delete(k)*

```

1  $P \leftarrow \text{getPredecessors}(k)$ 
2 while  $P$  is non-empty do
3    $p \leftarrow P.\text{pop}()$  //  $p$  could be predecessor of  $k$  in some list
4   if  $p.\text{after.key} = k$  then remove  $p.\text{after}$  from the list
5   else break // no more copies of  $k$ 
6  $p \leftarrow \text{root}$  // clean up duplicate empty lists
7 while  $p.\text{below} \neq \text{NIL}$  and  $p.\text{below.after}$  is the  $\infty$ -sentinel do
8   remove the list that begins with  $p.\text{below}$ 

```

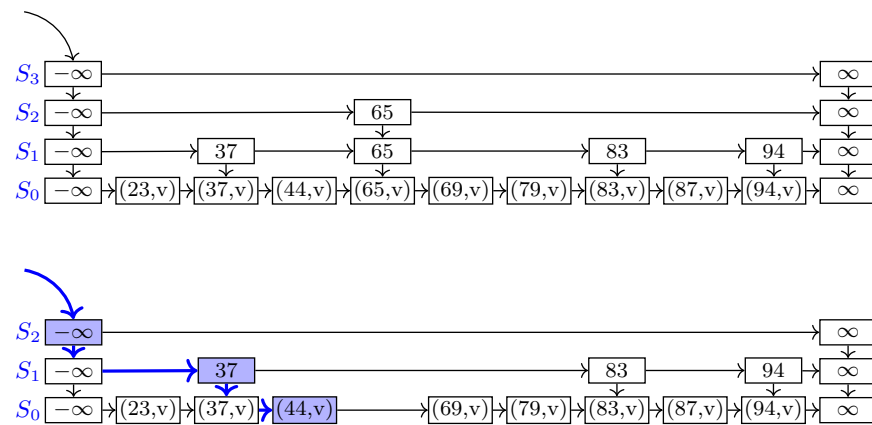


Figure 5.6: Deleting the key-value pair with key 65 from a skip list.

Analysis: In case of really bad luck with the coin-toss, the height of a tower could be arbitrarily large (not bounded relative to n), and so operations could be arbitrarily slow. But this is exceedingly unlikely, and with respect to expected run-time and space-usage, skip lists are good. To show this, we first need a few notations. We use X_k for the random variable that denotes the height of the tower of key k , so $P(X_k \geq i) = \frac{1}{2^i}$ by our choice of the tower-height. Let $|S_i|$ be the number of keys in list S_i , i.e., we do *not* count the sentinels for the length of S_i .

Observation 5.2. *The expected length of list S_i is $n/2^i$.*

Proof. (cs240e) Let $I_{i,k}$ be an *indicator-variable* that is 1 if $X_k \geq i$ (i.e., list S_i includes key k), and 0 otherwise. So $|S_i| = \sum_{\text{key } k} I_{i,k}$ and therefore

$$E[|S_i|] = E\left[\sum_{\text{key } k} I_{i,k}\right] = \sum_{\text{key } k} E[I_{i,k}] = \sum_{\text{key } k} P(X_k \geq i) = \sum_{\text{key } k} \frac{1}{2^i} = \frac{n}{2^i}. \quad \square$$

Lemma 5.1. *The expected height of a skip list is at most $\log n + O(1)$.*

Proof. (cs240e) Let I_i be an indicator-variable that is 1 if $|S_i| \geq 1$ and 0 otherwise. Recall that the height of a skip list is h , where the lists are S_0, \dots, S_h (with S_h the sentinel-only list). Since the h lists S_0, \dots, S_{h-1} all contain keys, therefore $h = \sum_{i \geq 0} I_i$.

We have two upper bounds for I_i : by definition $I_i \leq 1$, and also $I_i \leq |S_i|$. Therefore $E[I_i] \leq \min\{1, \frac{n}{2^i}\}$. The two upper bounds are equal when $i \approx \log n$, so we use this as the place to break up the sum $\sum_i E[I_i]$ to use the upper bound that is better in each part. Therefore

$$\begin{aligned} E[h] = \sum_{i \geq 0} E[I_i] &\leq \sum_{i=0}^{\lceil \log n \rceil - 1} E[I_i] + \sum_{i \geq \lceil \log n \rceil} E[|S_i|] \\ &\leq \sum_{i=0}^{\lceil \log n \rceil - 1} 1 + \sum_{i \geq \lceil \log n \rceil} \frac{n}{2^i} \\ &\leq \lceil \log n \rceil + \sum_{j \geq 0} \frac{2^{\lceil \log n \rceil}}{2^{j + \lceil \log n \rceil}} \\ &\leq \log n + 1 + \sum_{j \geq 0} \frac{1}{2^j} = 3 + \log n \quad \square \end{aligned}$$

Corollary 5.1. *The expected space of a skip list is in $\Theta(n)$. Specifically, the expected number of nodes is $2n + o(n)$.*

Proof. (cs240e) Each list S_i has $|S_i|$ nodes that store keys. Hence the expected number of nodes with keys is

$$E\left[\sum_{i \geq 0} |S_i|\right] = \sum_{i \geq 0} \frac{n}{2^i} = n \underbrace{\sum_{i \geq 0} \frac{1}{2^i}}_{=2} = 2n.$$

There are $2h + 2$ nodes that do not store keys (the sentinels on each list S_0, \dots, S_h), but we have $E[h] \leq \log n + O(1) \in o(n)$ and so the bound holds. \square

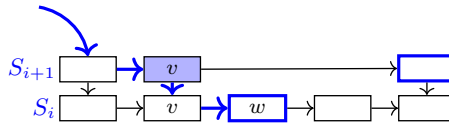
Let us finally analyze the time to do *getPredecessors* (this is the bottleneck in the time for *search*, *insert* and *delete*). Searching for predecessors in a skip-list consists of “dropping down” and “stepping forward” (i.e., going to the next item in the same list). We drop down at most h times, which is $O(\log n)$ expected time. But how often do we step forward?

Claim 5.1. *During getPredecessors , the expected number of forward-steps within list S_i is at most 1.*

Proof. (cs240e) Let v be the leftmost node in list S_i that we visited during the search. If $i = h$ (the topmost list) then we do not step forward at all and are done, so assume $i < h$ and we reached v by dropping down from list S_{i+1} .

Let w be the item after v in S_i . What is the probability that we step forward from v to w ? If w also exists in list S_{i+1} , then we compared (before dropping down to v) the search key k with $w.\text{key}$. We must have had $k \leq w.\text{key}$, else we would not have dropped down from v . Therefore in list S_i we will again immediately drop down.

Taking the contrapositive, if we step forward from v in list S_i , then the next node w in S_i did *not* exist in S_{i+1} . Put differently, the tower of w had height exactly i . The probability of this is $\frac{1}{2}$, because the decision to expand the tower of w into the list above was based on a coin flip.



So we step forward from v with probability at most $\frac{1}{2}$. (It is “at most” and not “equal” because even if w did not exist in S_{i+1} , we might still end up dropping down from v since we may have $k \leq w.\text{key}$). Repeating the argument, we step forward from w with probability at most $\frac{1}{2}$, presuming we arrived at w in the first place, so the probability of this happening is at most $\frac{1}{4}$. Repeating, we see that the probability of stepping forward i times is at most $1/2^i$. Therefore

$$E[\text{number of forward-steps}] = \sum_{\ell \geq 1} P(\text{number of forward-steps is at least } \ell) \leq \sum_{\ell \geq 1} \frac{1}{2^\ell} \leq 1.$$

Claim 5.2. *The expected total number of forward-steps is in $O(\log n)$.*

Proof. (cs240e) Define F_i to be the number of forward-steps on level S_i . We have $E[F_i] \leq 1$ as argued above, but also clearly $F_i \leq |S_i|$ and so $E[F_i] \leq E[|S_i|] = \frac{1}{2^i}$. With exactly the same breaking-apart of the sum as in the proof of Lemma 5.1 we can show that $E[\sum_{i \geq 0} F_i] \in O(\log n)$. \square

Lemma 5.2. *The expected run-time of search, insert and delete in a skip list is $O(\log n)$.*

Proof. First note that the expected run-time for *getPredecessors* is $O(E[h + \sum_{i \geq 0} F_i])$, because it consists of dropping down h times and stepping forward $\sum_{i \geq 0} F_i$ times. By the previous results therefore the expected time for *getPredecessors* is in $O(\log n)$. Once the predecessors are found, all other operations take $O(h)$ time, which is again expected to be in $O(\log n)$. \square

Improvements: If skip lists are implemented exactly as we described them, then they probably would not perform better than a randomized binary search tree in terms of space overhead: we need two references per node (*after* and *below*), and we would expect have $2n$ nodes. But with two minor tweaks skip lists can become very efficient in practice.

First, let us discuss one potential improvement that is not very helpful. In our examples, the sentinels may seem to take an overwhelming amount of space. But this is mostly an artifact of the examples being small. The number of nodes needed for the sentinels is $2h + 2$, which is expected to be in $O(\log n)$. This pales in comparison with the space needed for the keys, and is probably more than made up by not having to test ever whether we have reached the end of the lists.

However, it is a good idea to try to save space by eliminating some other references. In particular, we can avoid using lists for towers (hence one reference for each node). Note that the height of the tower is fixed when inserting a new key and never changes afterwards. We can therefore store the tower of key k as an array T_k of references. Here $T_k[i]$ represents the node of k in S_i ; it stores a reference to T_ℓ (where ℓ was in the next key in S_i), and so we can find the next node at $T_\ell[i]$. See Figure 5.7.

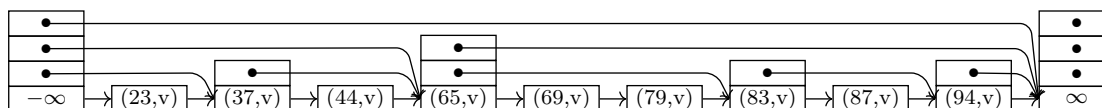


Figure 5.7: A skip list implemented as a list of arrays of references.

A final improvement concerns the probability $\frac{1}{2}$ with which we chose to increase the tower or not. It actually turns out that if this probability is chosen to be $\frac{1}{e}$ (for the Euler constant $e \approx 2.718$), then the expected time for search is less. And one can even show that the total space used is then less than the expected space used by a binary search tree. Details are left as an exercise.

Summarizing skip lists, they are a realization of ADT Dictionary that achieves $O(\log n)$ expected run-time for the operations. When appropriately optimized, they tend to perform better than binary search trees. Skip lists also have the advantage that they are lists (albeit lists of arrays) and so for example merging two skip lists is fairly straightforward, while merging two binary search trees is not.

5.2 Biased search requests

So far, we have assumed that all keys are equally likely to be *accessed* (i.e., be searched for, and the key is in the dictionary). In real life, this is not the case. A rule-of-thumb called the *80/20-rule* states that 80% of the effect is caused by 20% of the causes. Or, translated to the language of ADT Dictionary, 20% of the keys are responsible for 80% of the operations. And certainly there are many applications of ADT Dictionary where such *biased search requests* seem likely. For example, in a customer data base a recent customer will likely order again soon, but customers that have not ordered in a long time have possibly moved away (or even died) and might never order again. In a music collection, some favorite pieces get played over and over again, whereas other pieces are *passé* and only there for historical reasons.

Intuitively, we should put keys that are frequently accessed at a place where they are easy to find (at a small index in an array, or close to the root in a tree). All data structures could be inspected in this light—should we do any changes to make it more likely that a frequently accessed element is in a good place? We will study this for unsorted lists/arrays and binary search trees.

There are two possible scenarios to consider:

- *Static scenario*: We know beforehand how frequently a key is going to be accessed. Put differently, we are given a set of keys k_1, \dots, k_n , and with each key k_i also the *access-frequency* f_i , i.e., how many of the access-requests will be for key k_i . Equivalently, we could also be given which proportion of accesses are for one particular key; this is called the *access-probability*.¹ In our examples we will usually assume that we have the frequencies (because then there are fewer fractions), but the two concepts are equivalent because we can obtain the probabilities by dividing the frequencies by the total number of accesses.

key	A	B	C	D	E
access-frequency	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

Table 5.8: In the static scenario we know the possible keys, and either the access-frequencies or the access-probabilities.

In the static scenario, we can try to find (for a given realization of ADT Dictionary) the *optimum* assignment of keys to locations, i.e., the assignment that minimizes

$$\text{expected access-cost} = \sum_{\text{key } k} \underbrace{P(\text{want to access } k)}_{\text{given}} \cdot (\text{cost of accessing } k)^1$$

¹These probabilities describe the distribution of instances, and have nothing to do with any random choices that an algorithm might take. In particular, ‘expected access-cost’ is *different* from the expected run-time of a randomized algorithm. The name ‘weighted-average-case cost’ would perhaps be more suitable to emphasize the difference.

However, the scenario is fairly unrealistic; we cannot really expect to know how frequently keys are requested.

- *Dynamic scenario:* Here we do not know how frequently a key is going to be accessed. In consequence, we cannot hope to build the best-possible data structure. However, we can still do some heuristics. In particular, we can change the data structure when we have had accesses, to bring those items that were recently accessed to a place where the next access will be cheap. This is inspired by the principle of *temporal locality* (i.e., closeness with respect to time): If we have accessed an item, then it is fairly likely that it will be accessed again soon.

5.2.1 Optimal static order in an unsorted list/array

Let us first consider the static scenario in an unsorted array (an unsorted list can be handled similarly). We use the items from Table 5.8, and consider storing them in order A, B, C, D, E . Is this a good order? To compute the expected access-cost, let us count only key-comparisons (to avoid having to mix arithmetic with asymptotic notation). If key k is at index $idx(k)$ of the array, then searching for k takes $O(1 + idx(k))$ time (we presume that we search from the left, i.e., first compare with index 0, then index 1, etc., until we found k .) Therefore the expected access cost is

$$\sum_{\text{key } k} \underbrace{P(\text{want to access } k)}_{\text{given}} \cdot \underbrace{(1 + idx(k))}_{\text{\# comparisons used to access } k}$$

With this, order A, B, C, D, E has expected access-cost

$$\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \approx 3.31.$$

We could better: Order D, B, E, A, C has expected access-cost

$$\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54.$$

Observe that the second order puts the items in non-increasing order of access-probability. It is intuitive that this should work well, but let us prove this formally.

Claim 5.3. *In a list/array, the static order with the smallest possible expected access-cost is the one that sorts the item by non-increasing access-probability.*

Proof. Let us prove the contrapositive: if we did not put the items in non-increasing order, then a different order would have a smaller expected access-cost. So assume we have keys k_0, \dots, k_{n-1} , with $idx(k_i) = i$, and assume that the order of the keys is not non-increasing. So it is increasing somewhere, say $P(k_{i-1}) < P(k_i)$ for some index $0 < i < n$. Define a new order by exchanging k_i and k_{i-1} , i.e.,

$$k'_j = \begin{cases} k_j & \text{if } j \neq i-1, i \\ k_i & \text{if } j = i-1 \\ k_{i-1} & \text{if } j = i \end{cases}$$

3337 The old order had cost $C = \sum_{j=0}^{n-1} (1+j)P(k_j)$, and the new order has cost $C' = \sum_{j=0}^{n-1} (1+j)P(k'_j)$.
 3338 Therefore

$$\begin{aligned} C' - C &= \sum_{j=0}^{n-1} (1+j) \left(P(k'_j) - P(k_j) \right) \\ &= (1 + (i-1))(P(k_i) - P(k_{i-1})) + (1+i)(P(k_{i-1}) - P(k_i)) = P(k_{i-1}) - P(k_i) < 0. \end{aligned}$$

3339 Hence $C' < C$ and the new order is strictly better. \square

3340 5.2.2 Optimal static binary search trees (cs240e)

3341 One could similarly ask for the best binary search tree that one could have for given access
 3342 probabilities. The expected access cost is now $\sum_k P(k) \cdot (1 + \text{depth of } k)$. One could guess that,
 3343 similarly for an array or list, the maximum-frequency item should be where we search first, i.e.,
 3344 at the root. This turns out to be false!

3345 Consider the access-probabilities in Figure 5.9, and the tree in the middle that would have
 3346 been built with a greedy-algorithm (always use the maximum-probability item as the root, split
 3347 the rest of keys according to the BST-order property, and recurse in the subtrees.) The resulting
 3348 tree has access cost $\frac{48}{26}$. But the right tree has access cost $\frac{47}{26}$.

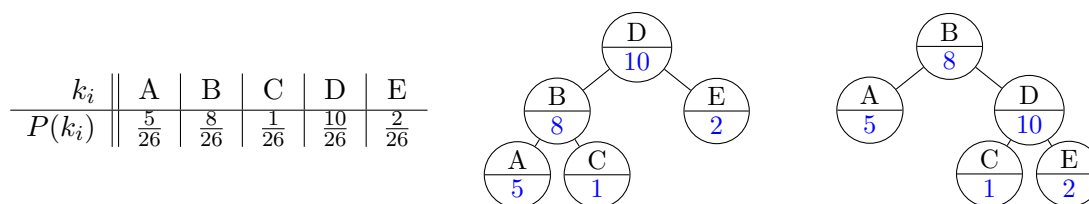


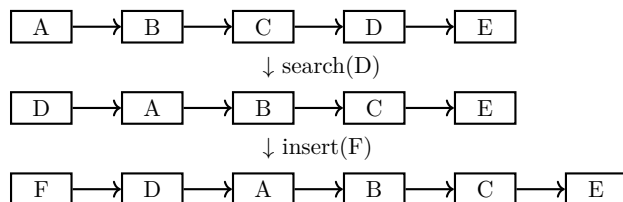
Figure 5.9: A set of access-probabilities and two possible binary search trees for it. For ease of reading we show the frequencies, rather than the probabilities, in the trees.

3349 So the greedy-algorithm does not work! There *is* a polynomial-time algorithm to find the
 3350 optimal binary search tree, but it is based on dynamic programming (a topic that you will see
 3351 in cs341) and will not be covered in cs240.

3352 5.2.3 Self-adjusting lists/arrays

3353 Now let us go into the dynamic scenario, where we do not know the access-probabilities be-
 3354 forehand. So we obviously cannot put the items into any useful order beforehand, but we can
 3355 dynamically change the order as access-requests are received. In particular, move the most
 3356 recently accessed item to a place where it is easy to find.

Move-to-front heuristic in a list: Let the *front* of a list be the place where we begin the search (in our pictures this is the leftmost item). A very natural strategy for dynamically updating unsorted lists is to move the most recently accessed item to the front. This is called the *move-to-front heuristic* or *MTF*.



Doing this update requires very little overhead: if we keep the previous node during the search, then we can remove the node where we found the key in constant time and re-insert it at the beginning of the list. This takes $O(1)$ time. So if there is even the slightest bias in the search requests, then the time needed to do the update will be made up for by the time saved in future searches. As such, the move-to-front heuristic is highly advisable in *any* implementation that uses unsorted lists. (We will see one application in Section 7.3.1.)

Efforts have been made to argue why the move-to-front heuristic is good not only in practice, but also in theory. This may at first sight seem hopeless. Clearly the MTF-heuristic can be very bad: we could always request the item that is farthest from the front (for the list this means ‘at the right end’), and then each search takes $\Theta(n)$ time. But we can observe that in this case all keys were searched for equally often, which means that the optimal static order would *also* have been bad (it would also have taken $\Theta(n)$ time on average to search for a key).

So the way to analyze the move-to-front heuristic (and more generally any *online* algorithm that needs to make decisions without having complete information) is to compare it to the best-possible *offline* algorithm (an algorithm that has complete information). This is called *competitive analysis*.

Lemma 5.3. *The move-to-front heuristic is 2-competitive. Specifically, it takes at most twice as many comparisons that would have been taken using the optimal static ordering, assuming the total number of accesses is big enough.*

Proof. (cs240e) Fix an arbitrary sequence of access-requests. Let us write f_k for the frequency of searching for key k . For $\mathcal{A} \in \{MTF, OPT\}$, let us write $C_{\mathcal{A}}$ for the total number of comparisons done with algorithm \mathcal{A} , and $X_{\mathcal{A}}(A, B)$ for the number of comparisons between keys A and B when using algorithm \mathcal{A} . Observe that

$$C_{\mathcal{A}} = \sum_{\text{key } A} f_A + \sum_{\text{keys } A \neq B} X_{\mathcal{A}}(A, B) \quad (\text{for } \mathcal{A} \in \{MTF, OPT\})$$

because searching for A costs one comparison when we actually find A , and some number of comparisons with other keys when we look at all keys in front of A .

Consider any two distinct keys A and B . Up to symmetry, we may assume that A comes before B in the optimal static order. This implies $f_A \geq f_B$ (the static order sorts by non-increasing frequency). Also

$$X_{OPT}(A, B) = \underbrace{\#\{A-B\text{-comparisons while searching for } A\}}_0 + \underbrace{\#\{A-B\text{-comparisons while searching for } B\}}_{f_B} = f_B$$

because a search for A reaches A before B (so uses no A - B -comparison) while a search for B compares B with A once before reaching B , and there are f_B such searches.

We claim that $X_{MTF}(A, B) \leq 2f_B + 1$. To see this, consider how a sequence of accesses affects $X_{MTF}(A, B)$ and changes the order in the MTF-heuristic. (In the example below, C stands for “any key other than A or B ”.)

operation	change to $X_{MTF}(A, B)$	resulting order with MTF
insert C		neither A nor B is in list
insert A		only key A is in the list
search C		only key A is in the list
search A		only key A is in the list
insert B		key B is at front, hence before A
search C		key B is before A
search A	+1	key A is at front, hence before B (+)
search C		key A is before B
search A		key A is before B
search C		key A is before B
search B	+1	key B is at front, hence before A (*)
search C		key B is before A
search B		key B is before A
search C		key B is before A
search A	+1	key A is at front, hence before B (+)
search C		key A is before B
search A		key A is before B
search C		key A is before B

What operations contribute to $X_{MTF}(A, B)$?

- We compare B to A if we search for B , and A was currently before B in the list. (See comparison (*) in the table.) This can happen at most f_B times, because there are f_B searches for B .
- We compare A to B if we search for A , and B was currently before A in the list. (See comparison (+) in the table.) With the same argument this can happen at most f_A times, but we can get a better bound. Namely, we know that B currently is before A in the list. Therefore, at some earlier time t we inserted or searched for B . Moreover, since time t we have *not* searched for A (else we would have moved it forward). Therefore each such (+)-comparison can be attributed to a (*)-comparison, or to the initial insert of B . Hence there are at most $f_B + 1$ such comparisons.

Summarizing, we have $X_{MTF}(A, B) \leq 2f_B + 1 = 2X_{OPT}(A, B) + 1$, and therefore

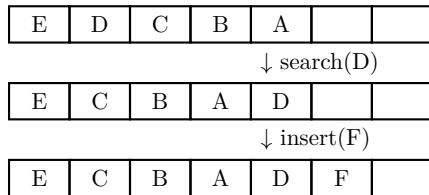
$$C_{MTF} = \sum_A f_A + \sum_{A \neq B} X_{MTF}(A, B) \leq \sum_A f_A + \sum_{A \neq B} (2X_{OPT}(A, B) + 1) \leq 2C_{OPT} - \sum_A f_A + \binom{n}{2}.$$

Note that $\sum_A f_A$ equals the number of accesses, while $\binom{n}{2}$ only depends on the number of keys. So once the number of accesses is at least $\binom{n}{2}$ we have $C_{MTF} \leq 2C_{OPT}$ as required. \square

One can show that this is tight, i.e., there are sequences such that $C_{MTF} \rightarrow 2C_{OPT}$ as the number of accesses goes to infinity. One can also show that the term $\binom{n}{2} - \sum_A f_A$ cannot be avoided in general. Details are left as exercises.

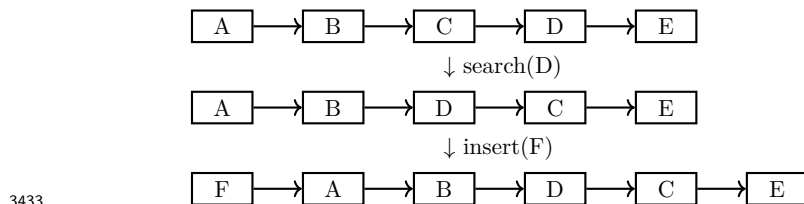
Move-to-front heuristic in an array: The move-to-front heuristic can be implemented in an unsorted array as well, but a little twist is needed. Recall that in an unsorted array, insertion happens at index $n - 1$ while (at least in most standard implementations) we start searching at index 0. This means that a newly inserted item k is very far from the place where we search for it. This is bad because the temporal locality rule suggests that k is likely to be accessed again soon. Moving k to index 0 would be too slow in an array, because we should keep all other items in their relative order, so we would have to move all other items one index to the right (“shuffle k down”), which takes $\Theta(n)$ time.

A better strategy is to change the meaning of ‘front’ for an unsorted array. In other words, change the searching routine so that it starts at index $n - 1$, and then searches at $n - 2, n - 3, \dots$. With this, the front becomes the rightmost item in the array, and a newly inserted item is automatically at the front. The search-routine now follows the move-to-front idea (keeping in mind that the front is at index $n - 1$, this means shuffling the item rightward when found). With this, we have the same functionality as move-to-front in an unsorted list, the only change is that items are in reverse order. So this strategy again is 2-competitive, and the overhead for doing shuffling is small enough that the strategy ought to be applied whenever a dictionary is stored as an unsorted array.



Other heuristics: There are other strategies for dynamically maintaining a list/array. One criticism of the move-to-front heuristic is that the change is very drastic. If we search for one key k that has very low access-probability, then key k will be moved all the way to the front, and will be increasing the access-cost for many other keys before it finally slowly moves to the back again.

3431 To avoid such drastic changes while still adjusting somewhat, one idea is the *transpose*
 3432 *heuristic* where the accessed key is moved towards the front, but only by one index/node.



3434 There are no competitiveness results for the transpose heuristic, and in fact, one can argue
 3435 that it could perform arbitrarily much worse than the optimal static ordering for some sequences
 3436 of access.

3437 One could design other strategies for updating an unsorted list or array. For example, a mix
 3438 of the above two strategies is to move an accessed item about halfway between where it was and
 3439 the front. Another strategy that works very well in practice (but requires extra storage-space)
 3440 is to store with each key how often it has been accessed in the past, and to keep the keys in
 3441 descending order of these frequencies. As more and more requests arrive, the stored frequencies
 3442 will be good approximations of the actual access-frequencies, and so the order will get closer
 3443 and closer to the optimal static order, at the price of having to store an extra integer per key.

3444 5.2.4 Self-adjusting binary search trees (cs240e)

3445 As for lists/arrays, the natural strategy for dynamic updates in a binary search tree would be
 3446 to move the most recently accessed item to the front (the root for a binary search tree). Since
 3447 we must maintain the binary-search-tree property (else future searches will fail), such a move
 3448 has to happen with *rotations*. Thus, the move-to-front heuristic for binary search trees would
 3449 be (when accessing key k) to rotate the node x containing k upward until it is at the root. (Put
 3450 differently, pretend that x has the highest priority and do *fix-up-with-rotations* from the treaps.)
 3451 See Algorithm 5.7 for the code.

Algorithm 5.7: *MTF-BST::insert*(k, v)

```

1  $z \leftarrow \text{BST}::\text{insert}(k, v)$ 
2 while ( $z$  has a parent  $p$ ) do
3   if ( $z = p.\text{left}$ ) then  $z \leftarrow \text{rotate-right}(p)$ 
4   else  $z \leftarrow \text{rotate-left}(p)$ 
  
```

3452 It turns out that a slightly different strategy for doing rotations works better and can be
 3453 analyzed theoretically.

Definition 5.2. A splay tree is a binary search tree where after every operation, we apply zig-zig or zig-zag-rotations (and perhaps one single rotation at the root) so that the accessed item is at the root.

Here *zig-zig* and *zig-zag rotations* are defined as follows. They both operate near a node z that has a parent p and a grandparent g , and they move z two levels up. The *zig-zag rotation* is exactly the same as a double-rotation (see Figure 4.10). It is applied if the path z - p - g forms a “zig-zag”, i.e., contains one left child and one right child. The *zig-zig rotation* is new, and is applied if z - p - g is a *zig-zig*, i.e., either z, p are both left children or z, p are both right children. See Figure 5.10(left). Note that there are two ways to move z two levels up if we are in a zig-zig; Figure 5.10(right) shows the other possibility. It will be *vital* for the analysis that we use the left version, i.e., *first rotate at g and then at p* .

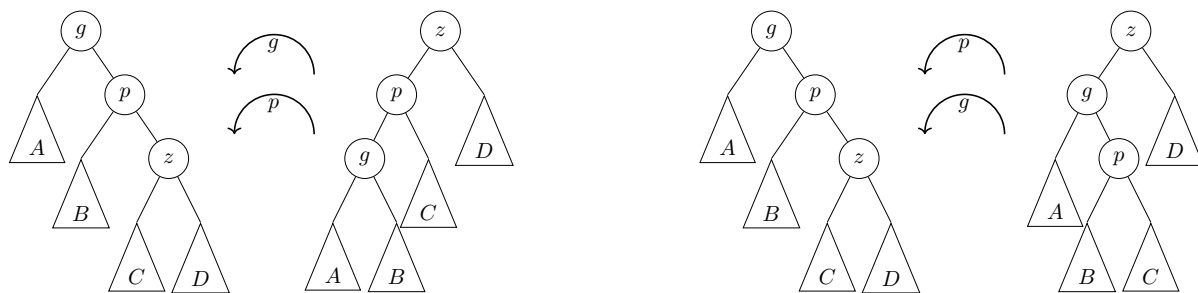


Figure 5.10: Two possible ways of moving z up if both z and p are right children. The left one (the *zig-zig-rotation*) will turn out to be better.

Algorithm 5.8 gives the pseudocode for *insert* and Figure 5.11 gives an example. (The code is almost identical for *search* and *delete*, except that we apply the appropriate BST-operation.) It should be emphasized that we *always* rotate a node z up to the root, even in case of *delete* or an unsuccessful search where node z needs to be the nearest node to where the search ended. This does not make a lot of sense for biased accesses, but is needed to keep the amortized run-time over all operations small.





Analysis: While clearly the worst-case access time in splay trees can be linear, the rotations tend to re-balance the tree, and so overall (i.e., amortized over operations) the run-time of operations will be logarithmic.

Before we prove this, let us get an idea of why splay trees work better than just using the MTF-heuristic by studying one example. Consider a tree T that has height $n - 1$, i.e., it is a path that always goes leftward. If we search for the minimum item, then one can easily verify that with the MTF-heuristic the new tree has height $n - 1$ (though in a different structure), while with the splay-tree rotations, the height is much reduced to $\approx n/2$; see Figure 5.12.

Now we show that the amortized time for an access in a splay tree is $O(\log n)$, using a potential function argument. Let d be the depth at which we found the key (for *search*) or

Algorithm 5.8: *splayTree::insert(k, v)*

```

1  $z \leftarrow BST::insert(k, v)$ 
2 while ( $z$  has a parent  $p$  and a grand-parent  $g$ ) do
3   if : then                                     // Zig-zig rotation rightward
4      $p \leftarrow rotate\_right(g)$ ,  $z \leftarrow rotate\_right(p)$ 
5   else if : then                                     // Zig-zag rotation rightward
6      $g.left \leftarrow rotate\_left(p)$ ,  $z \leftarrow rotate\_right(g)$ 
7   else if : then                                     // Zig-zag rotation leftward
8      $g.right \leftarrow rotate\_right(p)$ ,  $z \leftarrow rotate\_left(g)$ 
9   else // :                                     // Zig-zig-rotation leftward
10     $p \leftarrow rotate\_left(g)$ ,  $z \leftarrow rotate\_left(p)$ 
11 if ( $z$  has a parent  $p$ ) then                                     // single rotation
12   if ( $z = p.left$ ) then  $z \leftarrow rotate\_right(p)$ 
13   else  $z \leftarrow rotate\_left(p)$ 

```

3481 added the key (for *insert*), or did the structural change (for *delete*). Then the operation has
 3482 run-time $O(1 + d)$; we assume time units are such that it takes at most $1 + d$ time units.

For any node v , we use n_v to denote the size of the subtree rooted at v , including v itself. (This is needed *only* for the analysis; we do not need to store it in the data structure). Where appropriate, we write $n_v^{(i)}$ for the size after operation i , or simply n_v^{before} and n_v^{after} if the operation is clear from context. Now define for any splay tree S the function

$$\Phi(i) := \sum_{v \in S} \log n_v^{(i)}.$$

3483 See also Figure 5.13. This clearly is a potential function ($\Phi(i) \geq 0$ and $\Phi(0) = 0$) and hence
 3484 defines an amortized run-time bound. We claim that this amortized run-time is logarithmic,
 3485 and will only argue this for *insert* (the other cases are even simpler).

3486 We consider *insert* (similar as in previous amortized analysis examples) to be multiple sub-
 3487 operations, enumerated as $\mathcal{O}_i, \dots, \mathcal{O}_{i+R}$. Namely, in operation \mathcal{O}_i we search for the correct
 3488 location for the new key and add a new node, call it z . Then we perform $R \geq 0$ rotations
 3489 (zig-zig or zig-zag or single); call these $\mathcal{O}_{i+1}, \dots, \mathcal{O}_{i+R}$. We have $R = \lceil d/2 \rceil$, where as before d
 3490 is the depth of the new node z . To bound the potential-difference for *insert*, we first bound the
 3491 potential-differences for each of the operations $\mathcal{O}_i, \dots, \mathcal{O}_{i+R}$.

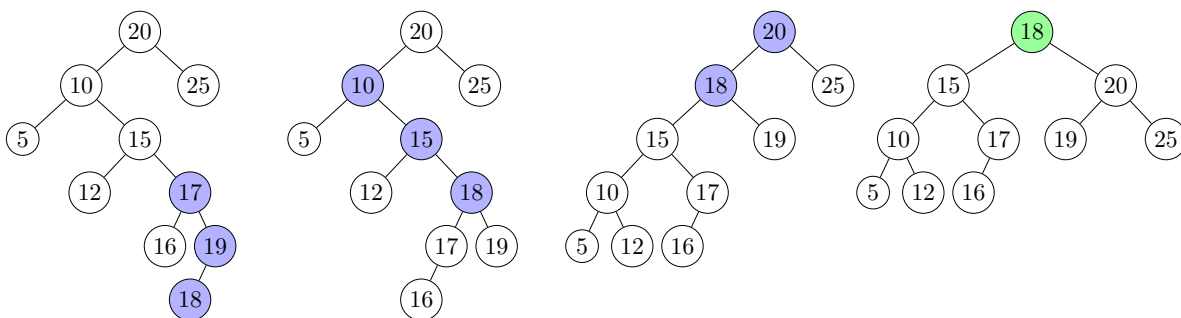


Figure 5.11: After inserting 18, apply zig-zag and zig-zig rotations (and one single rotation at the end) to get 18 to the root.

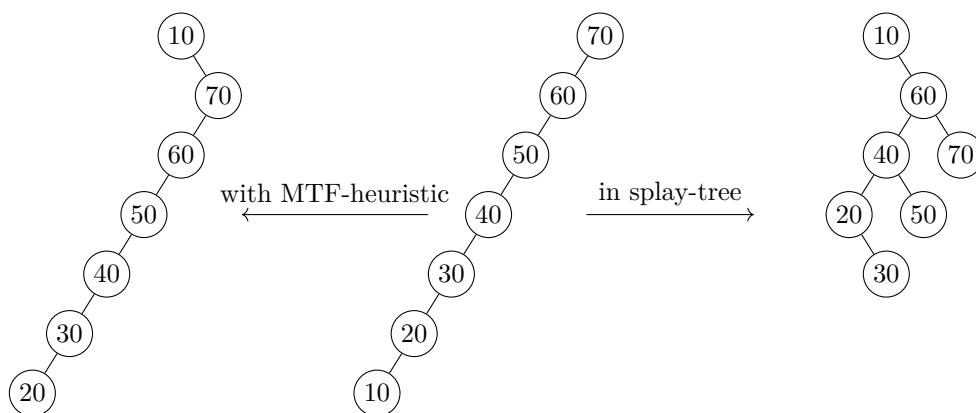


Figure 5.12: A very unbalanced tree (middle) remains unbalanced when accessing 10 with the MTF-heuristic (left), but becomes closer to balanced when using splay-tree rotations (right).

3492 **Claim 5.4.** *The sub-operation \mathcal{O}_i that adds a new node z as a leaf increases $\Phi(\cdot)$ by at most*
 3493 *$\log n$ (where n is the number of items after the insertion).*

Proof. Enumerate the path from z to the root as z, z_1, \dots, z_d , with z_d the root. When adding z , the size of subtrees at all of z_1, \dots, z_d increase by 1, while it is unchanged at all other nodes. So the contribution to the potential-function only changes at z_1, \dots, z_d (and we get a new contribution from the new node z). Since a parent has at least one more node than a child, we

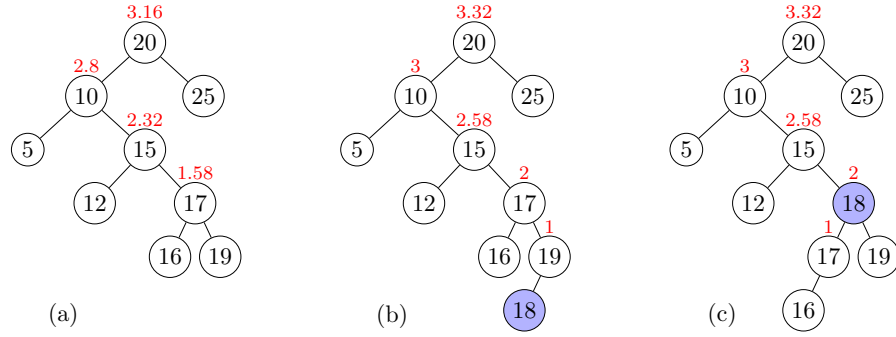


Figure 5.13: Contributions to the potential functions, and how they change during *insert*. Leaves contribute $\log n_v = \log 1 = 0$. (a) $\Phi = 9.86$. (b) After adding key 18, we have $\Phi = 11.9$, hence $\Delta\Phi = 2.04 \leq \log 10$. (c) After the zig-zag rotation, $\Delta\Phi = 0 \leq 3 \cdot 2 - 3 \cdot 0 - 2 = 3 \log n_z^{\text{after}} - 3 \log n_z^{\text{before}} - 2$.

have $n_{z_k}^{\text{after}} = n_{z_k}^{\text{before}} + 1 \leq n_{z_{k+1}}^{\text{before}}$ for $1 \leq k < d$. Hence

$$\begin{aligned}
 \Delta\Phi &= \sum_{v \in S^{\text{after}}} \log n_v^{\text{after}} - \sum_{v \in S^{\text{before}}} \log n_v^{\text{before}} \\
 &= \underbrace{\log n_z^{\text{after}}}_{=1} + \sum_{k=1}^d (\log n_{z_k}^{\text{after}} - \log n_{z_k}^{\text{before}}) \\
 &\leq 0 + \sum_{k=1}^{d-1} (\log n_{z_{k+1}}^{\text{before}} - \log n_{z_k}^{\text{before}}) + \log n_{z_d}^{\text{after}} - \log n_{z_d}^{\text{before}} \\
 &= \log n_{z_d}^{\text{before}} - \underbrace{\log n_{z_1}^{\text{before}}}_{\geq 1} + \underbrace{\log n_{z_d}^{\text{after}}}_n - \log n_{z_d}^{\text{before}} \leq \log n.
 \end{aligned}$$

3494

□

3495 Operations $\mathcal{O}_{i+1}, \dots, \mathcal{O}_{i+R-1}$ are all double-rotations, i.e., either zig-zig or zig-zag. Each
 3496 of them takes the same node z (originally at the insertion-location, but being moved up) and
 3497 moves it two levels higher. The following claim holds for all these double-rotations.

3498 **Claim 5.5.** *Let \mathcal{O}_j be a zig-zig rotation or zig-zag rotation that moves node z two levels up.*
 3499 *Then $\Phi^{\text{after}}(\mathcal{O}_j) - \Phi^{\text{before}}(\mathcal{O}_j) \leq 3 \log(n_z^{\text{after}}) - 3 \log(n_z^{\text{before}}) - 2$.*

3500 *Proof.* The rotation affects the sizes of subtrees only for z , its parent p and its grand-parent g ,
 3501 therefore

$$\begin{aligned}
 \Delta\Phi &= \log n_z^{\text{after}} + \log n_p^{\text{after}} + \log n_g^{\text{after}} - \log n_z^{\text{before}} - \log n_p^{\text{before}} - \log n_g^{\text{before}} \\
 &\leq \log n_p^{\text{after}} + \log n_g^{\text{after}} - 2 \log n_z^{\text{before}}
 \end{aligned}$$

3502 since $n_z^{\text{after}} = n_g^{\text{before}}$ and $n_p^{\text{before}} \geq n_z^{\text{before}}$. Now let us study further how the sizes of subtrees
 3503 change; Figure 5.14 shows again how a zig-zag and a zig-zig rotation operate.

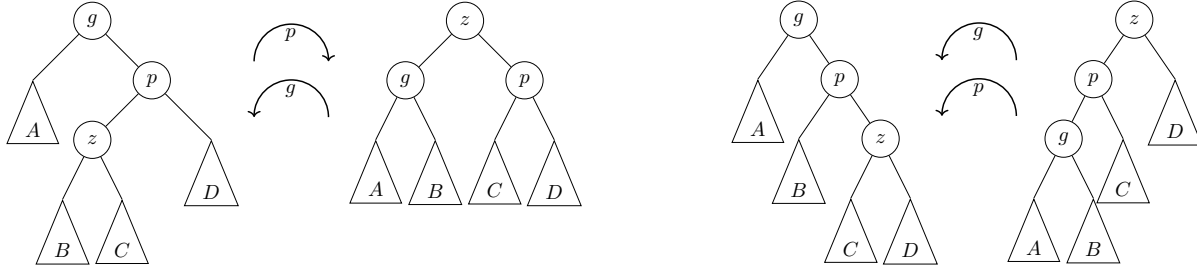


Figure 5.14: A zig-zag rotation and a zig-zig rotation.

3504 Consider first a zig-zag operation. Afterwards, the subtrees of p and g are disjoint (and
 3505 descendants of z), therefore $n_g^{\text{after}} + n_p^{\text{after}} < n_z^{\text{after}}$. Recall that \log is concave (see Figure 2.9 on
 3506 Page 75), so $\frac{1}{2}(\log a + \log b) \leq \log(\frac{a+b}{2}) = \log(a+b) - 1$ or $\log a + \log b \leq 2\log(a+b) - 2$. In
 3507 consequence we have

$$\begin{aligned} \Delta\Phi(\text{zigzag}) &\leq \log n_p^{\text{after}} + \log n_g^{\text{after}} - 2\log n_z^{\text{before}} \\ &\leq 2\log(n_p^{\text{after}} + n_g^{\text{after}}) - 2 - 2\log n_z^{\text{before}} \\ &\leq 2\log n_z^{\text{after}} - 2\log n_z^{\text{before}} - 2 \leq 3\log n_z^{\text{after}} - 3\log n_z^{\text{before}} - 2 \end{aligned}$$

3508 where the last inequality holds since $n_z^{\text{after}} > n_z^{\text{before}}$ and so $\log n_z^{\text{after}} - \log n_z^{\text{before}}$ is positive.
 3509 For the zig-zig operation we have $n_z^{\text{before}} + n_g^{\text{after}} \leq n_z^{\text{after}}$ since the corresponding subtrees are
 3510 disjoint.² Applying concavity of logs therefore $\log n_z^{\text{before}} + \log n_g^{\text{after}} \leq 2\log n_z^{\text{after}} - 2$. We also
 3511 know $n_p^{\text{after}} \leq n_z^{\text{after}}$ and hence have

$$\begin{aligned} \Delta\Phi(\text{zigzag}) &\leq \log n_p^{\text{after}} + \log n_g^{\text{after}} - 2\log n_z^{\text{before}} \\ &\leq \log n_z^{\text{after}} + (2\log n_z^{\text{after}} - 2 - \log n_z^{\text{before}}) - 2\log n_z^{\text{before}} \\ &\leq 3\log n_z^{\text{after}} - 3\log n_z^{\text{before}} - 2 \end{aligned}$$

3512 as desired. □

3513 Operation \mathcal{O}_{i+R} may be a single rotation; very similarly (and even easier) one shows that
 3514 then $\Delta\Phi(\mathcal{O}_{i+R}) \leq 3\log n_z^{\text{after}} - 3\log n_z^{\text{before}}$ (we leave the details as an exercise). With this, we
 3515 can now give the amortized run-time for splay trees.

3516 **Lemma 5.4.** *The amortized run-time of insert in a splay tree is $O(\log n)$.*

²Here is where we need that a zig-zig-operation is used, rather rotating z up with single operations.

3517 *Proof.* The actual run-time for insertion is at most $1 + d$ time units. The potential difference
 3518 is the sum of the potential differences of the sub-operations, which form a telescoping sum that
 3519 we can simplify:

$$\begin{aligned}
 \Delta\Phi(\text{insert}) &= \Phi(i+R) - \Phi(i-1) = \sum_{j=i}^{i+R} (\Phi(j) - \Phi(j-1)) = \sum_{j=i}^{i+R} \Delta\Phi(\mathcal{O}_j) \\
 &\leq \underbrace{\log n}_{\Delta\Phi(\mathcal{O}_i)} + \sum_{j=i+1}^{i+R-1} \underbrace{(3 \log n_z^{(j)} - 3 \log n_z^{(j-1)} - 2)}_{\Delta\Phi \text{ for zig-zig or zig-zag}} + \underbrace{(3 \log n_z^{(i+R)} - 3 \log n_z^{(i+R-1)})}_{\Delta\Phi(\mathcal{O}_{i+R})} \\
 &\leq \log n + 3 \log n_z^{(i+R)} - 3 \log n_z^{(i)} - 2(R-1) \leq \log n + 3 \log n - 2R + 2.
 \end{aligned}$$

Here it is important that *all* rotations are acting on the same node z for bringing upward, so that these terms indeed cancel out. Since $R = \lceil d/2 \rceil$, we have $d \leq 2R + 1$ and so the amortized run-time is at most

$$T^{\text{actual}}(\text{insert}) + \Delta\Phi(\text{insert}) \leq (1 + d) + 4 \log n - 2R + 2 \leq 4 \log n + 4 \in O(\log n)$$

3520 as desired. □

3521 Summarizing splay trees, they are as simple to implement as binary search trees (nodes
 3522 do not to store any extra information such as height or size or balance), and by doing some
 3523 rotations after every insertion, they achieve $O(\log n)$ amortized run-time **and** are self-adjusting.
 3524 The price to pay is that the height may at times be quite large, so that even *search* may take a
 3525 long time. (This is in contrast to scapegoat trees, where the height is *always* $O(\log n)$.)

3526 5.3 Take-home messages

- 3527 • Randomization should be used so that the good average-case for binary search trees can
- 3528 be achieved without having to rely on good input.
- 3529 • Skip lists use randomization to achieve $O(\log n)$ expected run-time.
- 3530 • Biases in the input are frequent, and doing modifications such as move-to-front leads to
- 3531 noticeable improvements in practice.
- 3532 • (cs240e) Without needing to store any additional information, splay trees (i.e., binary
- 3533 search trees plus move-to-root rotations) and achieve $O(\log n)$ amortized run-time and
- 3534 good performance under biased search requests.

3535 5.4 Historical notes

3536 Dictionaries are such an important data structure that research on how to implement them best,
 3537 perhaps tailored to specific scenarios, is ongoing. The presentation-order in this chapter was

done according to how the data structure fits a particular purpose, and was *not* in historical order.

Treaps were introduced in 1980 by Vuillemin [Vui80] under the name *Cartesian trees* for the purposes of storing points (see also Chapter 8). Their use for simulating randomly built binary search trees was given by Aragon and Seidel [AS89]. The presentation here borrows heavily from ideas of Martínez and Roura [MR98] to avoid having to use real numbers for the probabilities.

Skip lists were introduced in 1990 by Pugh [Pug90], who incidentally was also very influential in the development of the programming language Java. Skip lists can actually be made deterministic [MPS92] and then act a lot like the 2-3-4-trees that we will see in Chapter 11.

The concept of biased accesses is quite natural; Knuth's textbook [Knu98, pp.399-401] gives a discussion of some distributions that one might consider and what their optimal access-costs could be. 2-competitiveness of the MTF-heuristic was proved by Bentley and McGeoch [BM85].

Splay trees were introduced in 1985 by Sleator and Tarjan [ST85], but are not the only form of self-adjusting binary search trees. In particular, one can adjust the priorities used in a treap to reflect access frequencies, and this gives a data structure that performs even better in some situations [Knu92].

Chapter 6

Special-key Dictionaries

Contents

6.1	Searching among sorted numbers	171
6.1.1	Lower Bound for search	172
6.1.2	Making binary search optimal (cs240e)	173
6.1.3	Interpolation search	176
6.1.4	Improving and analyzing <i>interpolation-search</i> (cs240e)	179
6.2	Dictionary for Words	185
6.2.1	Preliminaries	185
6.2.2	Binary tries	187
6.2.3	Variations of binary tries	191
6.2.4	Compressed tries	195
6.2.5	Multi-way tries	197
6.3	Take-home messages	199
6.4	Historical notes	199

We have now seen numerous realizations of ADT Dictionary, but all were oblivious to the type of key that was stored in them (except that the key had to be comparable). In this and the next section, we shift the focus to realizations that assume a particular structure of the key, but in exchange, typically achieve faster running times.

6.1 Searching among sorted numbers

You should be familiar with binary search from CS135: Search in a sorted array by comparing with the middle item and then recursing in the appropriate sub array. See Algorithm 6.1. You should also be familiar with that its run-time is $O(\log n)$ (more specifically, its number of recursions satisfies $T(n) \leq T(n/2) + O(1)$).

In this section, we will first argue that we cannot possibly search faster than this, and then proceed to find an algorithm that searches faster. This is not a contradiction because (as

Algorithm 6.1: *binary-search*(A, n, k)

Input : Sorted array A of size $n \geq 1$, key k

```

1  $\ell \leftarrow 0, r \leftarrow n - 1$ 
2 while ( $\ell \leq r$ ) do
3    $m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$ 
4   if ( $A[m] = k$ ) then return “found at index  $m$ ”
5   else if ( $A[m] < k$ ) then  $\ell \leftarrow m + 1$ 
6   else  $r \leftarrow m - 1$ 
7 return “not found, but would be between indices  $\ell-1$  and  $\ell$ ”

```

3585 for sorting) there will be two different models: The lower bound holds for comparison-based
 3586 algorithms, while the improvement assumes that we are searching among numbers and takes
 3587 advantage of this.

3588 6.1.1 Lower Bound for search

3589 So we first show that binary search is asymptotically as fast as possible, at least among comparison-
 3590 based algorithms.

3591 **Theorem 6.1.** *Any comparison-based realization of ADT Dictionary requires $\Omega(\log n)$ compar-*
 3592 *isons to search in n items, even if those items are given in sorted order.*

3593 *Proof.* Consider an arbitrary algorithm \mathcal{A} to search for a key k in n items x_0, \dots, x_{n-1} and
 3594 consider the corresponding decision tree T . (Recall that decision trees were defined in Section 3.3
 3595 and exist for any comparison-based algorithm.) For example, Figure 6.1 shows the decision-
 3596 tree for doing a search for a key k among three keys x_0, x_1, x_2 that are in sorted order, i.e.,
 3597 $x_0 \leq x_1 \leq x_2$.

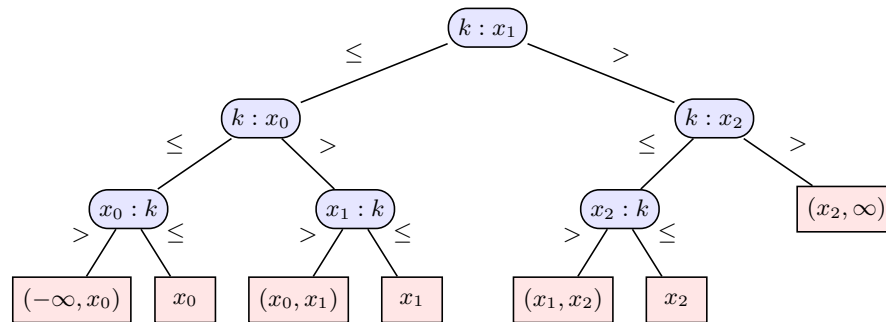


Figure 6.1: A decision tree to search among sorted keys. The output at the leaf is the interval (or unique place) where k can be.

Let us fix $2n + 1$ instances of size n . All instances use the same items with keys x_0, \dots, x_{n-1} (which are in sorted order), and the search-key k is either one of $\{x_0, \dots, x_{n-1}\}$, or falls into one of the $n + 1$ “gaps” between them (including the two gaps outside the range).

Items:	x_0	x_1	x_2	x_3	x_4	x_5	x_6	\dots	x_{n-1}
Search-keys:	\uparrow	\uparrow	\uparrow	\uparrow	\uparrow	\uparrow	\uparrow	\uparrow	\uparrow

These instances lead to at least $n + 1$ distinct outcomes: n outcomes where the answer is ‘key k equals x_i ’, and an outcome with the answer is ‘not found’. So we have $n + 1$ leaves that are reached from these instances. As in the proof of Theorem 3.2, at least one of these leaves is at depth $\log(n + 1)$ and the corresponding instance of *search* requires $\Omega(\log n)$ comparisons. \square

[(cs240e) We can raise the lower bound by a constant amount under the assumption that \mathcal{A} uses no equality-comparison, i.e., always uses one of $<, \leq, \geq, >$. We claim that then *all* $2n + 1$ instances must lead to different outcomes. Assume for contradiction that a search for two of our instances (say with search-values k and k' where $k < k'$) both lead to the same leaf ℓ . This leaf must return ‘not found’, else the algorithm would be incorrect for one of k, k' . We also know that none of the comparisons on the path to ℓ separated k and k' , so these comparisons were all with some value x_i where either $x_i \leq k$ or $k' \leq x_i$. But we know from our construction of instances that at least one key x_j satisfies $k < x_j < k'$. If we execute the search-algorithm for x_j , then this would hence *also* follow the path to ℓ , and return that x_j is not found, which is an incorrect answer. So all our $2n + 1$ instances must lead to distinct leaves.

In consequence, the lowest depth among these leaves is $\log(2n + 1)$, hence by integrality $\lceil \log(2n + 1) \rceil$, and hence some instance requires at least $\lceil \log(2n + 1) \rceil$ key-comparisons. So the worst-case number of comparisons is at least $\lceil \log(2n + 1) \rceil$.

With much the same technique as in Theorem 3.3 one can also show that on average over the instances there are $\Omega(\log n)$ key-comparisons. Details are left as an exercise.

6.1.2 Making binary search optimal (cs240e)

The code for binary search given in Algorithm 6.1 mirrors the version that you have seen in CS135. While this is very intuitive to understand (“look at the item in the middle, and if it’s not there, recurse in the half where the item must be”), it actually is not as fast as binary search can be implemented. The difference is a constant factor, i.e., the algorithm that we will see below still has worst-case run-time $O(\log n)$, but it will typically use half as many key-comparisons, which is a non-trivial improvement in practice.

The problem with converting the idea of binary search into an algorithm is that humans can easily do *3-way comparisons* (“is the item smaller, equal or bigger than the one I am looking for?”). In contrast to this, a comparison in a computer usually is a *2-way comparison* where the outcome is yes/no. (It is our choice whether to compare “ $<$ ” or “ \leq ”.) So if we want to emulate a 3-way comparison, we need to do two comparisons (as done in lines 4-5 of Algorithm 6.1).

But this seems wasteful, since two comparisons would be enough to distinguish between four different scenarios.

So to speed up binary search in practice, one writes a version where we only use one 2-way comparisons, and continue searching even if the key that we just looked at is equal, by including the comparison-index in one of the two sub-arrays. To reduce the number of comparisons to the absolute minimum, we also keep an indicator χ that reports whether we have ever repeated in the left sub-array. See Algorithm 6.2.

Algorithm 6.2: *binary-search-optimized*(A, n, k)

Input : Sorted array A of size $n \geq 1$, key k

```

1  $\ell \leftarrow 0, r \leftarrow n - 1, \chi \leftarrow 0$ 
2 while ( $\ell < r$ ) do
3    $m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$ 
4   if ( $A[m] < k$ ) then  $\ell \leftarrow m + 1$ 
5   else  $r \leftarrow m, \chi \leftarrow 1$ 
6 if ( $k < A[\ell]$ ) then return “not found, but would be between indices  $\ell-1$  and  $\ell$ ”
7 else if ( $\chi = 1$  or  $k \leq A[\ell]$ ) then return “found at index  $\ell$ ”
8 else return “not found, but would be between indices  $\ell$  and  $\ell+1$ ”
```

It is *incredibly* easy to get this algorithm wrong! For example, when computing m , should we round down or round up? Should we use $\ell + r$ or $\ell + r + 1$ to compute m ? And should we compare ‘ \leq ’ or ‘ $<$ ’, i.e., should the middle element, if equal, be included in the left or the right sub-array? If you take the wrong design-decision here, then binary search might well end up not terminating or accessing an item outside the bounds of the array. So let us be extra careful here and argue formally that this algorithm is correct.

Claim 6.1. *Algorithm 6.2 terminates and gives the correct answer.*

Proof. We need a loop-invariant with four parts, where for (B) we are defining $A[-1] := -\infty$ and $A[n] = \infty$.

$$(A) \ 0 \leq \ell \leq r \leq n-1, \quad (B) \ A[\ell-1] < k \leq A[r+1],$$

$$(C) \ k = A[r+1] \text{ implies } k = A[r], \text{ and } (D) \ \chi = 1 \text{ implies } k \leq A[r].$$

Clearly the loop-invariant holds initially by $n \geq 1, \ell = 0, r = n-1, \chi = 0$. We must show that the loop invariant continues to hold when executing the while-loop once. Along the way we also argue that $r - \ell$ gets strictly smaller so that the loop terminates eventually.

So consider a time when we enter the loop, so $\ell < r$. We use primed variables (such as ℓ', r', χ') to denote the value of the variable after this round of the loop. We have $m = \lfloor \frac{\ell+r}{2} \rfloor = \lfloor r - \frac{r-\ell}{2} \rfloor < r$ since $\ell < r$ and we are rounding down. Also $m = \lfloor \frac{\ell+r}{2} \rfloor \geq \ell$ by $r \geq \ell$.

If we repeat in the right sub-array, then we have $\ell' = m+1 \leq r$ since $m < r$. Also $\ell' = m+1 > \ell$ and $r' = r$. Therefore $0 \leq \ell < \ell' \leq r' = r \leq n-1$ as required for (A). By $\ell' > m \geq \ell$ the

sub-array is strictly smaller. Next, $A[\ell'-1] = A[m] < k$ by line 4, and $A[r'+1] = A[r+1] \geq k$, so (B) holds. (C) and (D) hold because it held before and χ and r did not change.

If we repeat in the left sub-array, then we have $\ell' = \ell$ and $r' = m$, and therefore $0 \leq \ell = \ell' \leq m = r' < r \leq n-1$ as required for (A). By $r' = m < r$ the sub-array is strictly smaller. Next, $A[\ell'-1] = A[\ell-1] < k$ and $A[r'+1] = A[m+1] \geq A[m] \geq k$ since A is sorted and line 4 did not apply, so (B) and (D) hold. Finally, $k = A[r'+1]$ can happen only if equality holds throughout the previous inequality, i.e., $k = A[m] = A[r']$, so (C) holds.

Since $r - \ell$ decreases with every round of the loop, we exit the loop eventually. At this point, we have $\ell \geq r$, hence $\ell = r$ by (A). If we return in line 6, then $A[\ell - 1] < k$ by (B) and we checked $k < A[\ell]$, so the answer is correct. If we return in line 7, then $k \leq A[\ell]$ (either explicitly checked or by (D)), but also $k \geq A[\ell]$ since we did not return in line 6, so again the answer is correct. Finally in line 8 by the conditions and (B) we know $A[\ell] < k \leq A[\ell+1]$, and by (C) we know that this makes $k = A[\ell+1]$ impossible. Therefore $A[\ell] < k < A[\ell+1]$ and the algorithm returns the correct answer. \square

We claim that this algorithm uses the exact minimum of possible comparisons, i.e., at most $\lceil \log(2n+1) \rceil$ key-comparisons.

Lemma 6.1. *Algorithm 6.2 uses at most $\lceil \log(2n+1) \rceil$ key-comparisons in the worst-case.*

Proof. Set $L = \lceil \log(2r - 2\ell + 3 - \chi) \rceil$; we will show that at most L comparisons are used, which proves the claim since initially $n = r - \ell + 1$ and $\chi = 0$. Reformulating the definition, we have $2r - 2\ell + 3 - \chi \leq 2^L$ or $r - \ell + 1 \leq 2^{L-1} + \frac{1}{2}(\chi - 1)$, which by integrality means

$$r - \ell + 1 \leq 2^{L-1} + \lfloor \frac{1}{2}(\chi - 1) \rfloor = \begin{cases} 2^{L-1} & \text{if } \chi = 1 \\ 2^{L-1} - 1 & \text{if } \chi = 0 \end{cases}$$

We proceed by induction on $r - \ell$. If $r = \ell$ then $L = \lceil \log(3 - \chi) \rceil$. If $\chi = 0$ then $L = 2$ and we do two key-comparisons. If $\chi = 1$ then $L = 1$ and we do only one key-comparison. Either way the claim holds.

Now consider the case where $r > \ell$ hence $L \geq \lceil \log(2 + 3 - \chi) \rceil \geq 2$. We choose $m = \lfloor \frac{\ell+r}{2} \rfloor$. If we repeat in the right sub-array, then $r' = r$, $\ell' = m+1$, $\chi' = \chi$ and hence

$$2r' - 2\ell' + 3 - \chi' = 2r - \underbrace{2m}_{\geq \ell+r-1} + 1 - \chi \leq r - \ell + 2 - \chi \leq \underbrace{(r-\ell+1)}_{\leq 2^{L-1} + \lfloor \frac{1}{2}(\chi-1) \rfloor} - (\chi-1) \leq 2^{L-1}.$$

By induction we use at most $L - 1$ comparisons in the sub-array, plus one to determine where to repeat, and so at most L comparisons in total as desired.

If we repeat in the left sub-array, then $\ell' = \ell$, $r' = m$, $\chi' = 1$ and hence

$$2r' - 2\ell' + 3 - \chi' = \underbrace{2m}_{\leq \ell+r} - 2\ell + 2 \leq r - \ell + 2 \leq \underbrace{2^{L-1} + \lfloor \frac{1}{2}(\chi-1) \rfloor}_{\leq 0} + 1 \leq 2^{L-1} + 1.$$

3677 If any of the inequalities are strict, then by integrality $2r' - 2\ell' + 3 - \chi' \leq 2^{L-1}$ (and then we
 3678 are done as in the previous case). So we are done unless $\chi = 1$ and $m = \frac{r+\ell}{2}$, so $r + \ell$ is even.
 3679 But then $r - \ell + 1 = r + \ell - 2\ell + 1$ is odd, while $2^{L-1} + \lfloor \frac{1}{2}(\chi - 1) \rfloor = 2^{L-1}$ is even by $L \geq 2$, so
 3680 we have $r - \ell + 1 < 2^{L-1} + \lfloor \frac{1}{2}(\chi - 1) \rfloor$ and one of the three inequalities is always strict. \square

3681 In contrast, it is easy to see that Algorithm 6.1 uses $2 \log(n + 1)$ on some instances (try
 3682 $n = 2^L - 1$ for some integer L and searching for the largest key). So in the worst case, the
 3683 improved version of binary search uses roughly half as many comparisons. However, in the
 3684 best case, the original version of binary search is better: It may find the correct key after two
 3685 comparisons, while the improved version always uses $\Theta(\log n)$ comparisons. One can analyze
 3686 the average case, and show that in fact the improved version beats the original version, even if
 3687 the key is known to exist in the array. Roughly speaking, the original version is better if the
 3688 key is found within the first $\frac{1}{2} \log n = \log(\sqrt{n})$ comparisons, and this can hold for only $O(\sqrt{n})$
 3689 keys. Details are left as an exercise.

3690 One final remark. We have analyzed here the two versions of binary search, because it
 3691 is especially simple to count exactly how many comparisons are used. But a similar comment
 3692 applies to many other situations where we work with keys based on 2-way or 3-way comparisons.
 3693 Examples include the partition-routine for *quick-sort*, *search* in a skip list, and others. In all
 3694 these situations a human doing an example would probably use 3-way comparisons and deal with
 3695 equal items immediately. But a computer-implementation should use 2-way comparisons and
 3696 deal with equality by delaying those comparisons until the end. In this textbook, we sometimes
 3697 use the more intuitive 3-way comparisons (see e.g. the simple variant of *partition*, Algorithm 3.1),
 3698 but more frequently use 2-way comparisons (see e.g. *skipList::getPredecessors*, Algorithm 5.3 or
 3699 *BST::rangeSearch*, Algorithm 8.4) both because it is more efficient in practice and because the
 3700 code tends to be shorter.

3701 6.1.3 Interpolation search

3702 We will improve binary search to make it asymptotically faster (and beat the lower bound),
 3703 under the assumption that the keys stored in a sorted array are numbers. These numbers
 3704 are not necessarily integer, not necessarily positive, and not necessarily distinct, though in our
 3705 examples we will usually use distinct positive integers. To explain the idea, consider the following
 3706 array. Where would you expect key $k = 100$ to be?

3707 Binary search would search half-way between the keys 40 and 120. But under the assumption
 3708 that keys are evenly distributed, we would expect key 100 to be much closer to key 120 than
 3709 to key 40. This is the idea of *interpolation search*: Guess the location of the key based on
 3710 interpolating among the values of nearby keys that we know.

3711 Specifically, assume that at some point during the search for k we know that $A[\ell] \leq k < A[r]$
 3712 for some indices ℓ and r . So key k (if it exists at all) has index somewhere in interval $[\ell, r)$. The
 3713 distance of k to the left end is $k - A[\ell]$, while the distance of the right end to the left end is

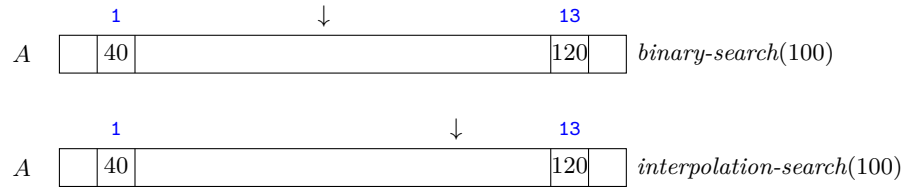


Figure 6.2: Binary search always compares to the middle item, while interpolation search uses the comparison-index where one would expect the search-key to be.

3714 $A[r] - A[\ell]$. So we would expect that a fraction of $p := \frac{k - A[\ell]}{A[r] - A[\ell]}$ of the items between $A[\ell]$ and
 3715 $A[r]$ is no bigger than k . (In the example, $p = \frac{100 - 40}{120 - 40} = \frac{3}{4}$.)

Let us study the similarity with *binary-search*. In binary search, we did not consider the key-value, but simply wanted to search halfway down the array, resulting in a formula of $m = \lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r - \ell) \rfloor$. Now for interpolation search, we want to change the fraction by which we shift. So the following formula to use for m suggests itself:

$$m = \ell + \lfloor p(r - \ell) \rfloor = \ell + \left\lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell) \right\rfloor.^1$$

3716 The idea of interpolation search is to check at index m , and then to proceed as in binary search,
 3717 i.e., recurse in the appropriate side if k was not found. Algorithm 6.3 gives the pseudo-code.
 3718 Notice how the code is extremely similar to *binary-search*, with the main change the definition
 3719 of the index m where we compare. (We also move some of the tests for whether we are done
 3720 inside the loop, since we need the conditions to ensure $A[\ell] \leq k < A[r]$ for the computation of
 3721 m .)

3722 Consider how *interpolation-search*($A[0..10]$, 449) performs on the following example:

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500

3723

- 3724 • Initially $\ell = 0$, $r = n - 1 = 10$, and we check at index $m = \ell + \lfloor \frac{449-0}{1500-0}(10-0) \rfloor = \ell + 2 = 2$.
 3725 But $A[2]$ is too small, so we repeat in the right part.
- 3726 • In the next round, $\ell = 3$, $r = 10$, and we check at index $m = \ell + \lfloor \frac{449-3}{1500-3}(10-3) \rfloor = \ell + 2 = 5$.
 3727 But $A[5]$ is too big, so we repeat in the left part.
- 3728 • In the next round, $\ell = 3$, $r = 4$. We find the key at $A[r]$.

3729 It may also be interesting to study how searching for 448 performs on this example, because
 3730 this clarifies why the while-loop terminates. The first two rounds are the same as in the above

¹(cs240e) As the analysis (in Claim 6.2 below) will show, this is actually *not* the correct formula; we should use “ $r - \ell - 1$ ” rather than “ $r - \ell$ ” and round up, not down, to guarantee the $O(\log \log n)$ expected run-time. In practice this makes no discernible difference, and to keep the pretty similarity with *binary-search* we will use $r - \ell$ here.

Algorithm 6.3: *interpolation-search*(A, n, k)

Input : Sorted array A of n integers, key k

```

1  $\ell \leftarrow 0, r \leftarrow n - 1$ 
2 while ( $\ell \leq r$ ) do
3   if ( $k < A[\ell]$ ) then return “not found, would be between indices  $\ell - 1$  and  $\ell$ ”
4   if ( $k > A[r]$ ) then return “not found, would be between indices  $r$  and  $r + 1$ ”
5   if ( $k = A[r]$ ) then return “found at index  $r$ ”
6    $m \leftarrow \ell + \lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell) \rfloor$ 
7   if ( $A[m] = k$ ) then return “found at index  $m$ ”
8   else if ( $A[m] < k$ ) then  $\ell \leftarrow m + 1$ 
9   else  $r \leftarrow m - 1$ 

```

// Can argue that we never reach this point.

example. Once $\ell = 3$ and $r = 4$, and we check at index $m = \ell + \lfloor \frac{448-3}{449-3}(4-3) \rfloor = \ell = 3$. But $A[\ell]$ is too small, so we repeat in the right part. In the next round, we have $\ell = 4 = r$, but still the while-loop executes one more time. Now if $\ell = r$ then one of the three conditions in lines 3-5 must be true, and so we exit with the appropriate one (in this case, the one in line 3).

Interpolation search is one of those algorithms where the idea is straightforward, but the implementation details are easy to get wrong. For example, it is much less obvious that the sub-array must get smaller every time. So the details of the correctness are non-trivial (but omitted here).

Interpolation works very well in the average-case. We first need to clarify what “average-case” means here, since again we have infinitely many instances of size n (and since we allow arbitrary numbers, not just integers, there are even uncountably infinitely many). To handle this, we assume that the instances are *uniformly distributed*. This means that we think of the (possibly uncountably infinitely many) instances of size n as a universe, and as the instance to have been chosen uniformly at random from this universe. (This may be a bit confusing initially, because this ‘random choice’ has *nothing* to do with any randomness that may exist inside the algorithm. In particular, we continue to analyze the *average-case* run-time (as opposed to the expected run-time), because interpolation search does not use any randomness.) We actually encountered a similar approach already in Section 5.2 when we studied biased search requests and defined the access probabilities in the specific situations where the instances (i.e., search requests) were not uniformly distributed.

Specifically for interpolation search, we hence think of the keys as having been chosen independent and uniformly from some interval, and ask what the average run-time is under this assumption. To state our result, we should briefly discuss the function $\log \log n$, which stands for $\log(\log n)$. This function grows exceedingly slowly, for example if $n = 2^{32} = 4,294,967,296 \approx 4$ billion then $\log \log n = \log 32 = 5$. We state the result here without proof; the next subsection proves such a bound for a closely related algorithm.

Lemma 6.2. *If the keys in the array are uniformly distributed, then the average-case run-time of interpolation-search is $\Theta(\log \log n)$*

However, the worst-case run-time of *interpolation-search* is $\Theta(n)$, for if the numbers are very unevenly distributed, then each recursion might reduce the size of the array by only 1. Consider the following example:

	0	1	2	3	4	5	6	7	8	9	10	
	0	1	2	3	4	5	6	7	8	9	11 ¹¹	<i>interpolation-search</i> (11)
	\uparrow											
	m											

(or generally $A[0..n-1] = \langle 0, 1, 2, 3, \dots, n-2, n^n \rangle$ and imagine that we search for n). Because $A[n-1]$ is huge, the denominator in the fraction for m is huge, so the fraction p is tiny and due to rounding down we have $m = \ell$. The item at $A[m]$ is too small, so we recurse in $A[1..n-1]$. But the argument repeats and repeats, and we always reduce the sub-array size by only 1, resulting in $\Theta(n)$ run-time.

It is possible to do a binary search and interpolation search “in parallel”: compare the key both with $A[m]$ as in *interpolation-search* and with $A[(\ell+r)/2]$ as in *binary-search*, and recurse in the appropriate sub-array. This makes the code much more complicated, but ensures that the sub-array never has size more than $n/2$ and so has worst-case run-time $\Theta(\log n)$ while the average-case run-time is still $\Theta(\log \log n)$.

6.1.4 Improving and analyzing *interpolation-search* (cs240e)

It turns out that with a minor twist to *interpolation-search*, we can both improve the worst-case run-time and make it easier to analyze. The main change is that we *force* the sub-array for the next round to have size $\approx \sqrt{n}$. To do so, we scan the array (starting from $A[m]$) until we find a sub-array of appropriate size that covers all possible locations for k . Finding this sub-array will take multiple *probes* (comparisons of k with $A[j]$ for some index j); the challenge is then to bound the number of probes.

Algorithm 6.4 shows the pseudo-code for *interpolation-search-improved*. The main change is in lines 8-12, where we change the left index ℓ until the sub-array of size $\approx \sqrt{n}$ that starts at ℓ satisfies the conditions for the next round. A few other changes were also done. First, we break the computation of index m up using some parameters μ, N, p , which will be useful in the analysis below. Second, we use $N = (\ell - r - 1)$ (rather than $(\ell - r)$) to compute m and we round up, not down; as the analysis will show this is the correct choice. Finally we can move the three ‘if’-checks to outside the loop; this much improves the actual number of key-comparisons and is possible because (as for *binary-search-optimized*) we keep the compared item in one of the sub-arrays and therefore always know $A[\ell] \leq k < A[r]$.

To see an example, consider again the array $\langle 0, 1, 2, 3, \dots, n-2, n^n \rangle$, when we search for n . (In the example below, $n = 11$.) In the first round, we have a tiny p , hence $m = \ell$ and $A[m]$ is

Algorithm 6.4: *interpolation-search-improved*(A, n, k)

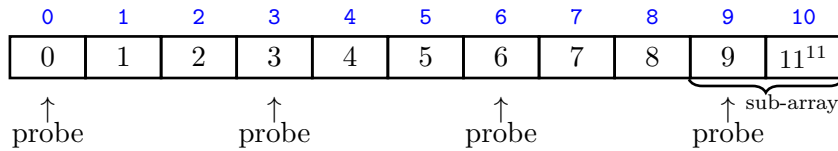
Input : Sorted array A of n integers, key k

```

1 if ( $k < A[0]$ ) then return “not found, would be left of index 0”
2 if ( $k > A[n-1]$ ) then return “not found, would be right of index  $n-1$ ”
3 if ( $k = A[n-1]$ ) then return “found at index  $n-1$ ”
4  $\ell \leftarrow 0, r \leftarrow n-1$ 
5 while  $r \geq \ell + 2$  do                                     // inv:  $A[\ell] \leq k < A[r]$ 
     $N \leftarrow r - \ell - 1, p \leftarrow \frac{k - A[\ell]}{A[r] - A[\ell]}, \mu \leftarrow p \cdot N, m \leftarrow \ell + \lceil \mu \rceil$ 
    if ( $A[m] \leq k$ ) then                                     // search rightward for a sub-array of size  $\approx \sqrt{n}$ 
        for  $h = 1, 2, \dots$  do
             $\ell \leftarrow m + (h-1)\lceil \sqrt{N} \rceil, r' \leftarrow m + h \cdot \lceil \sqrt{N} \rceil$ 
            if ( $r' \geq r$ ) then break out of for-loop          // out of bounds
            if ( $A[r'] > k$ ) then                               // a probe
                 $r \leftarrow r',$  break out of for-loop
        else ...                                             // proceed symmetrically going leftward
    //  $r \leq \ell + 1$  and  $A[\ell] \leq k < A[r]$ , so  $r = \ell + 1$  and  $k$  can only be  $A[\ell]$ 
14 if ( $k = A[\ell]$ ) then return “found at index  $\ell$ ”
15 else return “not found, would be between index  $\ell$  and  $\ell + 1$ ”

```

3792 too small. Since $N = n - 2 = 9$, we probe rightwards at every $\sqrt{N}^{\text{th}} = 3^{\text{rd}}$ index. The initial
3793 probes are all too small, and r' is out of bounds eventually, so we repeat in the sub-array from the
3794 last probe to r . The size has been reduced much more quickly, and the algorithm will terminate
3795 faster.



3796

3797 To discuss the run-time, we first need to re-consider our computer model. The pseudo-
3798 code uses $\lceil \sqrt{N} \rceil$ —can this be computed in constant time? This is really not clear. Note that
3799 $\sqrt{N} = 2^{\frac{1}{2} \log N}$, so one could use $\lceil \sqrt{N} \rceil \leq 2^{\frac{1}{2} \lceil \log N \rceil}$ as an approximation. (It can be shown that
3800 this is good enough for the run-time.) If we assume that we can compute both 2^x and $\lceil \log x \rceil$
3801 for an integer x in constant time (not unrealistic, since both are bit-shifts), then we hence can
3802 get a good-enough approximation of $\lceil \sqrt{N} \rceil$ in constant time.

3803 Hence the run-time in one round of the while-loop is proportional to how many probes there
3804 are. We always do at least one probe (when comparing k to $A[m]$), and also have a trivial upper
3805 bound on their number.

3806 **Observation 6.1.** *There are at most $1 + \lceil \sqrt{N} \rceil$ probes.*

Proof. Assume that we probe rightward, the other case is symmetric. We always do one probe at $A[m]$. For $i \geq 2$, the i^{th} probe happens for $h = i-1$ in the for-loop and hence is at index $m + (i-1)\lceil\sqrt{N}\rceil$, which for $i = \lceil\sqrt{N}\rceil + 1$ is at index $m + \lceil\sqrt{N}\rceil \cdot \lceil\sqrt{N}\rceil \geq \ell + N \geq r-1$. Since $\sqrt{N} \geq 1$, the probe after this would be at index r or bigger, hence be out of bounds. \square

Lemma 6.3. *Algorithm interpolation-search-improved has $\Theta(\sqrt{n})$ worst-case run-time.*

Proof. For the upper bound, the run-time satisfies the recursion $T(n) = T(n') + O(\sqrt{n})$, where $n' \leq \lceil\sqrt{N}\rceil + 1 \leq \sqrt{n} + 2$. We will permit ourselves to be a bit sloppy in the recursion here (see the proof of Lemma 1.5 for ideas on how to be more precise) and analyze the situation where $n' \leq \sqrt{n}$ and the time in each round is at most $c\sqrt{n}$ for some constant c . If we choose c so big that also $T(n) \leq c$ for $n < 16$, then we can prove by induction on n that $T(n) \leq 2c\sqrt{n}$.

This clearly holds for $n < 16$ since $\sqrt{n} \geq 1$. So consider the case where $n \geq 16$, hence $n' \leq \sqrt{n} \leq \frac{n}{4}$. Then using induction we have

$$T(n) \leq T(n') + c\sqrt{n} \leq 2c\sqrt{n'} + c\sqrt{n} \leq 2c\sqrt{\frac{n}{4}} + c\sqrt{n} = c\sqrt{n} + c\sqrt{n} = 2c\sqrt{n}.$$

The bound is tight in the above example where the outer-most round uses $\Omega(\sqrt{n})$ probes. \square

The really interesting result is that the average-case number of probes is a constant, where ‘average’ considers the model where all numbers were randomly and uniformly chosen.

Claim 6.2. *Assuming keys in the array are uniformly distributed, the average-case number of probes in one round is in $O(1)$.*

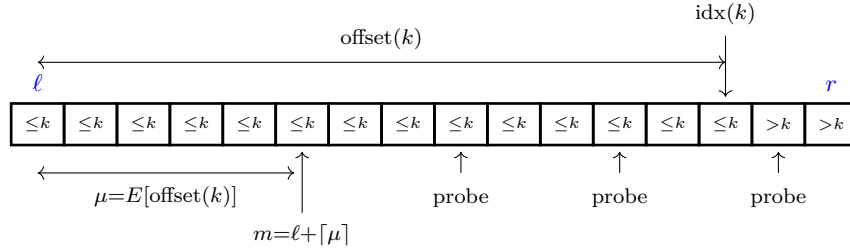
Proof. Since we consider the numbers as having been chosen randomly, we can define random variables that depend on the outcome of those choices. In particular, let $\text{idx}(k)$ be the random variable that denotes the largest index i such that $A[i] \leq k$. (In other words, either $A[i] = k$ or k is not in the array.) When we begin a round of the while-loop, we know that $A[\ell] \leq k < A[r]$, so $\ell \leq \text{idx}(k) \leq r-1$.

To be able to bound the number of probes, we need to study $\text{idx}(x)$ further, and in particular find its expected value. Actually, we will study the closely related random variable $\text{offset}(k) := \text{idx}(x) - \ell \in [0, \dots, r - \ell - 1] = [0, N]$. See Figure 6.3 and note that for any $0 \leq i \leq N$ we have

$$\begin{aligned} \text{offset}(k) = i &\Leftrightarrow \text{idx}(k) = \ell + i \Leftrightarrow A[\ell.. \ell + i] \leq k \text{ and } A[\ell + i + 1..r] > k \\ &\Leftrightarrow \text{exactly } i+1 \text{ items in } A[\ell..r] \text{ are no bigger than } k \\ &\Leftrightarrow \text{exactly } i \text{ items in } A[\ell+1..r-1] \text{ are no bigger than } k. \end{aligned}$$

Here the last equivalence holds because $A[\ell]$ is never bigger than k , and $A[r]$ is always bigger than k . So what $\text{offset}(k)$ hence really measures is how many of the $N = r - \ell - 1$ items in $A[\ell+1..r-1]$ happen to be no bigger than k . Any one such item x satisfies $A[\ell] \leq x \leq A[r]$ and is uniformly chosen within this interval. So

$$P(x \leq k) = \underbrace{\frac{A[\ell] \quad k \quad A[r]}{P(x \text{ in here})}} = \frac{k - A[\ell]}{A[r] - A[\ell]} = p.$$

Figure 6.3: The set-up for the proof of *interpolation-search-improved*.

Since the items in $A[\ell+1..r-1]$ are chosen independently, therefore

$$P(\text{offset}(k) = i) = P(\text{exactly } i \text{ items in } A[\ell+1..r-1] \text{ are no bigger than } k) = \binom{N}{i} p^i (1-p)^{N-i},$$

3830 because there are $\binom{N}{i}$ possible subsets of size i , and for each of them the probability that exactly
 3831 those are smaller than k is p^i , and the probability that all the others are not smaller than k is
 3832 $(1-p)^{N-i}$. To put it in probabilistic terms, $\text{offset}(k)$ follows a binomial distribution with respect
 3833 to parameters N and p . The following is known for binomial distributions:

$$\begin{aligned} \text{expected value} &= E[\text{offset}(k)] = N \cdot p = \mu \\ \text{variance } \sigma^2 &:= V[\text{offset}(k)] = N \cdot p(1-p) \leq \frac{1}{4}N \end{aligned}$$

3834 In particular, we have $E[\text{idx}(k)] = \ell + E[\text{offset}(k)] = \ell + \mu$, we would expect to find k at index
 3835 $\ell + \mu = \ell + N \cdot p$. (This tells us why we should use $N = r - \ell - 1$, rather than $r - \ell$, in the
 3836 formula for computing m .) Now m needs to be an integer, so should we round up or round
 3837 down? Consider the situation where k is not in the array. Then $\text{idx}(k)$ will actually be the
 3838 index of the predecessor of k , and the search-routine should return the interval to its right as
 3839 the place where k would have been. To find this, we should be rounding up. So this justifies the
 3840 formula for m in *interpolation-search-improved*.

Now we can bound the expected number of probes. Let us assume that we probed rightward (the other case is symmetric). We always do one probe at $A[m]$, and we nearly always to a second probe at $A[m + \lceil \sqrt{N} \rceil]$ (unless this is already out of bounds). For any integer i with $3 \leq i \leq \lceil \sqrt{N} \rceil + 1$, the i^{th} probe happens when $h = i - 1$ in the for-loop, hence at $A[m + (i - 1)\lceil \sqrt{N} \rceil]$

(unless this is out of bounds). Observe that we have

$$\begin{aligned}
P(\#\text{probes} \geq i) &= P(\#\text{probes} \geq i \mid (i-1)^{\text{st}} \text{ probe within bounds}) \cdot \underbrace{P((i-1)^{\text{st}} \text{ probe within bounds})}_{\leq 1} \\
&\leq P((i-1)^{\text{st}} \text{ probe did not give suitable interval}) \\
&= P\left(A[m + (i-2)\lceil\sqrt{N}\rceil] \leq k\right) \quad (\text{by definition of probing}) \\
&= P\left(\text{idx}(k) \geq m + (i-2)\lceil\sqrt{N}\rceil\right) \quad (\text{by definition of } \text{idx}(k)) \\
&\leq P\left(\text{idx}(k) \geq \ell + \mu + (i-2)\sqrt{N}\right) \quad (\text{lowering the bound increases probability}) \\
&= P\left(\text{offset}(k) - \mu \geq (i-2)\sqrt{N}\right) \\
&\leq P\left(|\text{offset}(k) - \mu| \geq (i-2)\sqrt{N}\right) \quad (\text{absolute values increase probability})
\end{aligned}$$

But recall that $\text{offset}(k)$ has expected value μ , so we are asking here for the probability that a random variable X is far away from its expected value μ . Chebyshev's inequality (Lemma A.3) states that $P(|X - \mu| \geq t) \leq \frac{\sigma^2}{t^2}$. Applying this with $t = (i-2)\sqrt{N}$, we get

$$P(\#\text{probes} \geq i) \leq P(|\text{offset}(k) - \mu| \geq (i-2)\sqrt{N}) \leq \frac{\sigma^2}{((i-2)\sqrt{N})^2} \leq \frac{N/4}{((i-2)\sqrt{N})^2} = \frac{1}{4(i-2)^2}.$$

3841 With this we can finally compute the average-case number of probes for *interpolation-search-improved*.
 3842 Since we modelled the input as having been randomly chosen, this is obtained by taking the
 3843 expected value of the number of probes under this choice.

$$\begin{aligned}
E[\#\text{probes}] &= \sum_{i \geq 0} i \cdot P(\#\text{probes} = i) = \sum_{i \geq 1} P(\#\text{probes} \geq i) \\
&\leq \sum_{i=1}^2 \underbrace{P(\#\text{probes} \geq i)}_{\leq 1} + \sum_{i=3}^{\lceil\sqrt{N}\rceil+1} \underbrace{P(\#\text{probes} \geq i)}_{\leq \frac{1}{4(i-2)^2}} + \sum_{i > \lceil\sqrt{N}\rceil+1} \underbrace{P(\#\text{probes} \geq i)}_{=0 \leq \frac{1}{4(i-2)^2}} \\
&\leq 2 + \sum_{i \geq 3} \frac{1}{4(i-2)^2} = 2 + \frac{1}{4} \sum_{j \geq 1} \frac{1}{j^2} = 2 + \frac{1}{4} \cdot \frac{\pi^2}{6} \in O(1).
\end{aligned}$$

3844

□

3845 Now we need to argue that few probes translate to small run-time.

3846 **Lemma 6.4.** *Assuming keys in the array are uniformly distributed, the average-case run-time*
 3847 *of interpolation-search-improved is in $O(\log \log n)$.*

Proof. The average-case run-time satisfies $T(n) \leq T(n') + O(1)$ (for some $n' \leq \sqrt{n} + 2$) since the average-case number of probes is constant. We will again do a slightly sloppy analysis where we assume that $n' \leq \sqrt{n}$. We will also ignore the case $n = 1, 2$ (where $\log \log n$ is not even defined and 0, respectively); this can be handled by defining separate upper bounds as in Lemma 1.5.

So let c be constant such that $T(n) \leq T(n') + c$ for some $n' \leq \sqrt{n}$ and also $T(n) \leq c$ for $n = 3$. Define $L = \lceil \log \log n \rceil$, which is at least 1 for $n \geq 3$. We will show by induction that $T(n) \leq cL$, which clearly holds for $n = 3$. For $n \geq 4$, we have $L \geq 2$ and

$$\log \log \sqrt{n} \leq \log \log \sqrt{2^{2^L}} = \log \log (2^{\frac{1}{2} \cdot 2^L}) = \log \log (2^{2^{L-1}}) = \log (2^{L-1}) = L-1 = \lceil \log \log n \rceil - 1.$$

By integrality therefore $\lceil \log \log \sqrt{n} \rceil \leq \lceil \log \log n \rceil - 1$ and $T(n) \leq T(n') + c \leq \lceil \log \log \sqrt{n} \rceil + c \leq c(L-1) + c = cL$ as desired. \square

Actual number of comparisons. It is insightful to inspect the bounds that we proved a bit more carefully—is $\Theta(\log \log n)$ really better than $\Theta(\log n)$? When functions are as slow-growing as $\log \log n$, the constant factor hiding behind the asymptotic notations can make a difference in the sense that the asymptotically faster run-time is actually faster only for such gigantic values of n that one would not actually want to use the algorithm.

Specifically, we know that the optimized version of binary search uses at most $\lceil \log(2n+1) \rceil \approx \log n + 1$ comparisons. In contrast, *interpolation-search* uses 5 comparisons in each round (presuming it does not exit). We did not analyze the precise number of rounds in the average-case, but it is at least $\log \log n$. So a rough estimate is that *interpolation-search* uses at least $5 \log \log n$ comparisons on average. Now

$$5 \log \log n \leq \log n + 1 \quad \Leftrightarrow \quad n \geq 2,013,740.$$

While dictionaries with 2+ million entries are not unheard of, they are not happening all the time either. So for day-to-day use, *interpolation-search* may actually be no better than *binary-search-optimized*.

The situation is quite different for *interpolation-search-improved*. First, the code was written carefully so that we only do *one* comparison per probe. Secondly, we upper-bounded the average-case number of probes by 2.5. Since 2 comparisons should be enough if $n = 3$, therefore $c \leq 2.5$ in the proof of Lemma 6.4, so the average-case number of comparisons is $2.5 \log \log n$. That does not look all that much better, but observe that

$$2.5 \log \log n \leq \log n + 1 \quad \Leftrightarrow \quad n \geq 2^4 = 16,$$

and dictionaries with 16 or more entries are very common. Even accounting for the fact that one analysis uses the average case while the other uses the worst-case, one would expect that *interpolation-search-improved* should work noticeably faster in real-life applications.

6.2 Dictionary for Words

We now shift to a totally different kind of keys: words. These violate many of our assumption on keys (they do not fit into one memory cell, and comparing them does not take constant time), but they have obvious applications when storing real-life dictionaries or anything else indexed by strings, such as DNA-sequences.

6.2.1 Preliminaries

Let us clarify a few definitions and notation first. A *word* (also called *string*) is an ordered sequence w of *characters* that have been taken from some *alphabet* Σ . The *length* of a word w , denoted $|w|$, is the number of characters that it contains. Crucially, there is no limit on the length of a word. We use $w[i]$ to refer to the i th character of a word (starting at index 0), i.e., we treat words as if they were stored in an array of characters. The set of all possible words with characters from alphabet Σ is denoted by Σ^* .

The alphabet Σ must be a finite set of characters, and typically the number of characters is quite small. (On the other hand, we assume throughout that $|\Sigma| \geq 2$, because the data structures and algorithms that we will see for words are generally trivial and meaningless if the alphabet has only one character.) The most commonly used alphabets are:

- $\Sigma = \{0, 1\}$, so examples of words are 001, 11010, 11.

In this case a character is also called a *bit* and a word over this alphabet is called a *bit-string*. Conceptually a bit-string is very similar to a number in base 2; the difference is that when considering numbers as a sequence of bits (as we did for example with radix sort) the number of bits is constant, while for bit-strings there is no limit on the length. Also, bit-strings may have leading 0s. Bit-strings are useful for storing numbers that are too large to fit into one memory cell.

- $\Sigma = \text{ASCII}$, so examples of words are `hello`, `English`, `fun!`.

You should have seen ASCII before, but as a quick reminder, it consists of the characters and digits used in English, plus some punctuation, symbols and control characters. In total there are $128 = 2^7$ characters, corresponding to length-7 bit-strings. The table below shows a few characters; note that `A` is mapped to $65 = (1000001)_2$, so the English characters are simply enumerated in base-2 except that there is a leading 1. (You are not expected to memorize ASCII; we will give you relevant pieces when needed.) Whenever we speak of ‘English text’, we mean words over alphabet ASCII that would be used in typical text in the English language.

char	null	start of heading	start of text	end of text	...	0	1	...	A	B	...	~	delete
code	0	1	2	3	...	48	49	...	65	66	...	126	127

ASCII is woefully under-equipped to deal with foreign languages that have diacritical marks, not to mention languages that are not based on the roman alphabet at all. There is an extension to ASCII called ISO-8859 which uses 8 bits (hence has 256 characters) and

handles most Western languages. Most results stated here for English text should transfer to other languages and alphabets where the number of distinct characters is non-trivial (say at least 20) but not huge (say below 300).

- $\Sigma = \{A, C, G, T\}$, so examples of words are **AGAACT** or **CGCTTATC**.

These words occur naturally in bioinformatics; the alphabet is used to encode DNA (and such words are hence called *DNA-strings*). Closely related is also the alphabet that consists of 61 of the 64 possible 3-letter combinations of $\{A, C, G, U\}$; this is used to represent the output of DNA transcription.

It is often convenient to have a special symbol that indicates ‘end-of-the-word’, and which occurs as the last character of the word and nowhere else. In ASCII-strings stored in computers, this character is automatically appended in most programming languages (it is control character ETX with ASCII-number 03). In DNA transcripts, it is expressed via one of the three stop-codons (the missing three triplets from $\{A, C, G, U\}$). In our examples below, we will use ‘\$’ for the end-of-word symbol, and assume throughout that this is not in the alphabet (hence not used anywhere else in the word). The end-of-word symbol is *not* counted towards the length of the word.

Finally we need to talk about how to *compare* two words. We use the *lexicographic order*, which is the order in which words would be listed in a dictionary. For example

$$\text{be\$} <_{lex} \text{bear\$} <_{lex} \text{beer\$}.$$

Formally, lexicographic order is defined as follows:

- Find the leftmost character where w_1 and w_2 are different.
If we assume that words have end-of-word characters, then there will be such a character unless $w_1 = w_2$.
- If at this character word w_1 has the smaller character, then $w_1 < w_2$, otherwise $w_2 < w_1$. We assume here silently that characters can be compared. This is true for bit-strings (we have $0 < 1$). It is also true for ASCII, which is naturally mapped to numbers, and these numbers have been assigned such that the usual rules for English language are satisfied:

$$0 < 1 < \dots < 9 < A < B < \dots < Z < a < b < \dots < z.$$

Symbols are smaller than letters (and usually smaller than numbers), but you are not expected to learn their order; it will be specified as needed.

- The end-of-word character \$ is *always* smaller than all other characters in the alphabet. With that, a *prefix* of a word w (i.e., a word that consists of the first few characters of w) is *always* considered smaller, because the leftmost character where the two words are different is the end-of-word character of the prefix, and this character is smaller than whichever character is at w .

We use $\text{strcmp}(w_1, w_2)$ to denote the process of comparing two words w_1 and w_2 ; it returns a value in $\{-1, 0, +1\}$ depending on whether w_1 is smaller, equal or bigger than w_2 . See Algorithm 6.5. We write it here in a slightly more general way which also permits (for w_1) to

3932 compare a substring of w_1 , i.e., a word $w_1[i..i']$ for some $i \leq i'$. We even allow index i' to be
 3933 beyond the end of w_1 ; this will make some pseudo-code in Chapter 9 simpler.

Algorithm 6.5: $strcmp(w_1, w_2, i \leftarrow 0, i' \leftarrow w_1.length-1)$

Input : Word $w_1[0..n-1]$, word $w_2[0..m-1]$, indices $0 \leq i \leq i'$ where $i \leq n-1$
Output: Compares $w_1[i.. \min\{i', n-1\}]$ to w_2

```

1 int  $k \leftarrow \min\{m, \min\{i', n-1\} - i + 1\}$            // number of chars to compare
2 for  $j \leftarrow 0$  to  $k - 1$  do
3   if  $w_1[i + j]$  is before  $w_2[j]$  in  $\Sigma$  then return  $-1$ 
4   if  $w_1[i + j]$  is after  $w_2[j]$  in  $\Sigma$  then return  $+1$ 
   // inv:  $w_1[i..i+k-1] = w_2[0..k-1]$ 
5 if  $k < m$  then return  $+1$                                // Substring of  $w_1$  was prefix
6 if  $k < \min\{n-1, i'\} - i + 1$  then return  $-1$            //  $w_2$  was prefix
7 return 0
```

3934 Note that *strcmp* is *not* a constant-time operation; we may be lucky and determine the
 3935 outcome when comparing the leftmost character of the words, but if we are unlucky (e.g. when
 3936 the words are equal or one is a prefix of the other) then we must compare all characters until
 3937 one of the words runs out of characters. Therefore the worst-case run-time of $strcmp(w_1, w_2)$ is
 3938 $\Theta(\min\{|w_1|, |w_2|\})$. (One can argue that the average-case run-time of *strcmp* is in $\Theta(1)$; this is
 3939 left as an exercise.)

3940 6.2.2 Binary tries

3941 We now turn to the most natural method of storing words, which also turns out to be quite
 3942 efficient. We will explain it first for bit-strings, and then generalize to arbitrary words later.
 3943 The data structure is called a *binary trie*. The word ‘trie’ is pronounced to rhyme with “try”,
 3944 and was chosen because it occurs within **re**trieval: it is a data structure that permits efficient
 3945 retrieval of data associated with words.

3946 We make the following assumptions about the words D that we want to store:

- 3947 • The elements of D are bit-strings, i.e., $\Sigma = \{0, 1\}$.
- 3948 • No word in D is a prefix of another word in D .

3949 This assumption is true if the words that we store have end-of-word characters attached,
 3950 or if all words have the same length. We will discuss shortly how to handle tries where
 3951 this assumption is violated.

3952 We also need the concept of a *ternary tree*, which is a rooted tree where every node has exactly
 3953 *three* (possibly empty) subtrees.

3954 **Definition 6.1.** A binary trie (also called radix tree or standard trie) is a rooted ternary tree
 3955 T where every link is labeled with 0, 1 or \$. A word x is stored in trie T if there exists a leaf ℓ
 3956 such that the labels on the path from the root to ℓ equal the characters of word x .

It may seem confusing that we call a *ternary* tree a *binary* trie, but the reason is that the end-of-word symbol \$ does not really count. We will soon discuss variants where we omit the end-of-word symbol, and then the trie is indeed a binary tree. The name *radix trie* stems from the similarity of tries to radix sort: both rely on individual bits of the key to structure the search.

See Figure 6.4 for an example of a binary trie. As usual, in our figures we only show the stored keys (which are words in this case), and do not show the associated values.

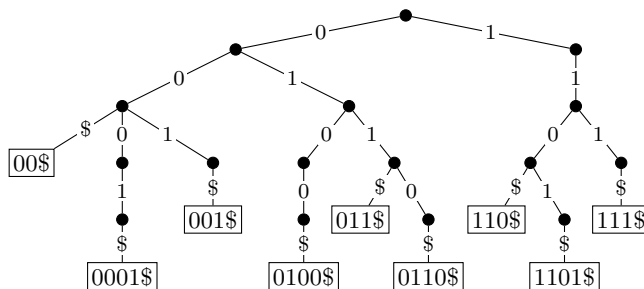


Figure 6.4: A trie that stores $\{00\$, 0001\$, 001\$, 0100\$, 0110\$, 0111\$, 110\$, 1101\$, 111\ \$\}$.

Note that a trie resembles a decision tree! We do not store data at the internal nodes, instead they are comparisons/questions that will help us find the data at the leaves. (For a trie, at an interior node the question is “what is the d th bit of the string?”, where d is the depth of the node.)

Operations. It is very easy to search whether a trie stores a given word x . We know all the bits of x , and we know the unique place where it is stored in the trie, if at all. So we simply follow the path down from the root, always taking the link that corresponds to the current bit of x , until we either find x at a leaf or until there is no suitable link (then x was not in the trie). See Algorithm 6.6 for the code and Figure 6.5 for an example. (This code *requires* that words end with \$—can you see why?) The run-time is $\Theta(|x|)$.

Insertion in a trie is very easy. First, search for the word x that we wish to insert. This will return that node v that is missing the suitable link where x would have been stored. So insert a child of v labeled with the corresponding bit of x , and then expand the tree downward with one further node for each bit of x that was not parsed yet. The run-time for inserting in a trie is again $\Theta(|x|)$. Figure 6.6 shows the trie that results from inserting 0111\$ in the binary trie of Figure 6.5; note that we searched for this word in the trie earlier and hence already know the node v .

Deletion is also very easy in a trie. First search for the word x that we wish to delete. Once found at leaf v , go back up from v and delete the nodes until you reach an ancestor w of v that has at least two children. We keep ancestor w , because it stores some other words in its sub-trie. Figure 6.7 illustrates an example. The run-time for deletion is again $\Theta(|x|)$.

Algorithm 6.6: *trie::search*($v \leftarrow \text{root}, d \leftarrow 0, x$)

Input : node v of the trie; level d of v , word x stored as array of chars
// pre: The links on the path to v are labelled with $x[0], \dots, x[d-1]$

```

1 if  $v$  is a leaf then
2   | return  $v$ 
3 else
4   | let  $v'$  be the child of  $v$  for which the link from  $v$  is labeled with  $x[d]$ 
5   | if there is no such child then
6   |   | return “not found, but would be a descendant of  $v$ ”
7   | else
8   |   | trie::search( $v', d + 1, x$ )

```

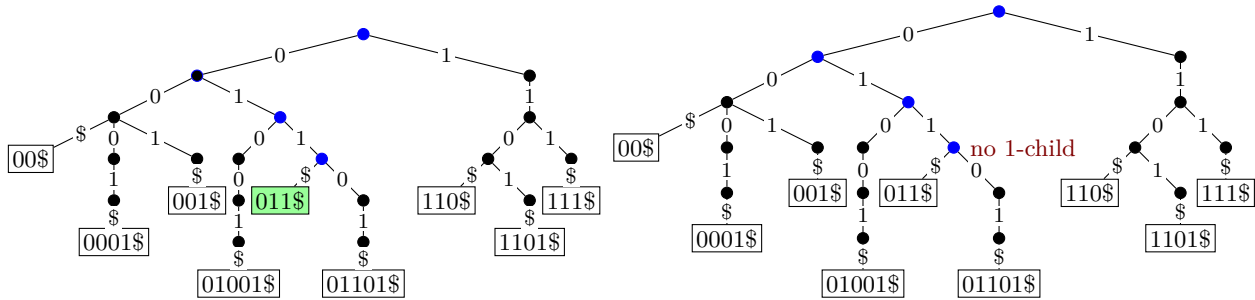


Figure 6.5: Example of searching for 011\$ and 0111\$ in a binary trie.

Analysis. For all three operations, the run-time is $O(|x|)$, because we handle $|x|+2$ nodes² and each node has a constant number of children, so we can find the appropriate child in constant time. Note in particular that this run-time is *independent* of the number n of words that has been stored. It also has the advantage that short words are found more quickly, and short words are usually more common (at least in human languages).

How much space does a trie use? This will obviously depend on the number n of stored words, but also crucially on the length of each word. For each word x , there are $|x|+2$ nodes on the path from the root to the leaf that stores x . So we know that the total number of nodes for all words is $O(\sum_{\text{words } x} |x|)$. This bound is tight, for example if words are very long and only the first $\log n$ characters are shared by some words. But most of the time, many words will share prefixes, and so we would expect the space to be much less than this upper bound.

²The ‘+2’ is needed because the root does not correspond to a character of x , and we do not count the end-of-word sign \$ for the length of x . However, in what follows we will be somewhat sloppy about such small additive constants, and in particular write $O(|x|)$ for terms that are really $O(1 + |x|)$. Since all except at most one word have $|x| \geq 1$, this makes no difference except for pathological cases.

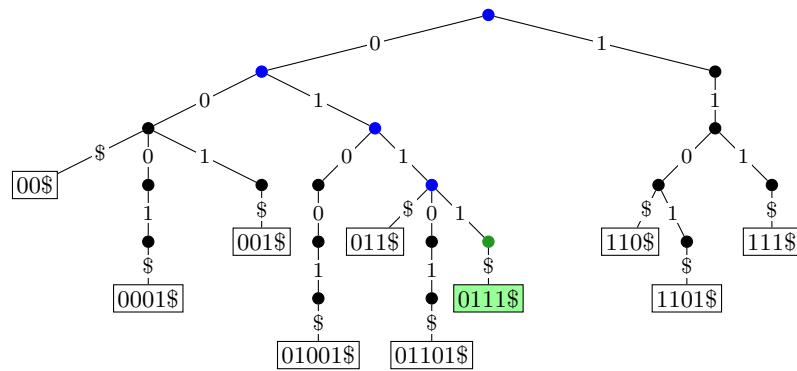


Figure 6.6: The result of inserting 0111\$.

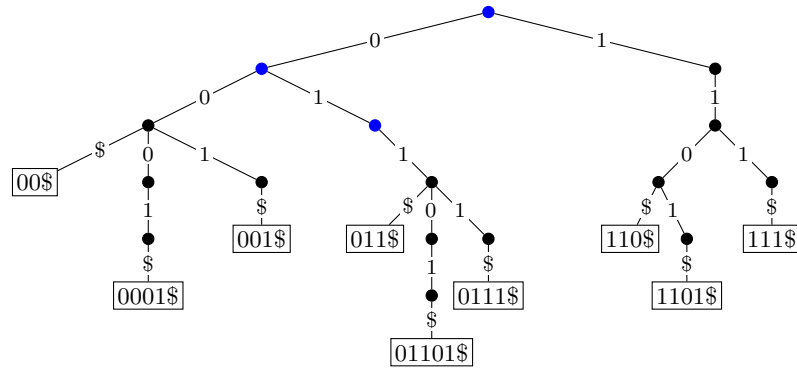


Figure 6.7: The result of deleting word 01001\$.

3996 **Prefix-search.** In later applications of tries, we will need a generalization of *search* that does
 3997 the following:

3998 *prefixSearch*(x): given a word x , is it a prefix of a word w stored in the dictionary?

3999 This is extremely easy to do in a trie. The only change that happens is that we act differently
 4000 when we reach the end-of-word character of w . For *search*, we required that the trie then has
 4001 a child labelled \$. For *prefixSearch*, there is no such requirement. Algorithm 6.7 gives the
 4002 pseudo-code.

4003 In case of success, the algorithm as given only returns the node v such that some word stored
 4004 in the subtree at v can serve as w (i.e., is a word for which x is a prefix). In some situations
 4005 we may want to actually *find* word w . If this is desired, then the trie additionally should store
 4006 *leaf-references*, which means that each interior node v stores a reference to some leaf ℓ in its
 4007 subtree. (These leaf-references must then be updated during insertion and deletion; details are

Algorithm 6.7: *trie::prefixSearch*($v \leftarrow \text{root}, d \leftarrow 0, x$)

Input : node v of the trie; level d of v , word x stored as array of chars
// pre: The links on the path to v are labelled with $x[0], \dots, x[d-1]$

```

1 if  $v$  is a leaf or  $d = x.\text{length}$  then
2   | return  $v$ 
3 else
4   | let  $v'$  be the child of  $v$  for which the link from  $v$  is labeled with  $x[d]$ 
5   | if there is no such child then
6   |   | return "not found"
7   | else
8   |   | trie::prefixSearch( $v', d + 1, x$ )

```

4008 left as an exercise.) With this, we can perform a prefix-search (and return the appropriate word
 4009 w) in time $O(|x|)$.

4010 **6.2.3 Variations of binary tries**

4011 The version of the binary trie presented above is probably the easiest to understand and the
 4012 easiest to code, but it is not the most efficient in practice. This is partly due to our choice
 4013 of pseudo-code (for example one should really use a while-loop, rather than recursion, in Al-
 4014 gorithm 6.6)), but binary tries are also very wasteful in space. We now briefly discuss three
 4015 variations that use less space.

4016 **Variation 1: No leaf labels:** In the binary trie we described earlier, each word was effectively
 4017 stored twice: Once explicitly at the leaf, and once implicitly through the sequence of characters
 4018 on the links that lead to the leaf. This is inefficient, and we could have saved ourselves the copy
 4019 of the word stored at the leaf. (We would still store the associated value at the leaf, but not the
 4020 key, i.e., the word.) See Figure 6.8.

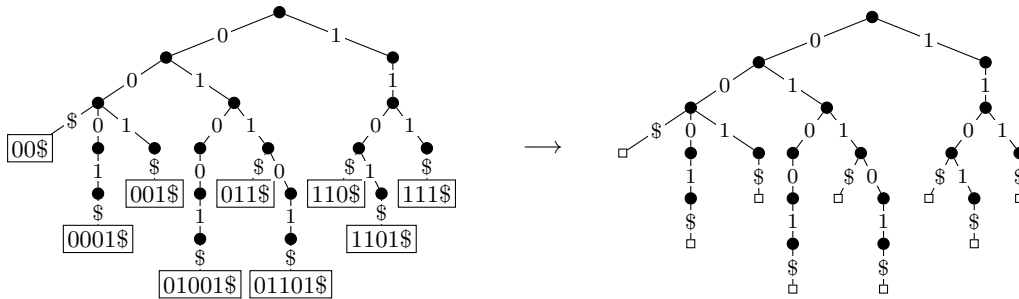


Figure 6.8: Variation 1: Do not store words at the leaves.

Variation 2: Allow proper prefixes: We previously assumed the words stored in the trie are prefix-free. This was needed so that words would end up at the leaves. But if we permit to store words at interior vertices, then we can accommodate words that are prefixes of another word. This then also eliminates the requirement of an end-of-word symbol, making the trie a binary tree. (On the other hand, it complicates the code.)

Combining this with the idea of the previous variation, we do not need to store the words at all. All we need to store is a *flag* with each node of the trie that indicates whether this node stores a word or not. (If the words stand for a larger key-value pair, then the flag would simply be the reference to an associated value, which would be NIL if there is no word stored at the node.)

Since we can dispose of the end-of-word symbol in this model, we can do even more to reduce the space-requirement. Recall that so far we assumed that each link stores a character of the alphabet. But if we do not have end-of-word symbols, then every node has only two subtrees, and we can agree on a convention that the left subtree represents bit 0 while the right character represents bit 1. With this, no information needs to be stored with each link, and the trie becomes a binary tree where the nodes store flags (or references to values). See Figure 6.9.

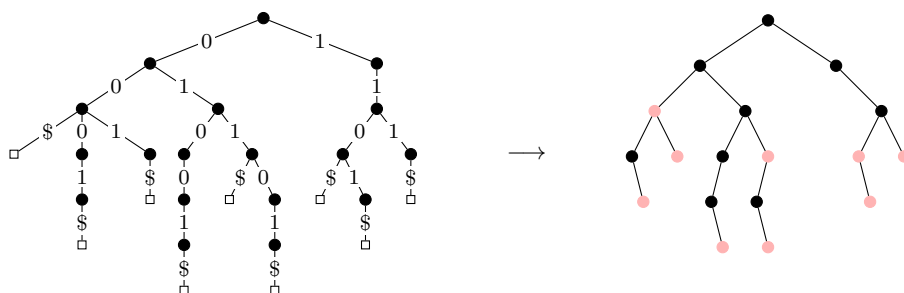
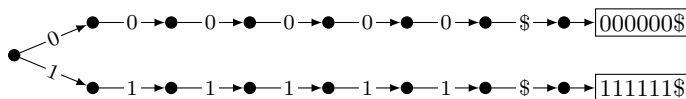


Figure 6.9: Variation 2: Store words via flags at nodes and omit end-of-word symbols.

Variation 3: Pruned tries: Consider the following binary trie (drawn sideways for ease of reading):



This is hugely wasteful—once we have parsed the first bit, the two words have been distinguished and we need not have stored anything more than the words itself. This is the idea of a *pruned trie*: stop expanding the trie if this is not needed for distinguishing words. Put differently, a node in the trie has a child only if it has at least two descendants. Figure 6.10 shows an example.

Note that in a pruned trie, we *must* store the full word at every leaf. To see why, consider searching for 000\$ and for 0001\$ in the pruned trie of Figure 6.10. Both searches bring us to the exact same leaf, but only one of the two words is actually stored in the trie. So the leaf must

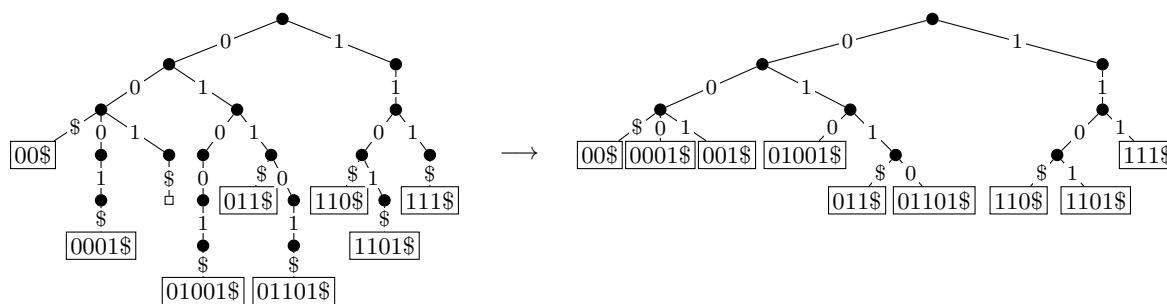


Figure 6.10: Variation 3: Pruned trie.

store sufficient information to be able to tell whether a word is in the dictionary or not. So one might object that a pruned trie does not really save space (compared to Variant 1), because the nodes that we save are made up for by the space that is needed in the leaves. However, in practice storing a bit-string takes relatively little space (if implemented well we can put 32 or 64 bits into one memory cell) while a node in a trie takes quite a bit of space (we need at the least the references to the children, which take one memory cell each). As such, pruned trie tend to be the most efficient variant of tries in practice.

More on pruned tries: (cs240e) There is a strong connection between pruned tries and MSD radix-sort for base-2 numbers. Recall (see also Figure 6.11) that MSD radix-sort first splits by the leading bit, then recurses in the resulting groups. This is the equivalent of the left and right subtree of the pruned trie. It then recurses on the next bit and the next bit, until there is one number left in each group. This corresponds to the pruned trie that splits until there is one word left in each subtree. Thus, the recursion tree of MSD radix-sort is exactly the same as the pruned trie of the bit-strings. Vice versa, we can sort words by inserting them into a pruned trie and reading the resulting leaves in order. This gives words in lexicographic orders (which is the same as the numeric order if all bit-strings have the same length.)

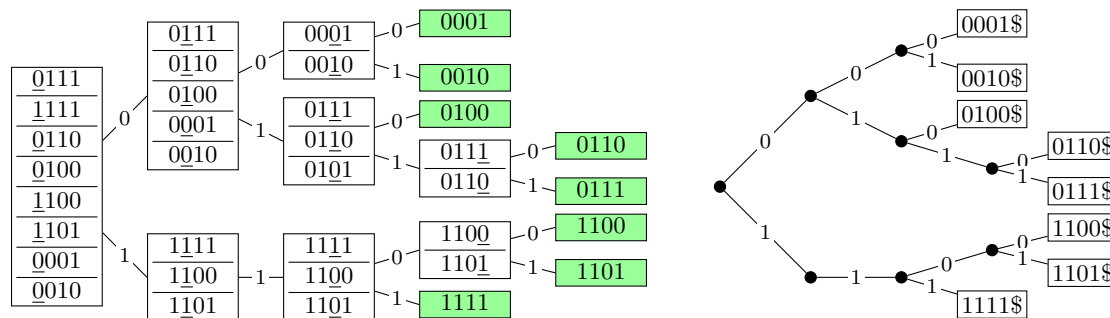


Figure 6.11: Tracing how MSD radix sort behaves gives a pruned trie (drawn sideways).

Another application of pruned tries is to store numbers with infinite precision. It is of course impossible to store infinite precision in finite computer memory, but using pruned tries we can store numbers as “precise enough”, i.e., with exactly as much precision as is needed to distinguish them. To do so, express the numbers as infinite-length bit-strings and store them in a pruned trie. See Figure 6.12

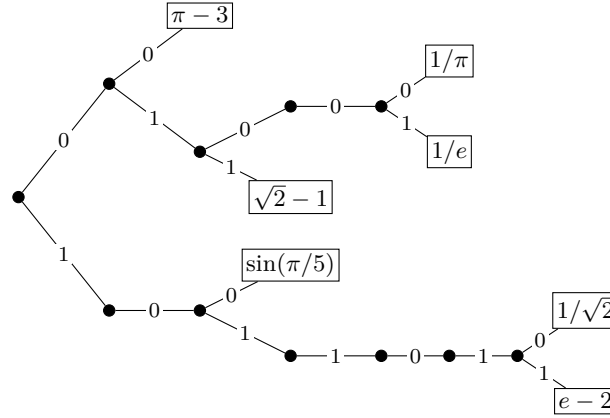


Figure 6.12: Storing some irrational numbers in $[0, 1)$ in a pruned trie. For example, $e - 2 = (0.71828182845\dots)_{10} = (.10110111111000010101\dots)_2$, but only the first 7 bits are needed to distinguish it from the other numbers.

When thinking about storing infinite-length bit-strings in a trie, you should worry about the run-time of the operations (the previous bound of $O(|x|)$ is no longer helpful). Certainly the worst-case run-time could be very large, but the average-case run-time is small, presuming that the strings are uniformly distributed. (Recall that this means that we view them as having been chosen randomly from the universe of all strings.) We only show this for *search* below.

Lemma 6.5. *If the (infinite-length) bit-strings stored in a prune trie were uniformly distributed, then the average-case search time is in $O(\log n)$.*

Proof. Let B_1, \dots, B_n be the stored numbers (i.e., infinite-length bit-strings) and let B be the (infinite) bit-string that we are searching for. Since B_1, \dots, B_n were uniformly distributed, each bit of each B_i was randomly chosen to be 0 or 1 with equal probability. We can therefore define a random variable I_i (for $i = 0, 1, \dots$) that is an indicator-variable with

$$I_i = \begin{cases} 1 & \text{if the } i\text{th bit of } B \text{ was looked at during search} \\ 0 & \text{otherwise} \end{cases}$$

Clearly then the run-time to search for B is proportional to $\sum_{i \geq 0} I_i$. We have $I_0 = 1$ (we always compare the leftmost bit of B at the root) and $I_i \leq 1$ for all i , but we now develop a different

bound. Namely,

$$I_i \leq \sum_{k=1}^n I_{i,k} \quad \text{where } I_{i,k} = \begin{cases} 1 & \text{if } B[i] \text{ is compared with } B_k[i] \text{ during the search} \\ 0 & \text{otherwise} \end{cases}$$

To compute $E[I_{i,k}]$, we must determine the probability that $B[i]$ gets compared with $B_k[i]$. This happens only if $B[0..i-1] = B_k[0..i-1]$, i.e., the two strings agree on the first i bits. But each bit of B_k was randomly chosen, which means that its leftmost bit equals $B[0]$ with probability $\frac{1}{2}$, its next bit equals $B[1]$ with probability $\frac{1}{2}$, and so on. Therefore

$$E[I_{i,k}] = P(I_{i,k} = 1) = P(B[0..i-1] = B_k[0..i-1]) = \frac{1}{2^i} \quad \text{and} \quad E[I_i] \leq \sum_{k=1}^n E[I_{i,k}] \leq \frac{n}{2^i}.$$

4076 So we now know $E[I_i] \leq \min\{1, \frac{n}{2^i}\}$ and we want to compute $\sum_{i \geq 0} E[I_i]$. This is *exactly* the
 4077 same problem as we had when computing the height of the skip list (Lemma 5.1), and with
 4078 verbatim the same proof one shows that the sum is in $O(\log n)$. \square

4079 6.2.4 Compressed tries

4080 Pruned tries save some space, compared to standard binary tries, by pruning off paths at the
 4081 bottom of the trie that are not needed to distinguish words. We now take this concept one
 4082 step further and also compress paths in the middle of the trie whenever they are not needed to
 4083 distinguish words. The result is a *compressed trie* (also known as *Patricia trie*, where ‘Patricia’
 4084 stands ‘Practical Algorithm to Retrieve Information Coded in Alphanumeric’). A compressed
 4085 trie satisfies the following:

- 4086 • Like a binary trie, a compressed trie is a ternary tree where every link is labeled with 0,
 4087 1 or \$, and every leaf stores a word.
- 4088 • Different from binary tries, every interior node stores an index.
- 4089 • If an interior node v stores index d , then any word w stored at a leaf below v has at least
 4090 $d + 1$ bits and all such words have the same first d bits.
- 4091 • If an interior node v stores index d , and the link to one child v' is labeled with c , then all
 4092 words w in the subtree at v' satisfy $w[d] = c$.

4093 A different (and possibly simpler) way to think of a compressed trie is to start with a binary trie,
 4094 label every interior node with its level, and then remove any node that has only one child, by
 4095 making that child the child of the parent. See Figure 6.13 for a binary trie and the corresponding
 4096 compressed trie.

4097 The main advantage of compressed tries is that we are potentially saving quite a few nodes.
 4098 In particular, every interior node has at least two children. By induction one easily shows that
 4099 a tree with this property and n leaves has at most $n - 1$ interior nodes. Therefore:

4100 **Observation 6.2.** *If a compressed trie stores n words then it has at most $2n - 1$ nodes.*

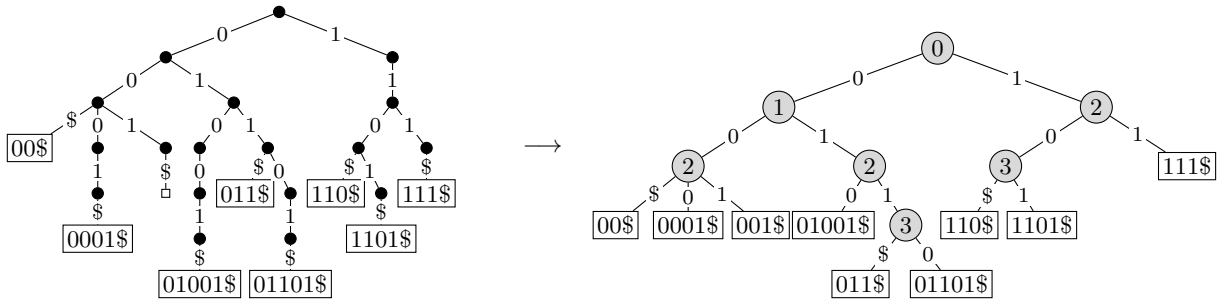


Figure 6.13: An example of a compressed trie.

It is also worth noting that compressed tries, just like binary tries, are unique.

One disadvantage is that a compressed trie is much less intuitive for a human to read. It is very easy to overlook that indices at nodes can “skip” a level and to misunderstand which word should be where. This will make the operations a bit harder to implement, see below.

Search: Searching in compressed trie is conceptually similar to searching in a binary trie (“go down in the trie to the only place where the word could possibly be”), but has two major changes. First, in a binary trie an interior node on level i splits by $w[i]$, so there was no need to store what bit to compare. In contrast to this, in a compressed trie we must use the stored index to determine which bit to compare. Second, in a binary trie *all* characters of the word were compared when going down to the leaf, while in a compressed trie only some characters are compared. To be sure that we found the correct leaf, we hence must do a full string-compare once we have reached the leaf. Algorithm 6.8 and Figure 6.14 give the pseudocode and an example.

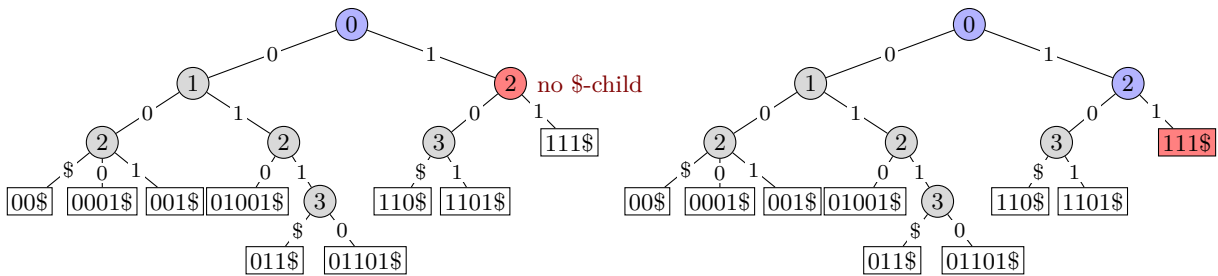


Figure 6.14: Searching in a compressed trie reports ‘not found’ for two different reasons: $search(10\$)$ stops when there is no suitable child, while $search(101\$)$ stops when we find at the leaf that the stored word does not match.

Insertion, deletion, prefixSearch: Both *insert* and *delete* are conceptually easy in compressed tries, but the details are a bit complicated. The idea is that we ‘uncompress’ the path

Algorithm 6.8: *compressedTrie::search*($v \leftarrow \text{root}, x$)

Input : Node v of the trie; word x to search for

```

1 if  $v$  is a leaf then
2   | return strcmp( $x, v.\text{key}$ )
3 else
4   |  $d \leftarrow$  index stored at  $v$ 
5   | if  $x$  has at most  $d$  bits then
6   |   | return “not found”
7   | else
8   |   | let  $v'$  be the child of  $v$  for which the link from  $v$  is labeled with  $x[d]$ 
9   |   | if there is no such child then
10  |   |   | return “not found”
11  |   | else
12  |   |   | compressedTrie::search( $v', x$ )

```

4116 towards where the word x that we want to insert/delete is stored. We then add/remove word
4117 x , and then compress the trie again (i.e., remove all nodes that now have exactly one child).
4118 The details are complicated, because in order to uncompress, we must find out exactly what
4119 the characters were on the internal nodes that had been compressed. To do so, we again store
4120 *leaf-references*, i.e., each internal node v stores a reference to some leaf ℓ in its subtree. It
4121 does not matter which leaf ℓ is picked for the leaf-reference—we promised in our definition of a
4122 compressed trie that all words that are stored at these leaves have the same missing characters.
4123 And then we need to update the leaf-references during insertion and deletion.

4124 We will not give the pseudo-code of this, but only note that despite these difficulties, insertion
4125 and deletion of word x can still be done in $O(|x|)$ time. Likewise, *prefixSearch* can be done in
4126 $O(|x|)$ time.

4127 **6.2.5 Multi-way tries**

4128 So far, we assume that the alphabet used by words is $\Sigma = \{0, 1\}$. But both the binary trie
4129 and its compressed variant generalize quite naturally to bigger alphabets Σ . Every node now
4130 has up to $|\Sigma| + 1$ children (one for every possible character in the alphabet Σ , and one for the
4131 end-of-word character \$). Figure 6.15 illustrates a such a *multi-way trie* and its compressed
4132 version.

4133 The operations for multi-way tries are implemented verbatim as they were for binary tries.
4134 The only change concerns finding the appropriate child. Specifically, we have a node v and a
4135 character c , and we want to find the child v' such that the link from v to v' is labeled with
4136 character c (or report that no such child exists). How quickly can we do this? There are up
4137 to $|\Sigma| + 1$ children, so if $|\Sigma|$ is big, then this would make a difference in terms of run-time and

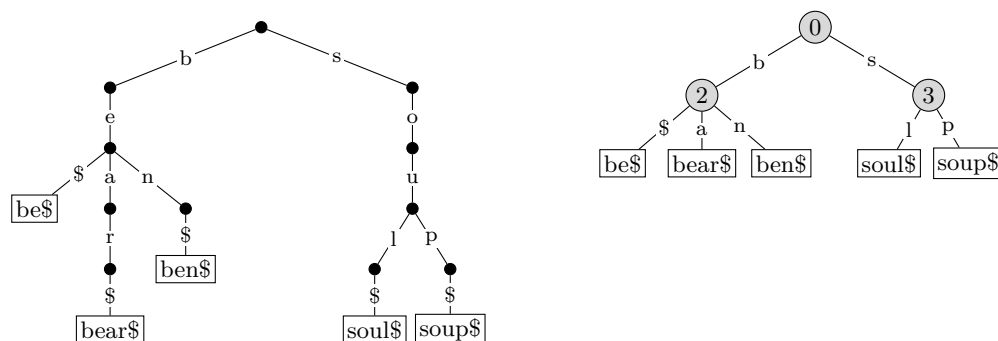


Figure 6.15: A multi-way trie and its compressed version holding strings {bear\$, ben\$, be\$, soul\$, soup\$}.

space-requirements. We have multiple options here:

- The simplest idea is that every node stores an array $child[0 \dots |\Sigma|]$. Presuming that we can map $\{\$ \} \cup \Sigma$ naturally to integers in $\{0, \dots, |\Sigma|\}$, then $child[i]$ stores the child corresponding to character i . This makes finding the appropriate child a constant-time operation, but it wastes space because every node uses an array of size $|\Sigma| + 1$, even if it only has one or two children.
- A better idea for saving space is to store for each node a list of all children that it actually has. Then the space used is the minimum possible (and can be seen as overhead for the child), so the total space of the trie is again $O(\sum_w |w|)$. But this has the disadvantage that we must search for the appropriate child in the list, which means scanning the entire list in the worst case.
- This second idea is just a special case of the idea of storing the children as a *dictionary*! Namely, each child is the value, and the keys are the characters on the links to the children. (Yes, we are using a dictionary to store a dictionary.) So we can really use any kind of dictionary implementation that we have seen. An AVL-tree would guarantee that we can find the child in $O(\log |\Sigma|)$ time. In practice an unsorted list with MTF heuristic would likely perform well (some characters are far more likely to be used for words than others). Even better is to use *hashing*, a topic that we will see in the next chapter.

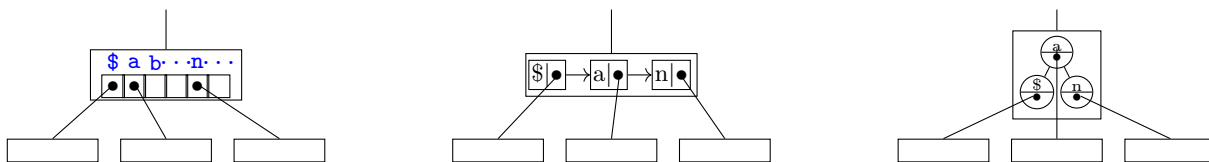


Figure 6.16: Three ways of implementing child-references in a multi-way trie.

Under the (realistic) assumption that the alphabet has constant size, the operations all take

$\Theta(|\Sigma| |x|) = \Theta(|x|)$ time in the worst case, and the space is $\Theta(|\Sigma| N) = \Theta(N)$ (where N is the number of nodes). So in theory the distinction between these approaches does not matter. In practice it does matter when $|\Sigma|$ is fairly big (e.g. for $\Sigma = \text{ASCII}$ each node might waste 128 memory cells), but which solution is best depends much on the size of the alphabet and how the words are distributed.

6.3 Take-home messages

- There are many possible ways to realize ADT Dictionary, but all require $\Omega(\log n)$ in the worst case for searching if they are comparison-based.
- If we know that the keys have a special structure, then one should consider implementations of ADT Dictionary that are tailored to this structure and hence might beat the lower bounds for searching.
- Interpolation search requires only minor changes from binary search and can achieve a vastly faster search-time in sorted arrays of integers.
- Tries are a very natural way of storing words, and can be made space-efficient with suitable compression techniques. The run-time for their operation depends only on the length of the word, and not on the number of stored words.

6.4 Historical notes

Binary search is an extremely natural method; it apparently was first mentioned in 1946 by Mauchly (the original reference appears unavailable, but Knuth credits him in his textbook [Knu98, p.422]). Knuth discusses some variants extensively; in particular one can avoid division by 2 by using an algorithm whose decision tree looks like a Fibonacci-tree. Interpolation search was first mentioned by Peterson in 1957 [Pet57], though he does not analyze its run-time tightly. Proving the $O(\log \log n)$ average-case run-time was done roughly 20 years later by two papers independently [PIA78, YY76]; the variant and proof given here were presented by Perl and Reingold [PR77].

Tries likewise are a very old concept; already in 1912 Thue used a trie to illustrate some words ([Thu12], a translation (sadly without the pictures) can be found in [Ber95]). The name ‘trie’ was coined by Fredkin [Fre60]. Compressed tries were introduced by Morrison [Mor68], with the particular purpose of storing occurrences of words (patterns) within one master text; we will see more on this in Chapter 9.

Chapter 7

Hashing

Contents

7.1	Direct addressing	201
7.2	Hashing overview	202
7.3	Collisions	203
7.3.1	Hashing with chaining	204
7.3.2	Hashing with probe sequences	208
7.3.3	Cuckoo hashing	215
7.3.4	Hashing run-time summary	217
7.4	Choosing hash functions	218
7.4.1	Universal hashing	220
7.4.2	Hash functions for other data	225
7.5	Take-home messages	226
7.6	Historical remarks	227

We now turn towards *hashing*, a realization of ADT Dictionary that is extremely popular in practice, but fairly difficult to analyze theoretically unless we make stringent assumptions on the input-data. In some ways it is a continuation of Chapter 6 (what can we do if all keys are integers?), but important enough to warrant its own chapter.

7.1 Direct addressing

As a warm-up, let us first consider a special situation, where we know some upper bound M and we know that all keys in our dictionary are integers in the range $\{0, \dots, M-1\}$. Then there is a very simple strategy for storing the keys in an array: Use the key itself to tell us the index! Thus, initialize an array A of size M , with the plan to store key-value pair (k, v) at $A[k]$. Since we already know the key from the index, we actually only need to store the value, so $A[k] \leftarrow v$. See Figure 7.1 for an example.

This strategy is called *direct addressing*, because we directly read the address for storage

0	1	2	3	4	5	6	7	8
		dog				cat		pig

Figure 7.1: Direct addressing example. We have $M = 9$ and three key-value pairs (2, dog), (6, cat) and (8, pig).

from the key. The operations for ADT Dictionary are extremely easy to implement:

- *search*(k): Check whether $A[k]$ is NIL. If it is, then k is not in the dictionary (we never store it anywhere else). If it is not, then return $A[k]$ as the associated value; we know that this is correct since keys are unique.
- *insert*(k, v): Set $A[k] \leftarrow v$. Since we assume that k was not in the dictionary previously, this array-entry was NIL previously, so we do not overwrite any information.
- *delete*(k): Set $A[k] \leftarrow \text{NIL}$, which indicates that k is no longer in the dictionary.

Notice in particular that all three operations take constant time! This is not a violation to Theorem 6.1 because we are exploiting the special structure of the keys (they are integers in $\{0, \dots, M-1\}$) to create the data structure. The space usage is $\Theta(M)$.

This realization should remind you a little bit of bucket sort. If all keys are distinct, then we could sort them by inserting them in an array A with direct addressing, and then reading them back out from A in order of indices. This takes $O(n + M)$ time (where M is the upper bound). The main difference between bucket sort and direct addressing is that for bucket sort keys may appear multiple times, leading to a *collision*: two keys are vying for the same slot in the array. Collisions do not happen with direct addressing because ADT Dictionary promises that keys are distinct, but will be an issue with hashing below.

The main drawback of direct addressing is the space-use. The required space is $\Theta(M)$, where M is an upper bound on the maximum key. This has two problems. First, M might be quite big, compared to the number n of keys. (Imagine for example that we use phone numbers, i.e., 10-digit base-10 numbers, as keys. Then you would use an array 10^{10} to store just a few phone numbers.) Second, and worse, we might not know M , and so may need to use some estimation that is vastly too big.

7.2 Hashing overview

The idea of *hashing* is the following: Given keys, find a way to map the keys to integers such then we can apply direct addressing (modified so that we can deal with collisions). To make this more specific, let us assume that we know that all keys come from some *universe* U . The idea is to pick some integer M (the *hash-table-size*), some function h that maps U to integers in $\{0, \dots, M-1\}$ (the *hash-function*), and then to use direct addressing and place (k, v) at $T[h(k)]$ in an array T of size M . See Figure 7.2 for an example.

Let us expand on this idea a bit, because there are many important details:

- As opposed to direct addressing, there are very few restrictions on the universe U that the

	0	1	2	3	4	5	6	7	8	9	10
$T :$		45	13		92	49		7			43

Figure 7.2: Hashing-example. Universe U consists of all natural numbers, we use $M = 11$, and the hash-function is $h(k) = k \bmod 11$. The hash table stores key-value pairs with keys 7, 13, 43, 45, 49, 92. (As usual, we only show the keys, not the values).

keys are drawn from. We mostly study the case where U consists of integers (and typically $U = \{0, \dots, |U| - 1\}$), but later discuss other universes such as words. The only restriction on U is that a hash function h can be found that is both quick to compute and behaves well (we will clarify later what this means).

- The hash-table size M is something that the user can pick. We will discuss choices for M later, but typically it is a good idea to use a prime number for M and to choose $M \in \Theta(n)$ (where as usual n is the number of stored key-value pairs). To achieve the latter, we will occasionally need to change M as n grows or shrinks; this is called *re-hashing* and is discussed below.
- The hash-function h is also something that the user picks, as long as it satisfies $h : U \rightarrow \{0, \dots, M-1\}$. This choice is very critical, and we will discuss good options later. The *modular hash function* $h(k) = k \bmod M$ that was used in Figure 7.2 is a very popular hash function. For most of our run-time discussions below, we assume that the hash-function can be computed in constant time.
- The array T used to store the items is called the *hash-table*, and one entry in T is called a *slot*. We say that a key k *has hash-value* i or *hashes to slot* $T[i]$ if $h(k) = i$.

Summarizing the hashing idea, each key k would like to be stored at the slot of the hash-table that it hashes to.

7.3 Collisions

The main difficulty with hashing is that we may have a *collision*, which means that two key-value pairs are vying for the same slot. In the example of Figure 7.2, if we had both key 46 and key 13 in the dictionary, then $h(46) = 2 = h(13)$, so both key-value pairs want to be stored at $T[2]$, which is obviously not possible.

Collisions are unavoidable if $n > M$, i.e., if the number of items stored exceeds the number of available slots. Even if $n < M$, it is quite likely for collisions to occur. You may have heard of the *birthday paradox*: There are 366 possible birthdays, but if you have 23 people in a room, then with probability more than half two of them have the same birthday. In a similar way, one can argue that among n keys in a hash-table of size M , if $n \approx \sqrt{M}$ then two keys collide with probability more than half. This holds even if the hash-function is distributing keys as evenly as it possibly could.

There are many strategies to deal with collisions; Figure 7.3 shows the ones that we will

study in the following sections.

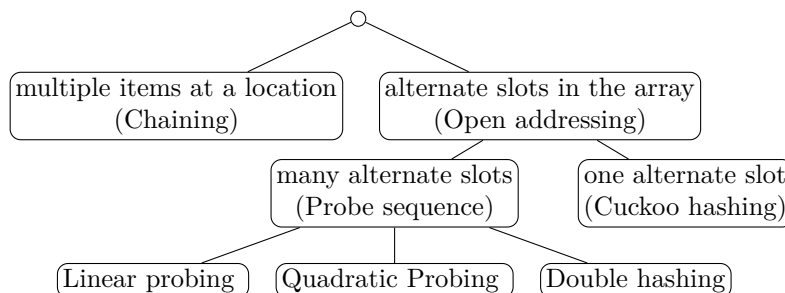


Figure 7.3: Collision-resolution strategies.

7.3.1 Hashing with chaining

The simplest collision strategy is to store all items that want one slot in this slot. Put differently, the slot $T[i]$ now refers to a data structure that can store multiple items. (We then call $T[i]$ a *bucket*.) Any realization of ADT Dictionary could be used for the buckets, but since there typically will not be many items competing for a slot, it is common to use an unsorted list. This is called *hashing with chaining* and should very much remind you of bucket sort. Figure 7.4 shows such a hash-table, using $M = 11$ and $h(k) = k \bmod 11$. (For ease of reading we show the hash-table downward.)

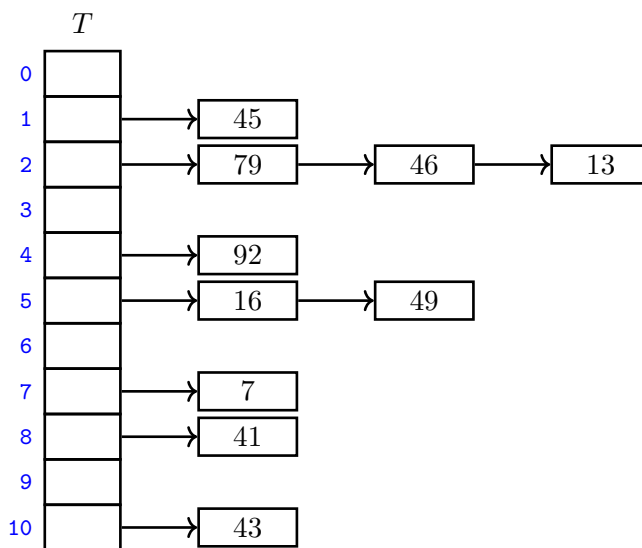


Figure 7.4: A hash-table that uses chaining to resolve collisions.

All operations are easily accomplished when applying hashing with chaining:

- To search for key k , compute $h(k)$ and then search for k in the list at $T[h(k)]$. Recall the move-to-front heuristic from Section 5.2.3; this should be applied here because it adds very little overhead and makes search much more efficient if the search-requests are biased.
- To insert a key-value pair (k, v) , compute $h(k)$ and then insert (k, v) in the list at $T[h(k)]$. (Recall that the pre-condition guarantees that k was not in the dictionary yet, so we do not need to check this.)
- To delete key k , first search for k , and then remove the corresponding list-item.

Insertion takes constant time; the other two operations have worst-case time $O(1+|T[h(k)]|)$, where $|T[h(k)]|$ denotes the size of the bucket at slot $h(k)$. In the worst case, a bucket could have size n if all keys hash to the same slot. (This is important to remember: In the worst case, hashing can be very bad!) But we would like to argue that on average, hashing performs well.

How to analyze hashing, and the uniform hashing assumption. So we want to analyze the run-time of hashing with chaining in the average case. At first sight, this seems easy. Consider the following result.

Observation 7.1. *The average size of a bucket is $\frac{n}{M}$.*

Proof. We have n keys and M buckets, so on average the bucket-size is $\frac{n}{M}$. \square

We will give a name to the fraction $\frac{n}{M}$, because it will be crucial for the analysis of hashing.

Definition 7.1. *A hash table with M slots storing n keys has load factor*

$$\alpha = \frac{n}{M} = \frac{\# \text{ items in the dictionary}}{\text{size of the hash-table}}$$

So Observation 7.1 can be re-stated as “the average size of a bucket is α ”. This is true, but unfortunately not very helpful. One might be tempted to conclude from Observation 7.1 that the average-case run-time for a search ought to be $O(1 + \alpha)$, but this argument has a flaw. Just because the average bucket size is small does not mean that searching for a key will hit an average bucket. We have ignored the influence of the hash-function!

To make this specific, consider the following (rather idiotic) hash-function that sets $h(k) = 0$ for all keys. Then *all* keys are being mapped to the same bucket, which hence has size n . The average bucket-size continues to be α , but any key that we search for needs to search in $T[0]$, which has length n . So the hash-table effectively acts like a list, and the average-case run-time to search in a list is $\Theta(n)$. So unless we specify assumptions about the hash-functions, we cannot hope to obtain bounds on the average-case run-time better than the trivial $O(n)$.

The same problem (in a more subtle way) can occur for *any* fixed hash-function h , assuming the universe is large enough (specifically $|U| \geq M(n-1) + 1$). By the pigeon-hole principle, then for some index i we must have at least n keys in U that hash to i . (Formally, if we let $U_i \subset U$ be all keys that hash to i , then $\max_i |U_i| \geq \frac{1}{M} \sum_i |U_i| = \frac{1}{M} |U| \geq (n-1) + \frac{1}{M}$ and so by integrality

max_i |U_i| ≥ n.) If we happen to use n such keys for the items in the dictionary, then again the hash-table acts like a list and the average-case run-time for searching for a key that is in the dictionary is $\Theta(n)$.

If we want to obtain better average-case bounds for hashing, we therefore have to make stringent assumptions on how the sets of keys are distributed. This is both difficult to state and difficult to analyze, and so instead (somewhat similar to what we did in Chapter 3) we switch to analyzing a *randomized* version of hashing. This might puzzle you. The keys in the dictionary are given to us, so we cannot use randomness for them. And the slot of each key is determined by the hash-function, so we cannot use randomness for the slot either. So how can we randomize hashing? The trick is to use randomness when *choosing the hash function*. In other words, whenever we determine the hash-function (when initializing the hash-table or when re-hashing), we *randomly* pick a hash-function. (This is not as outlandish as it might seem; we will see one version of randomized hashing in Section 7.4.1.)

We make the following *uniform hashing assumption*:

Every possible hash-function (for a given table-size M) is equally likely to be picked as the hash-function.

While this assumption makes the analysis much easier, it is unfortunately completely unrealistic as we will discuss later.

Let us write some consequence of the uniform hashing assumption which will be useful in the future.

Lemma 7.1. *Assume the uniform hashing assumption holds. Then for any key k and any table-slot i in a table of size M , we have*

$$P(h(k)=i)=\frac{1}{M},$$

where the random variable for $P(\cdot)$ is the choice of hash-function h .

Proof. For any hash-function h that maps k to i , we could define $M - 1$ other hash functions h_j (for $j \neq i$) that map k to j and all other keys to the same slot as in h . Therefore, a fraction of $\frac{1}{M}$ of all possible hash-functions map k to i , and since all hash-functions are chosen equally likely the probability holds. \square

We use the expression ‘hash-values are uniform’ as a convenient shortcut for ‘ $P(h(k)=i)=\frac{1}{M}$ for all k, i ’. With much the same argument one also shows (details are omitted):

Lemma 7.2. *Assume the uniform hashing assumption holds. Then for any keys $k \neq k'$ and any table-slots i, i' in a table of size M , we have*

$$P(h(k) = i \text{ and } h(k') = i') = \frac{1}{M^2} = P(h(k) = i) \cdot P(h(k') = i'),$$

where the random variable for $P(\cdot)$ is the choice of hash-function h .

In other words, the events that $h(k) = i$ and $h(k') = i'$ are independent of each other. We use the expression ‘hash-values are independent’ as a convenient shortcut for the conclusion of this lemma.

Expected run-time of hashing with chaining. So from now on we analyze the randomized versions of hashing under the uniform hashing assumption. Since this is now a randomized algorithm, we will analyze its expected run-time.

Lemma 7.3. *Under the uniform hashing assumption, for any key k the expected size of bucket $T[h(k)]$ is at most $1 + \alpha$.*

Proof. Let $i = h(k)$; we hence want to know the size of $T[i]$. We have two cases. If k is not in the dictionary, then each of the n dictionary-items hashes to i with probability $\frac{1}{M}$, hence $E[|T[i|] = \frac{n}{M} = \alpha$. If k is in the dictionary, then it definitely belongs to $T[i]$. Of the $n - 1$ remaining dictionary-items, each hashes to i with probability $\frac{1}{M}$ since hash-values are independent. Therefore the expected size of $T[i]$ is $1 + \frac{n-1}{M} \leq 1 + \alpha$. \square

In consequence, under the uniform hashing assumption, searching for a key uses an expected number of at most $1 + \alpha$ key-comparisons. If we search for a key that we know to be in the dictionary, then on average over all keys there will be only $1 + \frac{1}{2}\alpha$ key-comparisons, because searching in a list of length ℓ takes $\frac{1}{2}(\ell + 1)$ comparisons on average over all items in the list.

Thus the expected run-time for search and delete in hashing with chaining is $\Theta(1 + \alpha)$. (The time for *insert* is in $O(1)$.) The space-use is $\Theta(n + M)$.

General technique: Re-hashing. We have no control over how many keys get inserted into the dictionary, but recall that we do have control over the table-size M . Since the run-time for hashing with chaining is $\Theta(1 + \alpha)$, we want to keep α small by increasing M when needed. This is called *re-hashing*, and works quite similar to what was done for dynamic arrays:

- Start with some initial small table-size M .
- During *insert* and *delete*, keep track of the size n of the dictionary.
- With this we always know the load-factor $\alpha = \frac{n}{M}$.
- If α gets “too big” (see below):
 - Choose a new table-size M' . Typically we choose $M' \geq 2M$, so that the new load factor α' satisfies $\alpha' \leq \frac{1}{2}\alpha$.
 - Find a new hash-function h' that maps to $\{0, \dots, M' - 1\}$.
 - Create a new hash-table T' of size M' .
 - Go through the entire hash-table T , remove each item and insert it in the new hash-table T' using hash function h' .

Note that re-hashing takes $\Theta(M' + n)$ time when doing hashing with chaining, and generally takes time $\Theta(M' + n * \textit{insert})$. Just as for dynamic arrays, this operation happens rarely enough that on average over all operations it roughly doubles the run-time. Hence amortized over all operations we can ignore the run-time needed for re-hashing.

Analysis of hashing with chaining continued. If we re-hash as needed then we can keep the load factor as small as we choose it to be. For example, we could re-hash whenever the load-factor exceeds 1 (i.e., whenever $n > M$). With this, under the uniform hashing assumption,

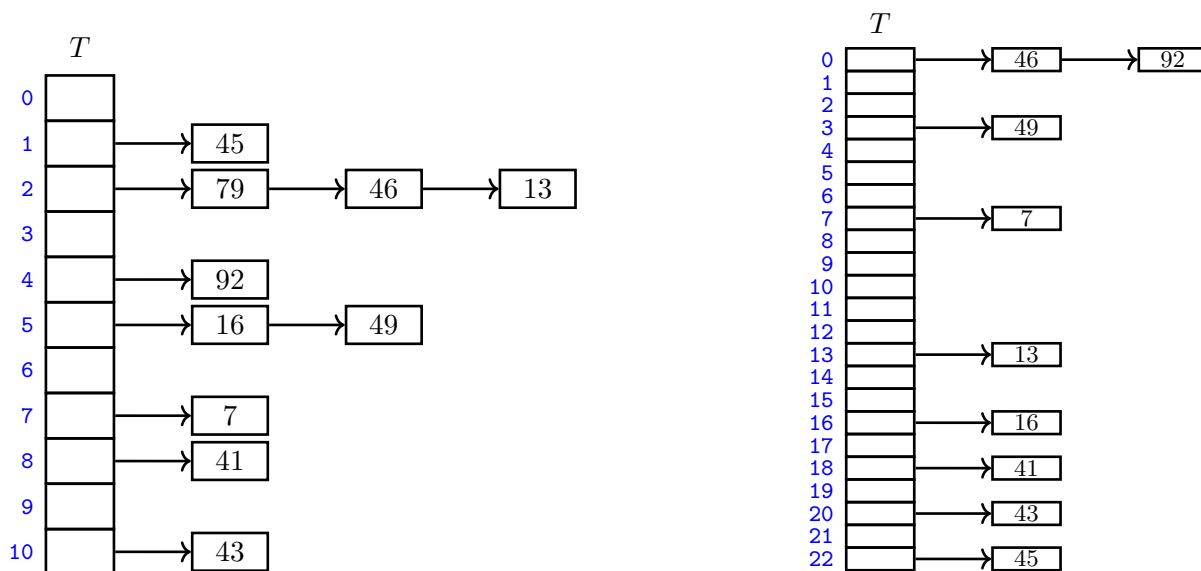


Figure 7.5: Rehashing does not remove all collisions, but makes them less likely. We use $M' = 23$ and $h'(k) = k \bmod M'$.

the expected time for all operation in hashing with chaining is $O(1)$ amortized. But, as said earlier, this crucially uses the uniform hashing assumption—if you use a fixed hash-function, and especially if the keys are unevenly distributed and the hash-function not tailored to this, then these bounds do not hold.

Also recall that the space used was $\Theta(M + n)$. If we let the load factor get *too* small, then M is much much bigger than n , and so we waste space. For this reason, one should also re-hash if α gets too small during a deletion. For example, we could cut the size of the hash-table approximately in half whenever $\alpha < \frac{1}{4}$ (i.e., $M > 4n$). With this, the space used by hashing with chaining is $\Theta(n)$.

7.3.2 Hashing with probe sequences

In some sense, hashing with chaining is as good as we could wish for: If we maintain a load factor between $\frac{1}{4}$ and 1, then the space is $\Theta(n)$ and the expected run-time for all operations is $\Theta(1)$ assuming uniform hashing.

Nevertheless, hashing with chaining is not very popular. The reason is that while the space is $\Theta(n)$, the constant hidden by Θ is quite big. Each hash-table slot is a list, which might need overhead even if the list is empty. Each key-value pair is stored in one list-item, which again needs overhead for the reference. In practice, it would be preferable to do away with the list.

This section hence does collision resolution in an entirely different way: Allow a key to be at multiple spaces in the hash-table, and try them all. Specifically, for each key k we define a

probe sequence, which is a list of possible slots that key k is allowed to use in the hash table, ordered by preference. We denote this by

$$\langle h(k, 0), h(k, 1), h(k, 2), \dots \rangle$$

where $h(k, 0)$ is the preferred slot for key k , $h(k, 1)$ is the next-best slot for key k , etc. To give just one example, one easy-to-define probe-sequence is to use $h(k, i) = (k + i) \bmod M$, i.e., if one slot is full then simply try the immediate adjacent one. (We will discuss below why this is not a great strategy.)

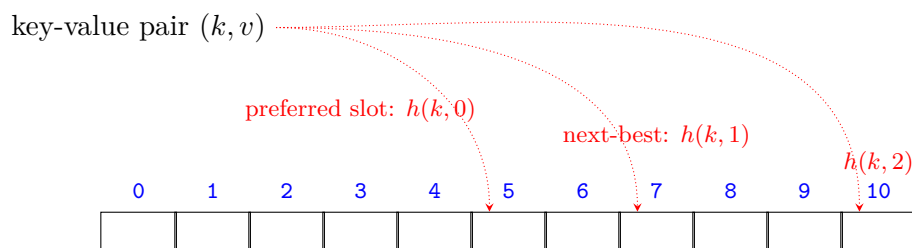


Figure 7.6: The idea of probe-sequences.

Operations. Let us consider how to do the operations for hashing with probe sequences:

- To search for a key k , we compare it with the key at $T[h(k, 0)]$, then with the key at $T[h(k, 1)]$, etc.. Each of these comparisons is called a *probe*, hence the name ‘probe sequence’. We continue probing until we either find the key (then we call this a *successful search*) or an empty slot (an *unsuccessful search*). Encountering an empty slot means that k was not in the hash-table because it would have been put into this slot.
- To insert a key-value pair (k, v) , we first search in the hash-table for k . This should be unsuccessful, so we stop when we find an empty slot. Then insert (k, v) there. See Figure 7.7.

0	1	2	3	4	5	6	7	8	9	10
	45	13		92	49		7	41	84	43

0	1	2	3	4	5	6	7	8	9	10
20	45	13		92	49		7	41	84	43

Figure 7.7: Inserting key 20 for $h(k, 0) = k \bmod 11$. We have $h(20, 0) = 9$, and hence must probe at three slots before finding an empty one.

However, deletion is a problem. We can search for and find the key k that we wish to delete, but if we simply remove the corresponding key-value pair from the hash-table, then

4423 future searches might not work correctly. Consider removing key 43 from the bottom hash-table
 4424 of Figure 7.7. If we simply deleted it, then slot $T[10]$ is empty. A future search for key 20 would
 4425 hence search at slot $T[9]$, then at slot $T[10]$, find an empty slot there and conclude that 20 is
 4426 not in the dictionary. Incorrect! Instead we apply the *lazy deletion* technique where we only
 4427 mark the entry as “deleted”. (Section 4.1.3 reviews this in more detail.) For hashing with probe
 4428 sequences, this specifically works as follows:

- 4429 • When wanting to delete key k , search for k , and mark the slot where it is found as ‘deleted’.
- 4430 • If a search encounters a ‘deleted’ slot, then keep searching.
- 4431 • If an insertion encounters a ‘deleted’ slot, then use it for insertion.
- 4432 • Maintain a global counter how many slots are marked ‘deleted’. If this counter gets too
 4433 big (say $n/2$ of the slots are marked ‘deleted’) then re-hash.

0	1	2	3	4	5	6	7	8	9	10
20	45	13		92	49		7	41	84	deleted

Figure 7.8: Deleting key 43 by applying lazy deletion.

Algorithm 7.1: *probeSequenceHash::insert(k, v)*

```

1 re-hash the table the load factor is too big
2 for ( $j = 0; j < M; j++$ ) do
3   if  $T[h(k, j)]$  is NIL or ‘deleted’ then
4      $T[h(k, j)] \leftarrow (k, v)$ 
5     return “item inserted at index  $h(k, j)$ ”
6   else
7     return “failure to insert”                                     // need to re-hash

```

Algorithm 7.2: *probeSequenceHash::search(k)*

```

1 for ( $j = 0; j < M; j++$ ) do
2   if  $T[h(k, j)]$  is NIL then
3     return “item not found”
4   else if  $T[h(k, j)]$  has key  $k$  then
5     return “item found at index  $h(k, j)$ ”
6   else // the current slot was ‘deleted’ or contains a different key
7     // ignore this and keep searching
7 return “item not found”

```

Algorithm 7.3: *probeSequenceHash::delete(k)*

```

1  $i \leftarrow$  index returned by probeSequenceHash::search(k)
2  $T[i] \leftarrow$  ‘deleted’
3 re-hash the table if there are too many ‘deleted’ items

```

The pseudo-code for all three operations is given in Algorithms 7.1-7.3. Note that *insert* has an “emergency break”: we stop searching for an unused slot after M probes. At this point, we have looked at all slots (or, if the probe sequence was not good, looked at a subset of slots repeatedly), and further probes will not bring us to slots that are empty. So we must re-hash. One should actually set the break much earlier, and for example stop after 10 probes. As we will see later in Table 7.12, if the load-factor α is kept small, the expected number of probes is a small constant. So if we use more than 10 probes, then this strongly indicates that we need to re-hash. We add a similar emergency break to *search*, though it should never be used if we use a probe-sequence that reaches all slots in the table and if we keep the load factor below 1.

Choosing probe sequences. It remains to discuss how to choose the probe sequence. There are many options here.

- **Linear probing:** We have already seen this strategy; it consists of setting $h(k, i) = (h(k) + i) \bmod M$. In other words, fix a hash function $h(k)$, and then use as probe sequence for key k the slot $h(k)$, then the next slot, then the next slot, etc., in the table. Linear probing has the advantage of being very easy to describe and implement. Also, it guarantees that if there is an empty slot in the table, then the probe sequence will find it, so *insert* always succeeds if there is any space in the table. However, it has the disadvantage of forming large *clusters*, which are consecutive slots that are occupied. If any future key hashes to *any* slot in such a cluster, then it will first need to inspect all later slots in the cluster, and it will then take the next slot, making the cluster even larger. Thus in practice linear problem will be slow.
- **Quadratic probing:** This is similar to linear probing, but uses a quadratic dependence on i . Thus, set $h(k, i) = (h(k) + c_1 i^2 + c_2 i) \bmod M$ for some constants c_1 and c_2 . This is just as easy to compute as linear probing, but eases the clustering problem a bit: Keys that hash to the same slot will still form a cluster, but keys that hash to different slots will follow different sequences and hence clusters will be smaller. It has the disadvantage that c_1 and c_2 need to be chosen carefully, else the probe sequence might not even visit all slots of the table.
- **Double hashing:** This avoids the above clustering problems by using *two* hash functions h_0 and h_1 , and setting $h(k, i) = (h_0(k) + i \cdot h_1(k)) \bmod M$. Double hashing, if done right, performs very well in practice. However, there are a number of issues that must be considered. First, we need $h_1(k) \neq 0$ for all keys, because otherwise the probe sequence will always try the same slot. Second, we really should have that $h_1(k)$

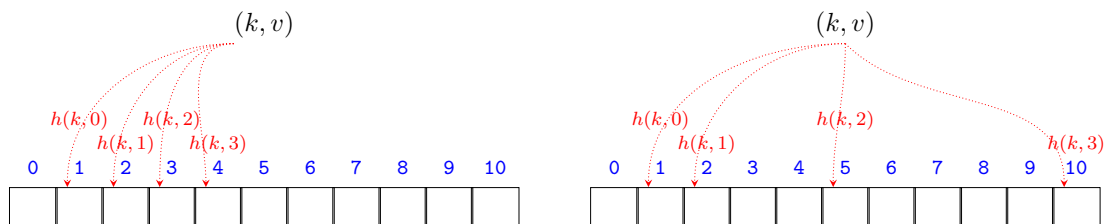


Figure 7.9: Linear probing (with $h(k) = 1$) and quadratic probing (with $h(k, i) = 1 + i^2$).

is relative prime with the table-size M , so that the probe-sequence explores all slots of the table. (One easy way to achieve this is to set M itself to be a prime number.)

General technique: Independent hash functions. The third, and most subtle, problem for double hashing is that h_0 and h_1 should be *independent*. Recall that throughout our theoretical analysis, we think of hash functions as having been randomly chosen among all possible hash functions. For two hash functions to be independent means that the two random variables that indicate this choice are independent.

In our examples (and also in practice, where hash functions are often not chosen randomly), one uses two hash functions that follow radically different methods of designing hash functions. We already saw the

$$\text{modular method: } h(k) = k \bmod M.$$

(It is also sometimes called *division method* since it we take the remainder when dividing k by M .) A very different strategy for finding a hash-function is to use the

$$\text{multiplication method: } h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor,$$

for some floating-point number A with $0 < A < 1$. The formula looks complicated, but the idea is simple:

- First multiply k with A (we will discuss choices for A in Section 7.4).
- Then take the fractional part (i.e., the part after the decimal point) of the result. The mathematical formula for this is $kA - \lfloor kA \rfloor$. Note that this gives a floating point a number in the interval $[0, 1)$.
- Next, multiply the result with M . This gives a floating-point number in the interval $[0, M)$.
- Finally, round the number down to get an integer in $\{0, \dots, M - 1\}$ as required.

Returning to double hashing, recall that we wanted an independent hash function, but we also required that $h_1(k)$ is non-zero. We can achieve this by modifying the multiplication method a little bit, and setting $h_1(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$. Figure 7.10 shows an example of inserting with double-hashing.

0	1	2	3	4	5	6	7	8	9	10
	45	13		92	49		7	41		43

0	1	2	3	4	5	6	7	8	9	10
	45	13	194	92	49		7	41		43

Figure 7.10: Inserting key 194 with double-hashing; we use $M = 11$, $h_0(k) = k \bmod M$ and $h_1(k) = 1 + 10[(\phi k - \lfloor \phi k \rfloor)]$, where $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618033\dots$. We have $h_0(194) = 7$ and $h_1(194) = 9$, so the probe-sequence for key 194 is $\langle 7, 5, 3, 1, \dots \rangle$.

Analysis of hashing with uniform probing (cs240e) . Analysis of hashing with probe sequences is significantly more complicated than for hashing with chaining, because now for different keys the probe sequences interact with each other. We will give no details for how to analyze linear probing and double hashing, but analyze here an idealized version where the probe sequences are well distributed. As for the uniform hashing assumption, we phrase this via a randomized approach, i.e., as if the probe sequence for each key had been chosen randomly. We analyze hashing with probe sequences under the following *uniform probing assumption*:

For any key k , the probe sequence $\langle h(k, 0), h(k, 1), \dots, h(k, M-1) \rangle$ is a permutation of $\{0, \dots, M-1\}$ that has been randomly chosen among the $M!$ permutations.

This assumption is stronger than the uniform hashing assumption, because it assumes not only that $h(k, 0)$ is equally likely any of the M slots, but also that later entries in the probe sequence follow no particular pattern. This assumption clearly does *not* hold for linear probing. It also does not hold for double hashing, because double hashing uses only $M(M-1)$ different probe-sequences, rather than $M!$ possible sequences. (However, the run-time bounds that can be shown for double hashing are actually almost as good as the ones for uniform probing.)

For our analysis under the uniform probing assumption, we need the following consequence:

Lemma 7.4. *Assume that uniform probing assumption holds and we have inserted n keys. Then for any $0 \leq i < M$ we have $P(\text{slot } i \text{ is occupied}) = \frac{n}{M} = \alpha$, where the random variable for $P(\cdot)$ is the choice of probe sequences.*

Proof. Let \mathcal{S}_i be the set of all choices of probe sequences for the n keys that result in an occupied slot i . Observe that for any $i \neq j$, we have $|\mathcal{S}_i| = |\mathcal{S}_j|$, because for any set of probe sequences in \mathcal{S}_i we can find one in \mathcal{S}_j by increasing all assigned slots by $j - i \pmod{M}$. Since all choices of probe sequences are equally likely to be used, the probability that i is occupied is the same for all i . Since n of the M slots are occupied, the bound follows. \square

A similar argument shows that the probability that some slot i is occupied is independent from the probability that some other slot j is occupied. With this, we can analyze the expected number of probes during a search.

Lemma 7.5. *Under the uniform probing assumption, the expected number of probes for a search in hashing with chaining is at most $\frac{1}{1-\alpha}$.*

Proof. Assume we are searching for some key k . Let X be the random variable that denotes the number of probes until we find an empty slot, so we want to bound $E[X]$.

Clearly $X \geq 1$ (we always probe at least once). The probability that we probe ℓ times can be upper-bounded by the probability that if the first $\ell - 1$ probes all hit occupied slots. (This is an upper bound since those $\ell - 1$ probes might have led us to find the key and stop the search.) By Lemma 7.4 each slot has probability α of being occupied, and these probabilities are independent for the $\ell - 1$ different slots. So the probability that the first $\ell - 1$ probes all hit occupied slots is $\alpha^{\ell-1}$ and

$$E[X] = \sum_{\ell \geq 1} P(X \geq \ell) \leq \sum_{\ell \geq 1} \alpha^{\ell-1} = \sum_{\ell \geq 0} \alpha^{\ell} = \frac{1}{1-\alpha}.$$

□

The above upper bound holds for any search, whether it is successful or unsuccessful. We can find a tighter bound for a successful search if we consider the *average-instance expected time*. You may be puzzled by this intermixing of ‘average’ and ‘expected’, but they refer to two different things: ‘expected’ refers to the randomly chosen probe sequences, while ‘average’ refers to the keys that we could search for. Formally, assume we store the keys x_0, \dots, x_{n-1} . Then an instance for a successful search is to search for one key x_j , and the average-instance time is hence taking the average of the expected search-time over these n keys.

Lemma 7.6. *Under the uniform probing assumption, the average-instance expected number of probes for a successful search is at most $\approx \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$.*

To see the difference for the bound, consider the case where α is very near to 1, say $\alpha = 0.999$. (We should not let α get this big in practice!) Then the upper bound for any search is $1/(1-\alpha) = 1000$. But on average for the keys in the dictionary, the search will take no more than $\frac{1}{\alpha} \log(1000) < 7$ probes.

Proof. At the time of inserting x_j (for $j = 0, \dots, n-1$), there were j items already in the dictionary, hence the load factor was $\alpha_j = \frac{j}{M}$. So the expected number of probes until we find an empty slot for x_j is at most $\frac{1}{1-\alpha_j} = \frac{1}{1-(j/M)} = \frac{M}{M-j}$ by the previous lemma. We put x_j at this empty slot, and so when searching for x_j later, the expected number of probes is at most $\frac{M}{M-j}$. Taking the average over all keys, therefore the average expected number of probes is at most

$$\frac{1}{n} \sum_{j=0}^{n-1} \frac{M}{M-j} = \frac{M}{n} \sum_{j=0}^{n-1} \frac{1}{M-j} = \frac{1}{\alpha} \sum_{\ell=M-n+1}^M \frac{1}{\ell}$$

4537 after substituting $\ell = M - j$ in the sum. Recall the harmonic number $H_L = \sum_{i=1}^L \frac{1}{i}$ approxi-
 4538 mately equals $\ln(L)$, and therefore

$$\begin{aligned} \frac{1}{\alpha} \sum_{\ell=M-n+1}^M \frac{1}{\ell} &= \frac{1}{\alpha} \left(\sum_{\ell=1}^M \frac{1}{\ell} - \sum_{\ell=1}^{M-n} \frac{1}{\ell} \right) = \frac{1}{\alpha} (H_M - H_{M-n}) \\ &\approx \frac{1}{\alpha} (\ln M - \ln(M-n)) = \frac{1}{\alpha} \ln \frac{M}{M-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

4539

□

4540 7.3.3 Cuckoo hashing

4541 We now describe one other method of collision resolution called *cuckoo hashing* (the reasons for
 4542 this name will be clear soon). It works well in practice, in particular both *search* and *delete* take
 4543 constant *worst-case* time (ignoring the time for re-hashing).

4544 The main idea is to have two independent hash functions h_0, h_1 and two hash tables T_0, T_1 .
 4545 The two hash-table may or may not have the same size. We make the following promise:

4546 Key k is *always* at $T_0[h_0(k)]$ or $T_1[h_1(k)]$

4547 With this, both *search* and *delete* are extremely simple. Namely, to search for key k , search at
 4548 the only two possible slots; if it is not found there then it is not in the table. Likewise to delete
 4549 the item with key k , search for k and remove it from where it is found. (We do not need to do
 4550 lazy deletion with cuckoo hashing.)

4551 Note that the $O(1)$ worst-case run-time for searching beats the $\Omega(\log n)$ lower bound from
 4552 Theorem 6.1. This is not a contradiction because hashing relies on the keys being integers or
 4553 other items that are easily mapped to integers; it therefore is not comparison-based.

4554 The price to pay for the fast search and deletion in cuckoo hashing is that insertion is non-
 4555 trivial and potentially slow. Algorithm 7.4 shows the pseudo-code. Let us illustrate how this
 4556 algorithm work on tracing how we insert key 95 in the leftmost hash table in Figure 7.11.

- 4557 • We initially put the key k to be inserted at $T_0[h_0(k)]$. For $k = 95$, we have $h_0(95) = 7$ and
 4558 so we put it at $T_0[7]$.
- 4559 • But $T_0[7]$ was already occupied previously, in this case by key $k' = 51$. Here is where the
 4560 name “cuckoo hashing” comes from: we ‘kick 51 out of the nest’. Put differently, key k
 4561 *always* takes slot $T_0[h_0(k)]$, and if this slot was already occupied then the key k' that was
 4562 there needs to be put elsewhere.

4563 In the code, this is expressed via the swap in line 7: (k, v) takes slot $T_i[h(k)]$ while key k'
 4564 (and its associated value) are now stored in (k, v) and need to be inserted.

- 4565 • Now key 51 tries ‘the other table’ (in the code, this is expressed via ‘ $i \leftarrow 1 - i$ ’ in line 8).
 4566 So it gets inserted at $T_1[h_1(51)]$, which is $T_1[5]$.

4567 In this example we were lucky and $T_1[h_1(51)]$ was available. If it is not, then the process repeats.

4568 Let us see what happens when we try to insert key 26 in middle hash table in Figure 7.11.

Algorithm 7.4: *cuckooHash::insert(k, v)*

```

1  $i \leftarrow 0$ 
2 repeat
3   if  $T_i[h_i(k)]$  is NIL then
4      $T_i[h_i(k)] \leftarrow (k, v)$ 
5     return “success”
6   else
7      $\text{swap}((k, v), T_i[h_i(k)])$ 
8      $i \leftarrow 1 - i$ 
9 until we have tried  $2n$  times, where  $n = \text{size}$  includes new item
   // If we reach this point then the hash table is probably too full.
10 return “unsuccessful, should re-hash”

```

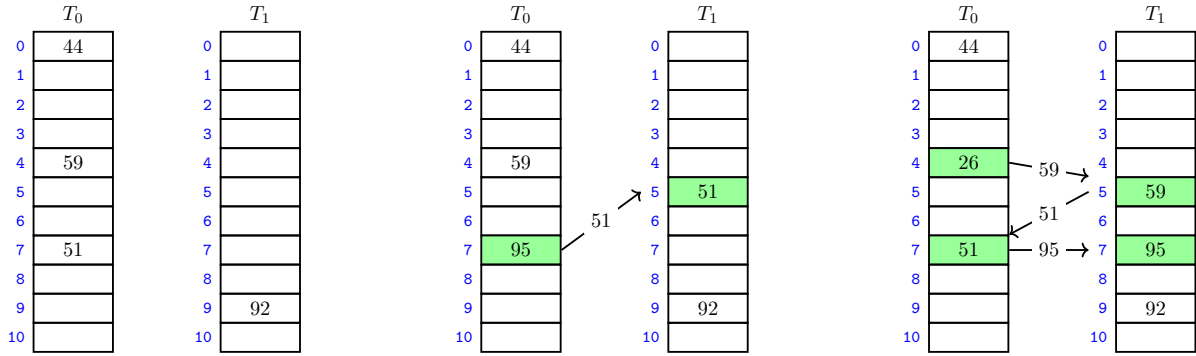


Figure 7.11: Inserting 95 and 26 when using hash-functions $h_0(k) = k \bmod 11$ and $h_1(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor$, where $\phi = 0.618033 \approx \frac{\sqrt{5}-1}{2}$.

- We put k at $T_0[h_0(k)]$. For $k = 26$, we have $h_0(26) = 4$ and so we put it at $T_0[4]$.
- But $T_0[4]$ was already occupied, in this case by key $k' = 59$. Thus key 59 is kicked out and tries the other table T_1 . It gets inserted at $T_1[h_1(59)]$, which is $T_1[5]$.
- But $T_1[5]$ was already occupied, in this case by 51. So 51 is kicked out and tries to get inserted in the other table (now back to T_0).
- This repeats once more: 51 kicks out 95 in table T_0 , so 95 tries table T_1 , where it finally finds an empty slot and the insertion routine ends.

Notice that there is no a-priori bound on how often one key kicks out another and how long this needs to continue until we find an empty slot. In fact, even with just three keys we could run into an infinite loop if they all three happen to hash to the same two slots. This is why the code has an ‘emergency break’. If we have kicked keys out $2n$ times, then there must have been a cycle formed by the process of kicking out, and it is pointless to try longer. This is an indication

that either the hash-tables are too full or the hash-functions were too badly distributed. The solution in this case is to re-hash the entire hash table and then to try insertion again. This emergency break should actually be applied much earlier (after a constant number of probes), because again the expected number of probes is constant (see Table 7.12).

We should also briefly discuss the space. As usual, we can keep the space used by the hash-table in $\Theta(n)$ by keeping the load-factor as a constant. (For cuckoo-hashing, we measure the load-factor as the number of items divided by the sum of the two table-sizes.) For cuckoo-hashing, it is usually recommended to keep the load-factor between $\frac{1}{4}$ and $\frac{1}{8}$, because as we discuss below it must be kept below $\frac{1}{2}$ for the analysis to work out. However, there is a possibility that we are forced to let the load-factor become even smaller (hence wasting space) if *insert* fails to insert a new item successfully and we therefore must re-hash. One can argue that this is rare and that therefore the expected space for cuckoo hashing is $O(n)$.

There are variations of cuckoo hashing that one could consider. For example, there is no need to keep the two hash tables separate; we could combine them into one table (and perhaps even open up both hash functions to use slots in both parts). It also seems rather strange that when inserting 26 in Figure 7.11, we did not even try to insert it in $T_1[h_1(26)]$ (which would have had a free slot); a natural variation of the insertion-routine could try this and on occasion save some switching. Finally, rather than two tables one could use three or four or even more tables, though in practice two or three tables seem to be the best.

7.3.4 Hashing run-time summary

We have seen a bit of the analysis for hashing with chaining (assuming uniform hashing) and for hashing with probe sequences (assuming uniform probing). With much more difficult proofs, one can analyze the expected number of some of the other probing strategies as well. We will not give details but only state the results in Table 7.12.

Warning: These bounds only hold under some assumptions. First, it is assumed that we re-hash when needed to keep α small. In particular, when using probe sequences we require $\alpha < 1$ (otherwise the hash table is full and there are no empty slots to insert). For cuckoo hashing we require $\alpha < \frac{1}{2}$. In consequence, the bounds for insertion and deletion only hold in an amortized sense. Also, the bounds use the uniform hashing assumption, and for double hashing and cuckoo hashing, the secondary hash function must be chosen independently from the primary hash function.

By default the stated bound on the number of comparisons is the expected-luck worst-instance number, i.e., it holds for any choice of key when taking the expectation over all possible random choices of hash-functions. We indicate places where the bound uses some other model, e.g. if it holds even if the hash-function is as bad as it can be (“worst-luck”) or if it holds only when averaging over all possible search-keys (“average-instance”).

You are not expected to memorize this table, but you should notice the following: As long as α is sufficiently small, all bounds resolve to $O(1)$. Recall that the load factor α is something

	Number of key-comparisons is at most:			Reference
	<i>insert</i>	<i>search</i>	<i>delete</i> (<i>successful search</i>)	
Chaining	1 (worst-luck)	$1 + \alpha$	$1 + \frac{1}{2}\alpha$ (avg-inst.)	Lemma 7.3
Linear Probing	$\frac{1}{2}\alpha \frac{2-\alpha}{(1-\alpha)^2}$	$\frac{1}{2}\alpha \frac{2-\alpha}{(1-\alpha)^2}$	$\frac{1}{2}\alpha \frac{2-\alpha}{1-\alpha}$ (avg-inst.)	[KW66]
Double Hashing	$\frac{1}{1-\alpha} + o(1)$	$\frac{1}{1-\alpha} + o(1)$	$\frac{1}{1-\alpha} + o(1)$	[LM93]
Uniform Probing	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \log \left(\frac{1}{1-\alpha} \right)$ (avg-inst.)	Lemma 7.5-7.6
Cuckoo Hashing	$\frac{\alpha}{(1-2\alpha)^2}$	1 (worst-luck)	1 (worst-luck)	[PR04]

Table 7.12: Summary of hashing results. We list an upper bound on the expected-luck worst-instance number of comparisons except where indicated otherwise.

that we have control over by re-hashing as needed; so we can achieve $O(1)$ amortized expected run-time for all operations.

However, all this only holds if we assume uniform hashing or uniform probing, neither of which is realistic in practice. If you fix one (or two) hash-functions, then for some sets of keys the run-times become $\Omega(n)$ for all operations except those where we wrote ‘worst-luck’.

7.4 Choosing hash functions

So far, we have done hashing analysis under the assumption that the hash function has been chosen randomly among all possible hash functions. This is completely unrealistic, because the only way to make this happen would be to write down for each key what the hash-function maps it to. But to store this information in a way that we can look it up when needed would require a dictionary for the keys... which is what we are trying to build in the first place.

So in real life, we must pick from a limited set of hash-functions to make it possible to compute the hash-function in constant time. If keys all are used equally often, then we could still achieve some useful properties (such as uniform hash-values) quite easily. But in practice, keys are not all equally likely. Without knowing more about the distribution of the keys, we can give here only some general guidelines:

- The hash function should avoid patterns that might exist in the data. For example, if the data contains prices, then the last digit will very frequently be a ‘9’. A hash function that depends heavily on the last digit (such as the modular hash function for $M = 10$ or $M = 100$) would be disastrous for such data.

In another example, ID numbers are often assigned consecutively, which means that the

first few digits are the same for all. (For example, all student ID numbers at UWaterloo seem to start with ‘20’ right now.) This makes the data very biased towards some values, and the hash function should be tailored to this. (For example, when storing UW student ID numbers, one might want to use only the last 6 digits for hashing.)

- On the other hand, the hash function should not ignore parts of the key that does contain information. For example, using $h(k) = k \bmod 100$ for base-10 numbers ignores all but the last two digits and would not be a good choice.

What does this mean for the specific methods that we have seen already? For the modular method, the choice of the table-size M is critical. We should *not* use $M = 10^\ell$ (if the keys are numbers in base-10) or $M = 2^\ell$ (if the keys are numbers in base-2), since this would effectively ignore the leading digits/bits of the number. This is a real pity, because $k \bmod 2^\ell$ would be especially fast to compute via bit-shifting. Instead, the recommendation is to use a prime number as M , which should (or so one would hope) avoid patterns in the data.

Finding a prime number of approximately the correct size is not trivial, but also not a big problem because we only need it during re-hashing, and during re-hashing we spend $\Theta(n + M)$ time for copying items over anyway. We want our prime to be of size at least $2M$ and by Bertrand’s postulate we know that there exists a prime that is no bigger than $4M$. Finding a prime within this range is straight-forward to do in run-time $O(M\sqrt{M})$ (for each candidate x , check whether any of the integers up to \sqrt{x} divides it), and with a bit more effort it can be done in $O(M \log \log M)$ time. (Details are omitted.) Therefore this is not a significant overhead.

For the multiplicative method, the table-size M is unimportant, and in fact using $M = 2^m$ for some integer m is not only acceptable but significantly eases the computation of the hash-function. On the other hand, the choice of the multiplicative factor A is extremely important. We want that the fractional part of $A \cdot k$ is evenly distributed over the interval $[0, 1)$, and for this, need that A has many decimal places not following any particular pattern. The ideal choice for A would be an irrational number, such as the *golden ratio* $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618033$. (Using the multiplicative method with $A = \phi$ is also called *Fibonacci hashing*.)

Dealing with the irrational number (cs240e) . However, we cannot store true irrational numbers in the computer, and the more bits we use to approximate it, the longer the computation of the hash-function will take, something we want to avoid.

To see how many bits of A we should really use, we will describe in a different way what the multiplication method does. Let us write $A = 0.a_1a_2a_3 \dots$ (in base 2). Let us also assume that the possible keys are the integers in $[0, \text{MAX_INT}]$, which in particular that we can write any key k using its bit-representation b_1, \dots, b_w where $w = \log(\text{MAX_INT} + 1)$. (In most computer

architectures we have $w = 16$ or $w = 32$, but in the example below, $w = 6$.)

$$\begin{array}{rcl}
 & & \begin{array}{c|c|c} \text{(leading bits)} & & \text{(bits of fractional part)} \\ \hline & & \begin{array}{c} 1 \quad 2 \quad 3 \end{array} \end{array} \\
 A \cdot k & = & \begin{array}{c} 0 \ 0 \ 0 \ \dots \ 0 \ 0 \\ +a_1 \cdot \ 0 \ b_1 b_2 b_3 \ \dots \ b_5 \ b_6 \\ +a_2 \cdot \ 0 \ 0 \ b_1 b_2 b_3 \ \dots \ b_5 \ b_6 \\ +a_3 \cdot \ 0 \ 0 \ 0 \ b_1 b_2 b_3 \ \dots \ b_5 \ b_6 \\ \vdots \\ +a_5 \cdot \ 0 \ 0 \ 0 \ 0 \ 0 \ b_1 \ b_2 \ b_3 \ \dots \ b_5 \ b_6 \\ +a_6 \cdot \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ b_1 \ b_2 \ b_3 \ \dots \ b_5 \ b_6 \\ +a_7 \cdot \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ b_1 \ b_2 \ b_3 \ \dots \ b_5 \ b_6 \\ +a_8 \cdot \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ b_1 \ b_2 \ b_3 \ \dots \ b_5 \ b_6 \\ \vdots \end{array}
 \end{array}$$

If we choose M to be a power of 2, say $M = 2^m$, then $h(k)$ corresponds to the m leading bits of the fractional part of $A \cdot k$. (In the example, $m = 3$.) How are these bits determined? The i th bit of the fractional part depends on bits a_i, \dots, a_{w+i-1} and all bits of k (plus any carry-overs that come from later bits). This illustrates somewhat why the multiplicative method tends to work well: All bits of $h(k)$ depend on all bits of k , or at least they could (it depends on the a_j 's).

We want the first m bits of the fractional part for $h(k)$, so we should definitely know at least $w + m - 1$ bits of A , and perhaps one more to account for the carryover from later bits. But bits of A beyond the first $w + m$ will make little difference. As such, we do not really have to handle the irrational number A , but we should use roughly the $w + \log M$ bits of A , where w is the maximum number of bits that can fit into one memory cell.

7.4.1 Universal hashing

We discussed that choosing a hash-function randomly and uniformly is impossible if we want to compute the hash-value in constant time. On the other hand, fixing one hash-function means that for some set of keys the operations will take $\Omega(n)$ time. In this section, we give a compromise between the two extremes. We choose the hash-function randomly, but only among a limited family \mathcal{H} of possible hash-functions that can be computed efficiently.

We give here one method of defining such a family of hash functions, assuming that the universe is $\{0, \dots, |U| - 1\}$. This method was proposed by Carter and Wegman and consists of defining

$$\mathcal{H}_{CW} := \{h_{a,b}\} \quad \text{where } h_{a,b}(k) = ((ak + b) \bmod p) \bmod M.$$

Here a, b, p, M satisfy the following:

- p is a prime-number with $p \geq |U|$ (finding this might take a large amount of time since $|U|$ is typically huge, but we only have to do it once unless the universe of keys changes),

- the table size M can be chosen arbitrarily (typically we use a power of 2) but M should be much smaller than $|U|$, hence smaller than p ,
- a is chosen *randomly* with $a \in \{1, \dots, p-1\}$,
- b is chosen *randomly* with $b \in \{0, \dots, p-1\}$.

Note that once we have chosen the hash-function (i.e., the values a and b), computing the hash-value of a given key is a simple arithmetic operation and takes $O(1)$ time. So the Carter-Wegman hash-functions satisfy the objective of being easy to compute. Crucially, as we will show below, one can also argue that they operate in some sense as well as if we had chosen randomly among all possible hash functions, and so they are an excellent middle ground between being able to compute the hash-value fast and avoiding any patterns that may exist in the data.

Analysis of randomly chosen hash functions (cs240e) . Let us first discuss further what exactly we want to achieve when picking a hash function randomly (and uniformly) from some family \mathcal{H} . Clearly we want that hash-values are uniform, i.e., $P(h(k) = i) = \frac{1}{M}$ for any key k and any slot i . But as we will argue now, this is not good enough!

To show this, we first give a convenient way to illustrate such a family \mathcal{H} as a matrix: use one column for each possible key, and one row for each hash function in \mathcal{H} . See Figure 7.13. Choosing a hash-function randomly then means choosing one row randomly, with all rows equally likely.

	keys		
\mathcal{H}_1	x	y	z
h_0	0	0	0
h_1	1	1	1

	keys		
\mathcal{H}_2	x	y	z
h_0	0	0	1
h_1	1	0	1

Figure 7.13: Examples of hash-function families for $M = 2$.

Now we illustrate why having uniform hash-values is not necessarily good enough. Consider \mathcal{H}_1 from Figure 7.13, which generalizes (for arbitrary M) to the hash functions $\{h_0, \dots, h_{M-1}\}$ where h_i maps *all* keys to the same slot. Clearly this will perform horribly in practice, because all keys create collisions. Nevertheless, when considering a randomly chosen hash function $h \in \mathcal{H}_1$ we have $P(h(k) = i) = \frac{1}{M}$, because there are M hash functions and only one of them maps key k to slot i .

What is the problem? We need not only that hash-values are uniform, we also need that hash-values are independent! This is violated for \mathcal{H}_1 : Once we know that $h(k) = i$, we know that $h(k') = i$ for *all* other keys k' , and so the hash-values are not independent.

True independence of hash-values is hard to achieve. Instead we will use a compromise and not demand independence, but only that the probability of a collisions is small. We say that a family \mathcal{H} of hash-functions *universal* if

for any two keys k, k' we have $P(h(k) = h(k')) \leq \frac{1}{M}$ where the random variable for $P(\cdot)$ is the choice of the hash-function h within \mathcal{H} .

It is very easy to see that the set of all possible hash-functions is universal (use Lemma 7.1 with $i = h(k')$). But small families of hash-functions can also be universal. Consider family \mathcal{H}_2 from Figure 7.13 where x, y, z are the only keys, and we pick each of h_0 or h_1 as hash-function with probability $\frac{1}{2}$. Then we have $P(h(x) = h(y)) = P(h \text{ was } h_1) = \frac{1}{2}$, and similarly $P(h(x) = h(z)) = \frac{1}{2}$ and $P(h(y) = h(z)) = 0 < \frac{1}{2}$. So \mathcal{H}_2 is universal. (Unfortunately it does not have uniform hash-values: $P(h(y) = 0) = 1$.)

One can show that if a family of hash-functions \mathcal{H} is universal, then hashing with chaining (and a hash-function randomly chosen from \mathcal{H}) still results in $O(1 + \alpha)$ expected run-time for all operations. (Details are left as an exercise.) Therefore, we view being universal as ‘good enough’ for a family of hash-functions.

Now we return to the Carter-Wegman family \mathcal{H}_{CW} of hash-functions. The goal is to show that this is universal. Recall that we assumed that every key belongs to $\{0, \dots, |U|-1\} \subseteq \{0, \dots, p-1\}$ (this is crucial!). Write $\mathbb{Z}_p := \{0, \dots, p-1\}$ as a convenient shortcut. To keep statements simpler, we will assume that all elements in \mathbb{Z}_p can be keys; having more keys cannot make proving universality any easier. To prove that \mathcal{H}_{CW} is universal, we will heavily use modular arithmetic; see Appendix B for some background. In particular, we assume familiarity with ‘ $a \equiv_p b$ ’ (which means that $a - b$ is a multiple of p) and ‘ $a \% p$ ’ (which means the outcome of the modulo-operation). Now consider first the functions

$$f_{a,b} : k \rightarrow (ak + b) \% p$$

i.e., $f_{a,b}$ is the hash-function $h_{a,b}$ except that we have not yet applied modulo M . Figure 7.14 shows some of the functions that we get for varying values of a, b and $p = 5$.

	keys				
	0	1	2	3	4
$f_{1,0}$	0	1	2	3	4
$f_{2,0}$	0	2	4	1	3
$f_{3,0}$	0	3	1	4	2
$f_{4,0}$	0	4	3	2	1

	keys				
	0	1	2	3	4
$f_{1,1}$	1	2	3	4	0
$f_{1,2}$	2	3	4	0	1
$f_{1,3}$	3	4	0	1	2
$f_{1,4}$	4	0	1	2	3

Figure 7.14: Some of the Carter-Wegman hash functions for $p = 5$.

Observation 7.2. $f_{a,b}(\cdot)$ defines a permutation of \mathbb{Z}_p .

Proof. By definition $f_{a,b}(k) \in \mathbb{Z}_p$, so we only have to show that no two of the p keys in \mathbb{Z}_p are mapped to the same value. Assume for contradiction that $f_{a,b}(k) = f_{a,b}(k')$ for $k \neq k'$. Observe that

$$f_{a,b}(k) - f_{a,b}(k') = ((ak + b) \% p) - ((ak' + b) \% p) \equiv_p (ak + b - ak' - b) = a(k - k').$$

Therefore $f_{a,b}(k) = f_{a,b}(k')$ implies that $a(k - k') \equiv_p 0$. Since p is a prime, this is possible only if one of a and $k - k'$ is a multiple of p . But $a \in \{1, \dots, p-1\}$ and $k - k' \in \{-p+1, \dots, p-1\}$ (since $k, k' \in \mathbb{Z}_p$), so this is possible only if $k = k'$, contradiction. \square

This observation explains a bit the idea for the Carter-Wegman hash-functions: function $f_{a,b}$ scrambles the keys into some other permutation of \mathbb{Z}_p , which hopefully destroys the patterns in the key and therefore lets the modular hash function work well. The sample of the functions in Figure 7.14 should make you suspicious of this idea, because those do not look as if they would scramble particularly well—in the functions $f_{a,0}$ we always have $f_{a,0}(0) = 0$, and in the functions $f_{0,b}$ consecutive keys map to consecutive values. However, this is an artifact of the subset of functions in \mathcal{H}_{CW} displayed here. If we vary *both* a and b , then the full list of functions shows much greater variety, and in particular, makes the hash-functions universal.

Theorem 7.1. *The set of hash functions \mathcal{H}_{CW} is a universal family.*

Proof. Fix two arbitrary keys $k \neq k'$; we need to show that $h_{a,b}(k) = h_{a,b}(k')$ with probability at most $\frac{1}{M}$. Since $h_{a,b}(k) = f_{a,b}(k) \% M$, we therefore need to bound

$$P(h_{a,b}(k) = h_{a,b}(k')) = P(\underbrace{(f_{a,b}(k)) \% M}_x = \underbrace{(f_{a,b}(k')) \% M}_{x'}) = P(x \equiv_M x')$$

We already argued that $f_{a,b}(k) \neq f_{a,b}(k')$, so $x \neq x'$, and $x, x' \in \mathbb{Z}_p$. We say that such an ordered pair (x, x') is *bad* if $x \equiv_M x'$. So the probability that we want to bound can be reformulated as follows:

$$P(h_{a,b}(k) = h_{a,b}(k')) = P(f_{a,b}(k) \text{ and } f_{a,b}(k') \text{ form a bad pair}).$$

Let us do an example of bad pairs. For $p = 5$ and $M = 2$ we would have

$$\text{bad pairs} = \{(0, 2), (0, 4), (1, 3), (2, 0), (2, 4), (3, 1), (4, 0), (4, 2)\}$$

because for each $x \in \mathbb{Z}_5 = \{0, \dots, 4\}$, we get a bad pair (x, x') if and only $x - x'$ is even, and we have written down all choices of $x' \in \{0, \dots, 4\}$ where this holds. We can bound the number of bad pairs.

Claim: There are at most $\frac{p(p-1)}{M}$ bad pairs. (For example, for $p = 5$ and $M = 2$ we had 8 bad pairs, and $\frac{p(p-1)}{M} = \frac{5 \cdot 4}{2} = 10 > 8$.)

To see why this inequality holds, fix first the left element x of a bad pair; this can be any number in $\{0, \dots, p-1\}$. The second element x' now must have distance to x that is a multiple of M . So x' belongs to one of the following numbers:

$$\begin{array}{ccccccccc} x-2M & & x-M & & x & & x+M & & x+2M \\ \cdots & | & & | & & | & & | & & | & \cdots \end{array}$$

But x' also belongs to \mathbb{Z}_p , which has p consecutive numbers. At most $\lceil p/M \rceil$ many numbers can belong to both sets; call this intersection I . For example, if $p = 5$ and $x = 2$, then $I = \{0, 2, 4\}$, and if $x = 1$ then $I = \{1, 3\}$. We have $x \in I$,

and we know $x' \neq x$, so there are at most $\lceil p/M \rceil - 1$ possibilities for x' , given x . Summarizing, therefore the number of bad pairs is at most

$$\sum_{x \in \mathbb{Z}_p} \#\{\text{possibilities for } x'\} \leq \sum_{x \in \mathbb{Z}_p} \cdot (\lceil \frac{p}{M} \rceil - 1) \leq p \cdot ((\frac{p-1}{M} + 1) - 1) = \frac{p(p-1)}{M}$$

which proves the claim.

Now we return to $P(h_{a,b}(k) = h_{a,b}(k'))$, which is the same as the probability that $f_{a,b}(k)$ and $f_{a,b}(k')$ form a bad pair. Let us reverse the view: If we fix a bad pair (x, x') , what is the probability that a and b were chosen such that $f_{a,b}(k) = x$ and $f_{a,b}(k') = x'$? This is considering the same event in a different way and so has the same probability. Or in other words

$$\begin{aligned} P(h_{a,b}(k) = h_{a,b}(k')) &= P(f_{a,b}(k) \text{ and } f_{a,b}(k') \text{ form a bad pair}) \\ &= \sum_{\text{bad pairs}(x, x')} P(f_{a,b}(k) = x \text{ and } f_{a,b}(k') = x') \end{aligned}$$

We need another small detour.

Claim: If we know that $f_{a,b}(k) = x$ and $f_{a,b}(k') = x'$, for some $x \neq x'$ and $k \neq k'$, then the values of a and b are uniquely determined from k, k', x, x' .

To see this, recall that $x - x' = f_{a,b}(k) - f_{a,b}(k') \equiv_p a(k - k')$. Since p is a prime, the set \mathbb{Z}_p is a field with respect to operators $+$ and \cdot . Therefore the inverse $(k - k')^{-1}$ exists within this field and $a \equiv_p (k - k')^{-1}(x - x')$. Since $a \in \mathbb{Z}_p$, therefore $a = (k - k')^{-1}(x - x') \% p$ is uniquely determined. As for b , we have $x = (ak + b) \% p$ and therefore $b = (x - ak) \% p$ is uniquely determined since a is. This proves the claim.

A small example may be helpful. Consider the case where $p = 5$, $f_{a,b}(1) = 3$ and $f_{a,b}(4) = 2$. The equation tells us that $x - x' = 3 - 2 \equiv_5 a(1 - 4) = (-3)a$. Since $-3 \equiv_5 2$, therefore $1 \equiv_5 2a$. Since $2 \cdot 3 = 6 \equiv_5 1$, we have $2^{-1} = 3$ (in the field \mathbb{Z}_5), and therefore $a = 1 \cdot 2^{-1} = 3$. From this we easily get $b = (x - ak) \% 5 = 3 - 3 \cdot 1 = 0$, and indeed one verifies that $f_{3,0}$ from Figure 7.14 gives this mapping.

Now returning to the probability that we want to bound, we have

$$\begin{aligned} P(h_{a,b}(k) = h_{a,b}(k')) &= \sum_{\text{bad pairs}(x, x')} P(f_{a,b}(k) = x \text{ and } f_{a,b}(k') = x') \\ &= \sum_{\text{bad pairs}(x, x')} P(a = (k - k')^{-1}(x - x') \% p \text{ and } b = (x - ak) \% p) \\ &= \sum_{\text{bad pairs}(x, x')} P(a = (k - k')^{-1}(x - x') \% p) \cdot P(b = (x - ak) \% p) \end{aligned}$$

where the last equality holds since b is chosen independently from a .

Note that $(k-k')^{-1}(x-x')\%p$ is not 0, because $(k-k')^{-1}$ is not 0, $x-x'$ is not 0 by $x \neq x'$, and neither of them is a multiple of the prime p since $(k-k')^{-1}, x, x' \in \mathbb{Z}_p$. So $(k-k')^{-1}(x-x')\%p$ is in $\{1, \dots, p-1\}$, and a was chosen randomly and uniformly in this set. Therefore $P(a = (k-k')^{-1}(x-x')\%p) = \frac{1}{p-1}$, and similarly one argues $P(b = (x-ak)\%p) = \frac{1}{p}$. So we can simplify further as

$$P(h_{a,b}(k) = h_{a,b}(k')) = \sum_{\text{bad pairs}(x,x')} \frac{1}{p-1} \cdot \frac{1}{p} = \#\{\text{bad pairs}\} \frac{1}{p(p-1)} \leq \frac{p(p-1)}{M} \frac{1}{p(p-1)} = \frac{1}{M}$$

as desired. □

7.4.2 Hash functions for other data

We have so far assumed that the keys are integers. But the concept of hashing can be applied to *any* kind of key, as long as there is some way to map the keys to slots. Often this is done with a detour: first map the key to an integer, and then map that integer to a slot with one of the hash-functions discussed earlier.

We demonstrate this on the example of words. (Recall that very large integers can also be interpreted as words, by using individual digits as characters.) There is a natural way to map ASCII-strings to a base-128 number, by mapping each character to the corresponding ASCII number. For example

$$A \cdot P \cdot P \cdot L \cdot E \rightarrow (65, 80, 80, 76, 69)$$

(we write the base-128 digits separated by commas for ease of legibility). Likewise any string over any alphabet Σ can be mapped to a base- $|\Sigma|$ -number.

In turn, any base- R number is naturally converted to a base-10 number via

$$A \cdot P \cdot P \cdot L \cdot E \rightarrow (65, 80, 80, 76, 69) \rightarrow \underbrace{65R^4 + 80R^3 + 80R^2 + 76R + 69}_{f(\text{APPLE})}$$

This mapping from a string w to an integer is sometimes called the *flattening function* and denoted $f(\cdot)$ (or $f_R(\cdot)$ if the radix R is not clear). If we use $R = |\Sigma|$, then the number $f(w)$ uniquely identifies the string w .

However, for hashing-purposes uniqueness is not needed since we never need to go retrieve the key from the hash-value. As such, it is actually a better idea to use a radix R that is different from $|\Sigma|$. Then the digits in $f(w)$ do not directly correspond to the characters in w , achieving a scrambling of w and hence (hopefully) destroying any pattern that might have existed. Experiments have been performed to see which radix R achieves the best distribution of hash-values; for ASCII some of the best ones are $R = 17, 31, 63, 127, 129$. (These have another advantage discussed below.)

The number $f(w)$ tends to be very large; too large to use as slot-index in a hash-table. Therefore we combine flattening of the word with the application of a hash-functions; for example we could use the modular hash-function. In summary, to use hashing for words, a commonly used hash-function is

$$h(w) = f(w) \bmod M.$$

If we compute $h(w)$ as described above by first computing $f(w)$ and then applying the modular hash function, this would likely lead to *overflow* (i.e., numbers that are bigger than what can be held in one memory cell). Observe that we need to compute $R^{|w|}$, which even for fairly short words (say $R = 127$ and $|w| = 5$) uses more than 32 bits. Luckily enough, this can be avoided by observing that ‘ $\bmod M$ ’ is distributive, i.e., we can apply it even inside the arithmetic formula used to compute $f(w)$ and hence reduce numbers to smaller values. A second trick to reduce the run-time is to use *Horner’s rule* which restates a polynomial as repeated multiplication. In particular, $h(APPLE)$ therefore is

$$\left(\left(\left(\left(\left((65R+80) \bmod M \right) R+80 \right) \bmod M \right) R+76 \right) \bmod M \right) R+69 \right) \bmod M$$

4800 The general algorithm to compute $h(w)$ for a word w is given in Algorithm 7.5. Clearly this
 4801 computes $h(w)$ in $O(|w|)$ time, and all intermediate values are no bigger than $R \cdot M + |\Sigma|$, which
 4802 should be manageable. Also note that if we choose R among the numbers listed above, and
 4803 M is a power of 2, then this algorithm can be implemented with only additions and bit-shifts.
 4804 Namely, assume that if $R = 2^i \pm 1$ for some integer i , then $h * R = h \lll i \pm h$, where ‘ $\lll i$ ’ is the
 4805 operation of shifting the bit-string leftward by i bits. Also, if $M = 2^m$ then $h \bmod M$ consists
 4806 of the rightmost m bits of h . So under these restrictions, computing *word-hash-value* is very
 4807 efficient.

Algorithm 7.5: *word-hash-value*(w, R, M)

Input : word w , radix R , table-size M

```

1  $h \leftarrow 0$ 
2 for ( $i = 0 \dots w.length-1$ ) do
3    $h \leftarrow h * R$ 
4    $h \leftarrow h + w[i]$ 
5    $h \leftarrow h \bmod M$ 
```

4808 7.5 Take-home messages

- 4809 • Hashing is a powerful and extremely popular technique, since it is easy to implement using
 4810 only arrays and tends to be efficient in practice.

- Under stringent assumptions on the hashing functions or probe sequences, the amortized expected run-time for all operations is in $O(1)$ if the load factor is kept sufficiently small.
- For any fixed method to choose a hash-function, there exist keys on which hashing will perform very badly. To guarantee a good performance while being efficient, one should choose a hash-function randomly from a universal set of hash functions.

7.6 Historical remarks

Hashing methods were introduced in the 1950s, though curiously the word ‘hashing’ was not used in writing until the late 1960s. This is probably due to a somewhat negative connotation for the word ‘to hash’, which means to break into small and un-recognizable pieces. (Think ‘to make a hash of it’ and ‘hash browns’.) For our purpose here, breaking the keys (which might follow some patterns) into un-recognizable pieces is a good thing!

According to Knuth’s textbook, the idea of hashing appeared in an internal IBM memorandum in 1953, and independently at other groups at IBM around the same time. The first publicly available article about hashing apparently was by Dumey in 1956 [Dum56]; he discusses the issue of collisions and proposes some ideas for resolving them, but provides no analysis. Williams proposed in 1959 to handle collisions via an ‘overflow table’ [Wil59]; from this the idea of hashing with chaining evolved over the next few years. The concept of open addressing (and uniform probing) was introduced by Peterson in 1957 [Pet57]; he also proved Lemma 7.6. As listed in Table 7.12, linear probing and double hashing were analyzed somewhat later [KW66, LM93]. Knuth’s textbook [Knu98, p.536ff] gives two other ways of analyzing linear probing, possibly simpler than [KW66] but giving slightly weaker bounds.

Cuckoo hashing, in contrast, has been developed comparatively recently by Pugh and Rodler ([PR04]; the first publication at a conference was in 2001). It has proved very popular, especially since all the potentially ‘bad’ behavior is pushed onto *insert*, while the typically-more-frequent *search* is always fast. In 2020 it was awarded the test-of-time award by the European Symposium of Algorithms (where it was first published).

Even the earliest papers on hashing assume (for their analysis) that hash values are assigned in some random manner, but the idea of picking from a specifically defined set of easily-computable functions came later. Family \mathcal{H}_{CW} was (as the name suggests) introduced by Carter and Wegman ([CW79], the conference-version appeared in 1977). There are many other ways of defining universal hash functions, and criteria other than ‘universal’ have also been studied. A relatively recent article by Thorup [Tho15] gives a good overview.

Chapter 8

Range searches

Contents

8.1	Introduction	230
8.1.1	Range searches	230
8.1.2	Higher-dimensional data	231
8.1.3	Two simple (but deficient) ideas	232
8.2	Quad trees	233
8.2.1	Dictionary operations	235
8.2.2	Quad-tree height	236
8.2.3	Range search in a quad-tree	238
8.2.4	Quad-tree extensions and discussion	239
8.3	kd-trees	241
8.3.1	Dictionary operations	242
8.3.2	Range search in a kd-trees	243
8.3.3	kd-trees in higher dimensions	247
8.4	Range trees	247
8.4.1	Dictionary operations	247
8.4.2	Range search in a binary search tree	249
8.4.3	Range search in a range-tree	252
8.4.4	Range trees in a higher dimension	254
8.5	3-sided range searches (cs240e)	254
8.5.1	Idea 1: Range trees, except use heaps.	254
8.5.2	Idea 2: Cartesian trees	256
8.5.3	Idea 3: Priority search tree	257
8.6	Take-home messages	258
8.7	Historical remarks	259

8.1 Introduction

While ADT Dictionary is a very useful data type when storing items where a fixed key is known, the more common method to store large amounts of data is in a data base. This typically comes with a query language that allows us much more complex accesses to the data. We will not give full details here of how one implements such a data base (take cs448 if you are interested), but focus here on two especially useful aspects of data bases: permitting range queries and storing multi-dimensional data.

8.1.1 Range searches

Let us first consider the following new operations that we could use to search among keys.

rangeSearch(x_1, x_2): given two keys $x_1 \leq x_2$, find *all* items with keys that are between x_1 and x_2 .

We are deliberately a bit vague in saying whether “between x_1 and x_2 ” includes or excludes the boundary x_1/x_2 . For many of the range search data structures listed below, items that could be on the boundary will be individually tested for being in the range, and as such it can be left up to the user whether the range is open or closed, i.e., excludes or includes its boundary.

Note that *rangeSearch* is a generalization of *search*, because we can search for any one key x by doing *rangeSearch*(x, x) (assuming we can support a version that includes the boundary).

Range search in some dictionary realizations. We first discuss how to implement a range search in some of the dictionary realizations that we have seen. Let us start with a sorted array A . Here the algorithm for a range search is very simple: Search for x_1 (typically with binary search or interpolation search); this should return the index i_1 with $A[i_1] = x_1$ if there is one, and otherwise it should return the index near where x_1 should be. Similarly search for the index i_2 where x_2 is or should be. For *rangeSearch*(14, 45) in the array below, this would give the following:

i_1					i_2				
5	10	11	17	19	33	45	51	55	59
\uparrow_{x_1}					\uparrow_{x_2}				

Now report all items in $A[i_1+1 \dots i_2-1]$, and report the “boundary”-items $A[i_1]$ and $A[i_2]$ if they fall in the range.

The run-time for this algorithm is $O(\log n + s)$, where “ $\log n$ ” is needed for the binary searches to find i_1, i_2 , and s is the number of items that are in the range, i.e., the size of the output. This is sometimes called *output-sensitive* analysis: the run-time depends on the size of the answer. Note that a run-time of $\Omega(s)$ is required simply to print all the items of the output, so the run-time to find the items in the range cannot be smaller. We could have $s = n$, so if we did

not keep s as a separate parameter, we could never hope to have a worst-case run-time better than $\Theta(n)$. On the other hand, sometimes s is very small (or even $s = 0$), and it would be nice to have an algorithm that can determine quickly (i.e., in $o(n)$ time) whether there is no item in the output. This is why it makes sense to keep s as a separate parameter.

We have given an algorithm for range search in a sorted array that has run-time $O(\log n + s)$. With the same basic technique (find x_1 , find x_2 , report all items in-between) we can also do a range search in $O(\text{search} + s)$ time in skip lists, binary search trees and tries. We will see this in detail for binary search trees below; for all others this is left as exercise to the reader.

On the other hand, range search in an unordered array requires $\Omega(n)$ time, because if there is any one element in the array that we do not look at, then this element may be in the range or not, and so the algorithm may give the incorrect answer. In particular, there are no tools to do range search in hash tables in $o(n)$ time.¹

8.1.2 Higher-dimensional data

For ADT Dictionary, we strictly separated between the key (which is used for searching) and the value (which is mostly ignored). However, often we may be interested in searching among all aspects of an item. As such, we now consider items to have multiple (and equally important) *attributes* (also known as *aspects* or *coordinates*), and we would like to be able to search by all of them. For most of this chapter, we assume that the items have exactly two attributes and that these attributes are numbers. (With this, each item becomes a point in 2-dimensional space.) One could easily handle other kinds of attributes such as words, either by mapping them to numbers (as we did in hashing) or by adapting the data structures that we will develop to use tries rather than binary search trees.

One simple idea to handle multi-dimensional data is to use multiple dictionaries, one for each attribute. With this we could search by each attribute. However, this does not help us with queries that consider all attributes at once, which we define now.

Orthogonal range queries. Consider the example where you are looking for a flight. You must leave between 9am and noon, and you can't afford to pay more than \$500. But you've had really bad experience with budget airlines, so you want to avoid the super-cheap flights as well. In consequence, you want to perform a *two-dimensional range search*: Among all flights, pick only the ones where the price is between 300 and 500, and the departure time is between 9 and 12.

Formally, an (*orthogonal*) *range search* (for multi-dimensional data) is to specify a range (interval) for each aspect, and find all the items whose aspects fall within the given ranges.

It is worth mentioning here that orthogonal range searches only scratch the surface of the topic of range searches. Often when searching for items you really want a tradeoff between various aspects—for example when searching for a laptop, you are probably willing to pay

¹There are some specialized hash-functions that are *locality-preserving*, i.e., that keep relative distances between keys and where hence some range search techniques are possible, but this is beyond the scope of cs240.

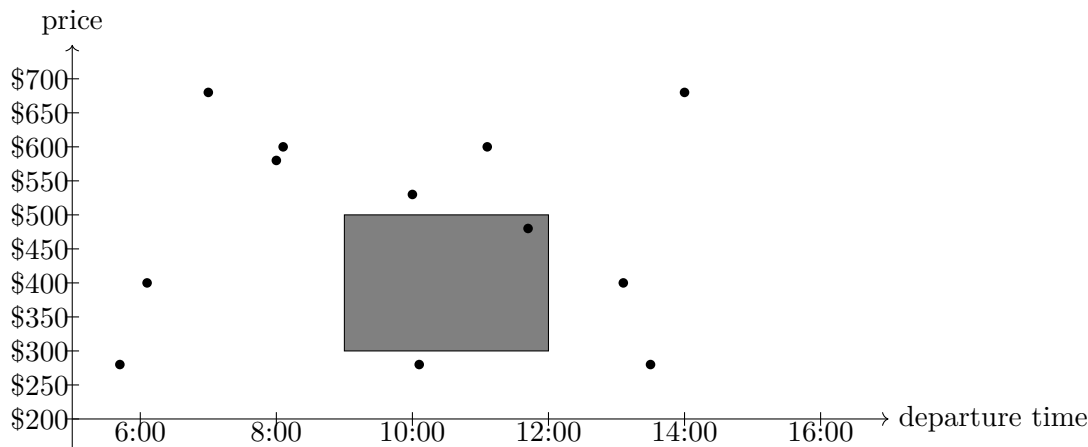


Figure 8.1: A two-dimensional orthogonal range search.

4942 a slightly higher price if the processor speed is higher, so the range search should really be
 4943 bounded by a linear function of the processor speed and the price. These kinds of searches are
 4944 called *non-orthogonal range searches*. There are tools for these, but they are beyond the scope
 4945 of cs240 (see a computational geometry course such as cs763).

4946 **General position assumption.** For the rest of this chapter, we will develop data struc-
 4947 tures that permit efficient orthogonal range searches in d -dimensional data (typically $d = 2$).
 4948 Throughout this chapter, we assume that the items are in *general position*. For orthogonal range
 4949 search, this means the following:

4950 No two x -coordinates are the same, and no two y -coordinates are the same.

4951 This assumption is not at all realistic, but will greatly simplify the presentations of some of
 4952 the data structures we will see later. It is possible to generalize the data structures to handle
 4953 arbitrary points via a technique called *symbolic perturbation*, which we will not explain in more
 4954 detail here.

4955 8.1.3 Two simple (but deficient) ideas

4956 We give here first two simple approaches to storing 2-dimensional points and answering range
 4957 searches, and argue that they do not work well.

4958 The first approach would be to map each point (x, y) to an integer (e.g. by combining the
 4959 aspects similar to how we did it for hashing of words), and then store these integers. The

problem is that this makes it impossible to do a range search efficiently, since we have lost the information about individual aspects. The only way to do range searches is to inspect *all* items, which is too slow.

A second idea would be to store a separate dictionary for each aspect, e.g. to store one array A_x with all points sorted by x -coordinate and another array A_y with all points sorted by y -coordinate. This is somewhat better, because we can at least eliminate some points quickly. To do a range search for query-rectangle $[x', x''] \times [y', y'']$, we would do a range search for $[x', x'']$ in A_x (say this returns set S_x), a range search for $[y', y'']$ in A_y (say this returns set S_y), and then compute the intersection $S_x \cap S_y$. While this approach might work decently in practice, it can be much too slow. In particular, we could have $|S_x| \approx n/2 \approx |S_y|$ and yet $S_x \cap S_y = \emptyset$, which means that computing S_x and S_y took $\Theta(n)$ time, but the output-size is $s = 0$ and so the run-time was much larger than it needed to be.

What we are aiming for in the remaining sections is to design data structures *specifically* for points and range searches on them. We will see three examples—quad-trees, kd-trees and range-trees—which are increasingly more complicated but also have increasingly better run-time for the range search.

8.2 Quad trees

The first data structure that we give here is quite bad in theory, but it also is very simple, and works quite well in practice, especially if points are evenly distributed within the plane. For this reason, it is very popular in computer graphics applications.

We have n points $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ in the plane. To store them, we first compute a *bounding box* R , which is a square that contains all points in S and that has the form $[0, 2^k) \times [0, 2^k)$ for some integer $k \geq 0$. (We silently make the assumption that all points in S have non-negative coordinates. This is usually true for graphics applications. For applications where it is false, we could translate all the points and the query-rectangles to make it true.) Note that such a bounding box R can be found in linear time by computing the maximum coordinates.

The *quad-tree* (with respect to point set S and bounding box R) is now defined recursively as follows:

- Define one node r that becomes the root of the quad-tree. Associate region R with r .
- If R contains 0 or 1 points, then the quad-tree consists only of node r , and r stores the point in R (if any).
- Otherwise, we split points and build the quad-tree recursively as follows:
 - Partition R into four equal sub-squares (*quadrants*) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$. Also split S into the four point-sets $S_{NE}, S_{NW}, S_{SW}, S_{SE}$ that fall into the four quadrants. This step motivates the choice of R to have side-length 2^k . Since we split R into four equal quadrants, each smaller square has side-length that is half the side-length of R . Since the side-length is a power of 2, we do not have to worry about rounding

issues. Furthermore, division by 2 is simply a bit-shift in the computer; this makes the computation of the four point-sets much faster and feasible to execute on hardware..

- Recursively compute the four quad-trees $T_{NE}, T_{NW}, T_{SW}, T_{SE}$, where T_i is the quad-tree with respect to S_i and R_i .
- The quad-tree of S is now obtained by setting the four obtained trees to be the four subtrees of root r .

See Figure 8.2 for an example of a point-set and its associated quad-tree. There are a few conventions that we must clarify:

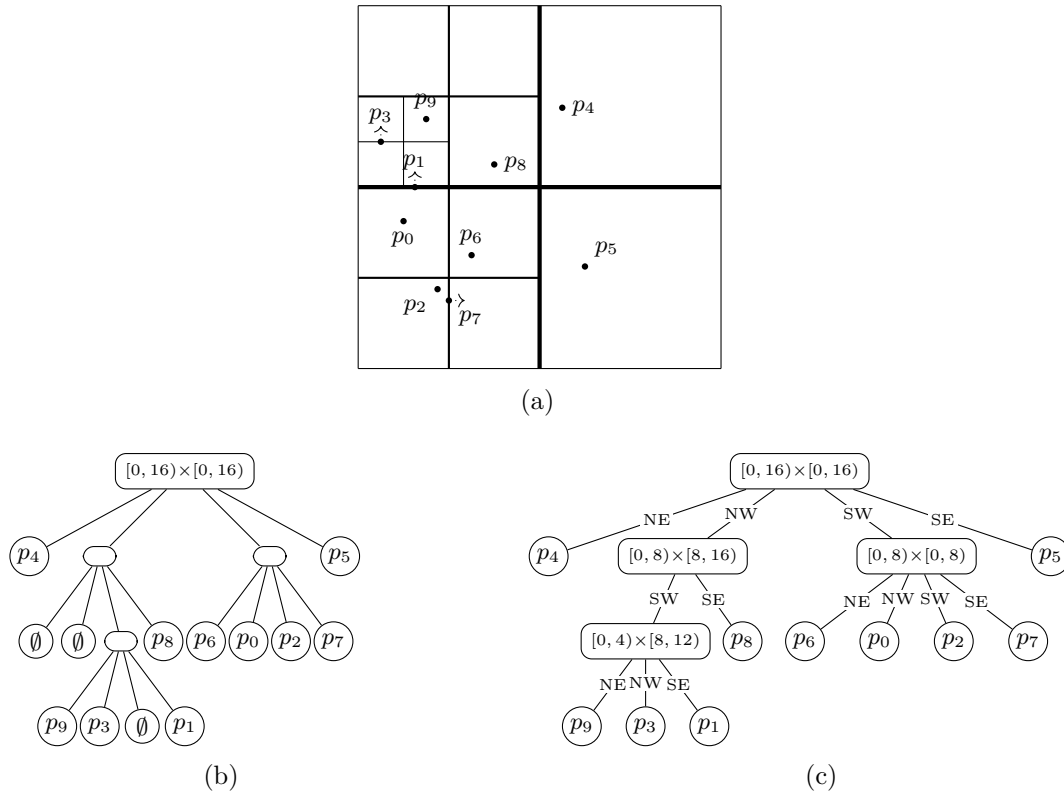


Figure 8.2: (a) A point set and how it is divided. We add small arrows at some points to indicate to which region they belong. (b) The quad-tree as stored by the computer. (c) A version that omits empty subtrees and shows the associated regions.

- As always for ADT Dictionary, we assume that no two keys are identical, i.e., no two points that we store are exactly the same. Therefore, any two points *will* be in separate regions eventually.
- What do we do with points that are on the split-lines? We will follow here the convention that points on split lines *always* belong to the right/top side (we will see below what motivated this). In other words, the quadrants R_{NE}, R_{NW}, R_{SW} and R_{SE} are open at the

top and right side, and closed on the bottom and left side. (Note that we assumed the same for the bounding box R for consistency.)²

- What is the order of subtrees? We use the convention that the subtrees of a node are always listed in order NE, NW, SW, SE , or in other words, they follow the order quadrant 1, 2, 3, 4. As long as this is followed, there is no need to label the edges to subtrees with the part that they represent. If, however, one wants to “clean up” the picture by omitting subtrees that store no points, then labeling edges becomes necessary.
- In our example, we showed with each node also the region that it is associated with. This is useful for humans to keep track of where we are, but is not needed for a computer. All the algorithms to come will parse a quad-tree top-down, i.e., starting from the root where we know the bounding box R . It is very quick to re-compute the region R_i of the child i that we go to (for $i \in \{NE, NW, SW, SE\}$), since this is only a division by 2, i.e., a bit-shift. Therefore, while there is such a region associated with each node, there is no need to store it explicitly at the node.

8.2.1 Dictionary operations

The definition of a quad-tree can at the same time also be seen as an algorithm to build it: While there is more than one point, split the region and the points, recurse, and put the subtrees together. Clearly the splitting of the points can be done in $O(n)$ time. The time to build the quad-tree is therefore $O(nh)$, where h is the height of the quad-tree.

Let us also look at how to implement the standard operations for ADT Dictionary in a quad-tree. Operation *search* is very easy: The information at each node of the quad-tree is enough to identify where a point is. Formally, when searching for point p , if the quad-tree has only one node then it either stores p or p is not in the quad-tree. Otherwise, the root splits the bounding box into four quadrants; find the quadrant that contains p and recurse in the appropriate subtree.

Operation *insert* is also not difficult. First check whether the new point q is in the bounding box. If it is not, then repeatedly double the size of the bounding-box and update the quad-tree correspondingly until the point is in the bounding box. Now, search as above to find the leaf ℓ of the quad-tree that should contain q . Then for as long as ℓ stores two points, apply the splitting operation at ℓ , and recurse in the newly created leaf whose region contains p . See for example Figure 8.3; the quad-tree here was obtained from the one in Figure 8.2 by inserting point p_{10} .

To *delete* a point p , first search for the leaf ℓ that contains p and remove p from it. If the parent p of ℓ now has only one point left in its subtree, then there is no longer a need to split at parent p ; we can delete ℓ and its siblings, store the point at parent p , and recurse at the parent of p .

²This is only a convention, and while you should follow it for cs240 assignments, you should keep in mind that in real life one could use a different convention, or even store points on the split-lines with the node r , as long as the search-routines are adjusted suitably.

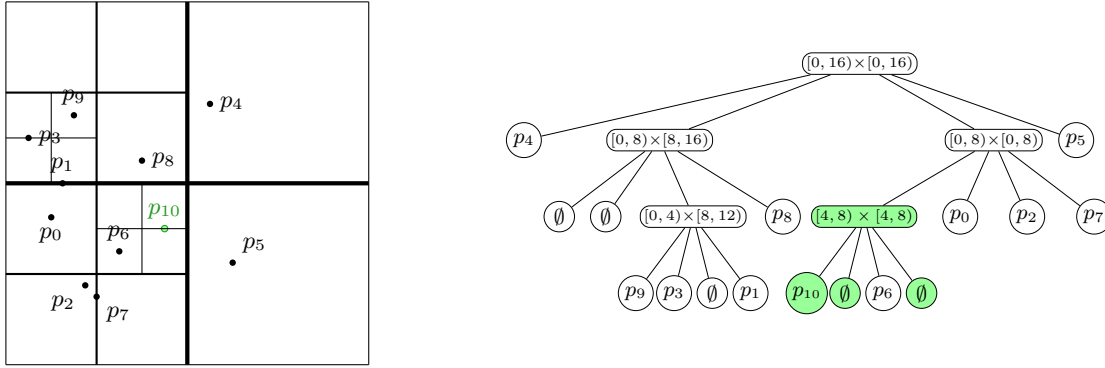
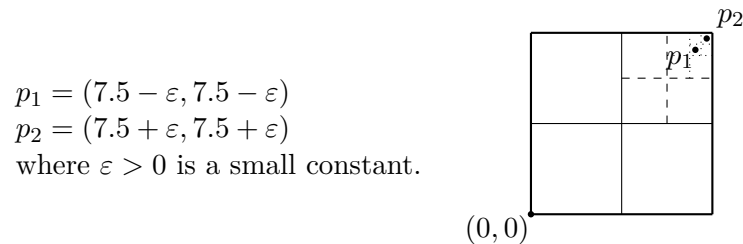


Figure 8.3: Inserting a point in a quad-tree.

8.2.2 Quad-tree height

All these operations take time $O(h)$, where h is the maximum height that the quad-tree had before or after the operation. This raises the crucial question: what is the height of a quad-tree in the worst case? Unfortunately, the answer is: bad. There is no bound on the height that depends only on n . (This is even worse news than we had for binary search trees, where the height may be big, but at least it is never more than $n - 1$.) To see that this is the case, consider the points in Figure 8.4. If we choose $\varepsilon > 0$ very small, then p_1 and p_2 are very close together, while the bounding box is big. (We have defined our bounding box to always have one corner at $(0, 0)$, so this forces the size. But even without the restriction we could force a large size with a third point at $(0, 0)$.) Therefore it will take many repeated splittings before p_1 and p_2 are in different regions, resulting in a large height of the quad-tree.

Figure 8.4: A set of three points for which the quad-tree height is arbitrarily large (depending on how we choose ε).

However, this can only happen if we allow arbitrarily small ε , or in other words, some points are very close together while others are quite far apart. In general, we can bound the height if we consider as parameter the *spread-factor* of the point set S , which is (roughly speaking) the ratio between the largest and the smallest distance of points. (Recall that no two points

are the same.) We will prove here only one special case of this, which also shows an interesting connection between quad-trees and tries.

Lemma 8.1. (cs240e) *Let S be a set of distinct points whose coordinates are non-negative integers in $\{0, \dots, U-1\}$. Then the height of the quad-tree is at most $\lceil \log U \rceil$.*

Proof. (cs240e) Let us do a small detour and consider a quad-tree that stores 1-dimensional “points” (i.e., numbers x_0, \dots, x_{n-1} in $\{0, \dots, U-1\}$). See also Figure 8.5. We can interpret each x_i as a bit-string of length ℓ , where $\ell = \lceil \log U \rceil$. The 1-dimensional quad-tree of these points would use bounding-“box” $[0, 2^\ell]$. At the root, it splits this box into two parts— $[0, 2^{\ell-1})$ and $[2^{\ell-1}, 2^\ell]$ —and split the points by whether they fall into the left or the right part. But notice that $x_i \in [0, 2^{\ell-1})$ if and only if the leading bit of x_i (in its base-2 representation) is a 0. Therefore, the split into left and right part is exactly the same as the split that would happen to the bit-strings of x_0, \dots, x_{n-1} if we stored them in a trie! The argument repeats in the quad-trees that are built farther down: the next split is by the next bit of each bit-string. Therefore, 1-dimensional quad-trees for integers are really the same as pruned tries for bit-strings, and their height is at most the length of the bit-string, which is $\ell = \lceil \log U \rceil$.

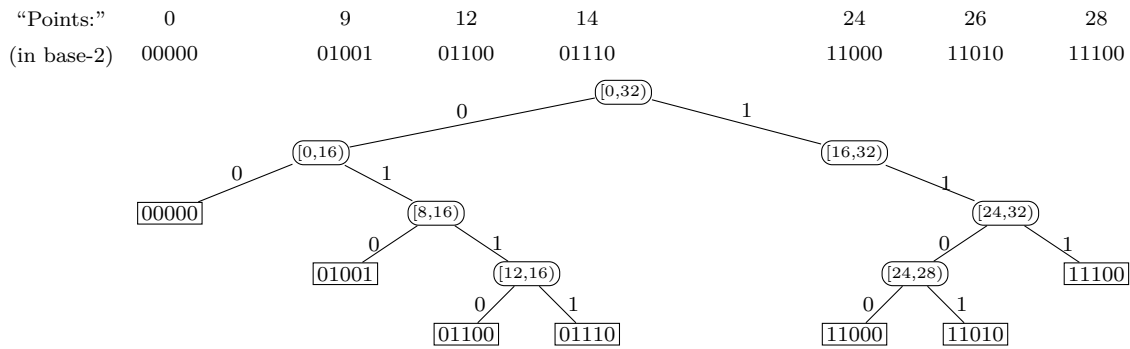


Figure 8.5: The quad-tree of 1d integral points is a trie.

For 2-dimensional quad-trees where points have coordinates in $\{0, \dots, U-1\}$, the argument is very similar, except that now the quad-tree splits 4-ways at each node, depending on whether the first bits of the x -coordinate and y -coordinate are 0 and 1, respectively. Again the maximum height is the length of these bit-strings, which is $\lceil \log U \rceil$. \square

Recall that the expected search-time in a pruned trie with n bit-strings is $O(\log n)$ (Lemma 6.5). With almost verbatim the same proof (but noticing that two points have the same common bit on *both* coordinates with probability $\frac{1}{4}$), one can hence also show:

Lemma 8.2. (cs240e) *If a quad-tree stores n distinct points with coordinates that were uniformly and independently chosen from $[0, 1)$, then the expected search-time is $O(\log_4 n)$.*

8.2.3 Range search in a quad-tree

Now we turn to the operation that motivated us to study quad-trees: an orthogonal range search. Say we are looking for all points that lie within the *query-rectangle* $A = [x', x''] \times [y', y'']$. (To keep with the convention for quad-trees, we assume that the query-rectangle is also open on the right/top side; the code can be adjusted to work for other situations.)

The idea for doing a range search is very easy: If the quad-tree stores just one point p , then simply test whether $p \in A$ and return p in case of a positive answer. Otherwise, the quad-tree has four sub-trees for four quadrants; recurse in all those subtrees whose associated region intersects query-rectangle A .

We will describe the code in a slightly more complicated way, because this will be a convenient warm-up for algorithms that are to come later. In particular, we define the following classification of a node v of the quad-tree. Let R_v be the region that is associated with v . Then we use the following classification:

- v is called an *inside node* (we show it colored *green*) if $R_v \subseteq A$. Notice in particular that *all* points in the subtree rooted at v then are inside A , and we simply report all of them.
- v is called an *outside node* (we show it colored *red*) if $R_v \cap A = \emptyset$. Notice in particular that *none* of the points in the subtree rooted at v are inside A , so we can stop recursing from v .
- v is called a *boundary node* (we show it colored *blue*) if neither of the above two holds, which means that R_v intersects A , but is not entirely inside A . In this case, we cannot say yet which of the points in the subtree rooted at v are inside A , and we must recurse in the children of v to determine this.

Algorithm 8.1 gives the code. We assume here that each node stores the associated region, an alternative (which is more complicated-looking but more space-efficient and likely faster in practice) would be to pass the bounding box as a parameter, and to recompute the region that is associated with it from the passed bounding box and to pass that along into the recursion.

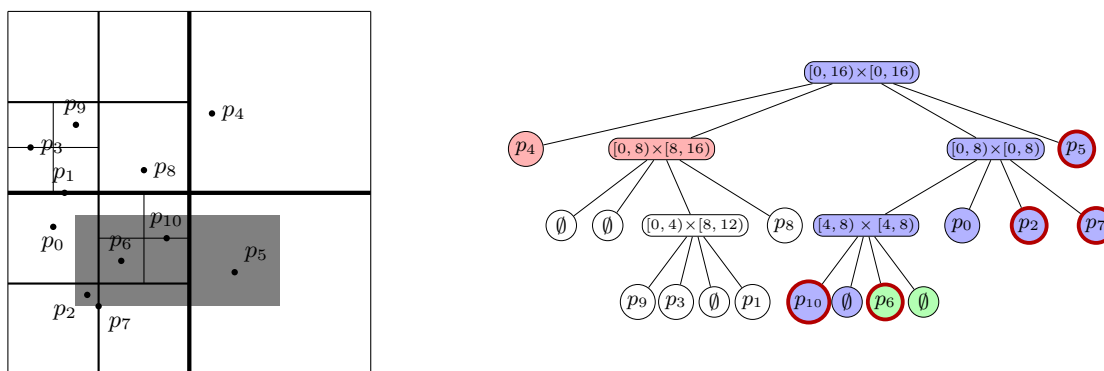


Figure 8.6: Range search in a quad-tree.

What is the run-time of this algorithm? First note that it is quite easy to test in constant

Algorithm 8.1: *quadTree::rangeSearch*($r \leftarrow \text{root}, A$)

```

Input : Node  $r$  of a quad-tree. Query-rectangle  $A$ 
1  $R \leftarrow$  square associated with node  $r$ 
2 if  $R \subseteq A$  then                                     // inside node
3   | report all points in subtree at  $r$  and return
4 else if  $R \cap A$  is empty then                         // outside node
5   | return
6 else                                                  // boundary node
7   | if  $r$  is a leaf then
8     | if  $r$  stores a point  $p$  and  $p \in A$  then
9       |   return  $p$ 
10    | else
11    |   return
12  | else
13    | foreach child  $v$  of  $r$  do
14    |    $\text{quadTree::rangeSearch}(v, A)$ 

```

time whether $R \subseteq A$: Each of the four half-planes that define R need to be subsets of the corresponding half-planes that define A . Likewise it is easy to test whether $R \cap A = \emptyset$, because then one of the four half-planes that define R must contain A on the other side. So the run-time of the algorithm is determined by how many nodes we “visit”, i.e., how many nodes of the quad-tree are the parameter v in some recursive call to *quadTree::rangeSearch*. Unfortunately, bad news. If we are unlucky, then this algorithm visits *all* nodes of a quad-tree and does not return a single point in the range as output. (Try to create such a point set!) Therefore, we cannot give a bound on the run-time better than the size of the quad-tree, which is $O(nh)$.

8.2.4 Quad-tree extensions and discussion

As promised earlier, let us briefly look at how to generalize quad-trees to higher dimensions. The 2d structure was called a *quad*-tree because at any step, we split the associated region into quadrants and recurse in the quadrants. In 3d, quadrants become *octants* (because in \mathbb{R}^3 a cube gets split into 8 parts that are cubes of half the side-length), and the structure is therefore called an *oct-tree*. Every node in an oct-tree has 8 children, corresponding to the octants. All the operations work in almost exactly the same way, with the only change that we now have eight children among which we choose the appropriate one.

One could also go to higher dimensions. In dimension d , a cube gets split into 2^d cubes of half the side-length. So the corresponding partition-tree would have 2^d children per interior node. This gets very big very fast, and quad-trees are rarely used beyond dimension $d = 3$.

Summarizing quad-trees, they are a very simple data structure to store points and perform range searches. Similar as for tries, the performance depends on the distribution of the input, but if coordinates are non-negative integers then the time for search, insert and delete is proportional to the number k of bits needed for the integers. There are no good bounds on the run-time for range searches. In practice, however, quad-trees are very fast. This holds because the pathological examples (both for the large height and for the inefficient range search) occur if there is lots of “whitespace”, i.e., some points are very far apart while other points are very close together. In realistic applications, points tend to be distributed more evenly, with some regions more dense than others but also large patches of fairly uniform density. Here, the range search will visit many nodes only if there are many points to report from these nodes. Therefore, the simplicity of quad-trees (and the fact that the arithmetic operations are limited to divisions by 2 and therefore implementable in hardware) make quad-trees a decent choice for some applications even though its worst-case run-time is bad.

One should mention that there are numerous variants of quad-trees. Apart from changing the convention of where points on the split-lines go, one could also do similar variations as we did for tries: We could store points directly at an inner node (especially if a point lies on the intersection of the two split-lines), or we could *compress* the quad-tree by omitting those inner nodes where all points fall into one of the four quadrants.

Last but not least, one should mention that a variant of quad-trees is very useful for storing pixelated images. We can view the pixels of an image as an $m \times n$ -matrix M , where each entry of M is a color. We can view this as a set of points (i, j) (one per entry in the matrix) where each point has a color. To store such an image (say for $m = n$ a power of 2) we use a quad-tree where we stop splitting as soon as all points in the subtree have the same color. See Figure 8.7 for an example. For images that have large patches of identical colors, this gives a very efficient method of describing the picture without having to list all the pixels.

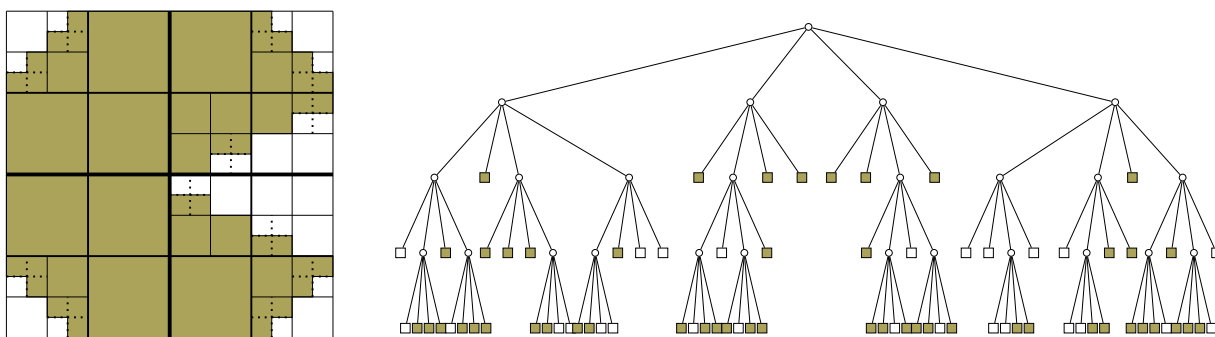


Figure 8.7: Pac-man represented using a quad-tree.

8.3 kd-trees

A quad-tree could be summarized as follows: Split the overall region (a square R) into four subregions (the quadrants), based on the dimensions of R . Our next data structure also splits the overall region (it will now be a rectangle R) into subregions (now we use only two). The main difference is that the decision of how to split will now be made based on where the *points* are, rather than on the dimensions of R . Specifically we divide R as follows:

- Split region R with a horizontal or vertical line (the *splitting line*) into two rectangles.
- The split is done such that approximately half of the points are in each of the rectangles.
- We alternate between vertical and horizontal splitting lines.

Similarly as for quad-trees, we store these splits in a tree (this time a binary tree since we split R into two regions), where the root stores the information needed to determine the splitting line, and the actual points are stored at the leaves. This structure is called a *kd-tree*, see Figure 8.8 for an example.

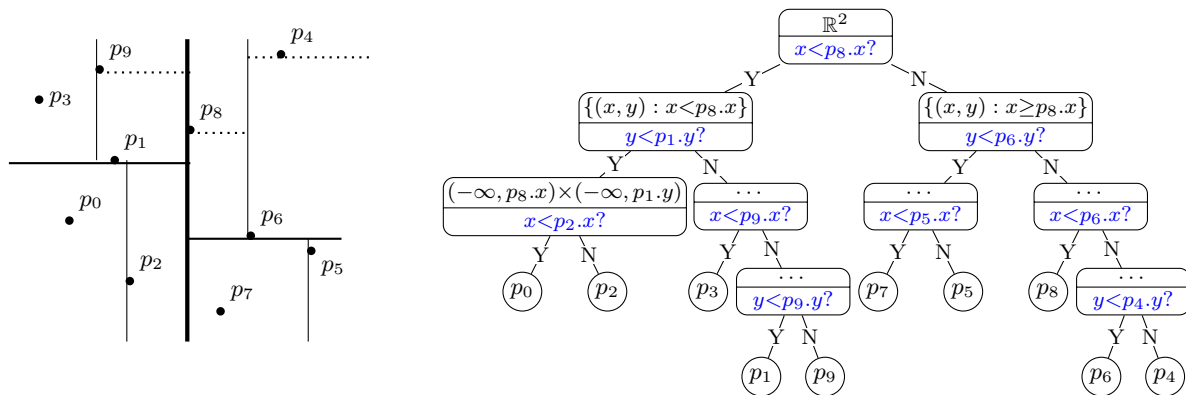


Figure 8.8: A set of points and the corresponding kd-tree. Not all associated regions are shown.

A few remarks/conventions are in order:

- The name “kd-tree” was chosen because ‘d’ stands for ‘dimensional’, while k indicates the actual dimension that is applied. So technically we should call the trees 2d-trees, but we will use kd-trees because the same concept applies with very minor changes for higher dimensions.
- We split by the median of the coordinates. Thus, on even levels the vertical splitting line has as x -coordinate the median of the x -coordinates of the points; on odd level the horizontal splitting line has as y -coordinate the median of the y -coordinates of the points. (Recall that for n numbers, the *median* is the result of $\text{select}(\lfloor \frac{n}{2} \rfloor)$.)

We assumed that points are in general position. Therefore we will end up with exactly $\lfloor \frac{n}{2} \rfloor$ points on one side and $\lceil \frac{n}{2} \rceil$ points on the other side. This will be crucial later, and we are *not* allowed to change this balance by a multiplicative factor. Because the split is so even, the height can be bounded (we only state the result, leaving the proof as an easy

exercise).

Lemma 8.3. *A kd-tree on n points in general position has height $\lceil \log n \rceil$.*

We should note that having points in general position is crucial here; otherwise the height could be unbounded! (Constructing such an example is left as an exercise.)

- In cs240, we always split by x -coordinates at the root and then alternate splitting between the levels.

One could think of other strategies. In particular, with badly distributed points the represented rectangles could get very long and skinny, i.e., have very large aspect ratio. Therefore some people find it more convenient to split along the longer side of the associated rectangle.

- As for quad-trees, points that lie exactly on the split line are added to the right/top side. Again this is only a convention, and other strategies could easily be supported with appropriate changes to the search routines.
- Notice that a kd-tree (and also a quad-tree) acts like a decision-tree. Specifically, the interior nodes are only for making search-decisions while all key-value pairs are stored at the leaves. (In general, most tree-based data structures could be converted into such a *decision-tree version*, but since this is somewhat wasteful in space, one usually only does this if it has other advantages. Here for kd-trees we did this so that we would stay as similar to quad-trees as possible.)

One could instead build a tree much like a kd-tree where internal nodes also store one key-value pair (typically the one used for doing the split). This changes the code for range-search slightly; details are left as an exercise.

8.3.1 Dictionary operations

The definition of a kd-tree comes with an algorithm to build it: If there is only one point, then the kd-tree is a one-node tree with this point. Otherwise, find the median x_m of the x -coordinates in the set S of points. Then partition S according to x_m into $S_<$ and $S_>$. Recursively build the kd-tree for $S_<$ and $S_>$, but using the y -coordinate, rather than x -coordinate, for the initial split. The kd-tree then consists of a root that stores x_m with the kd-trees for $S_<$ and $S_>$ as left and right child.

If we use *randomized-quick-select* to find the median x_m , then the expected run-time $T(n)$ to build the kd-tree satisfies $T(n) \leq 2T(n/2) + \Theta(n)$ and hence is in $\Theta(n \log n)$. One can reduce this to $\Theta(n \log n)$ *worst-case* time by pre-sorting the points twice, once by x -coordinate and once by y -coordinate, and pass pre-sorted lists along in the recursion so that median-finding can be done without *quick-select*. We leave the details as an exercise.

The space used by a kd-tree is linear, because when we split then both subtrees store at least one point. In consequence, every interior node has two children, which means that the number of interior nodes is one less than the number of leaves. Since there are n leaves, the space is $O(n)$.

To *search* in a kd-tree is conceptually identical to searching in a quad-tree: Start at the root, determine (based on the split-information stored at the root) which subtree contains the point in its associated region, and recurse in this subtree until you reach a leaf. At this leaf, either the stored point is the one that we are searching for, or the point was not stored. The run-time for this is $O(\log n)$ since we spend constant time on each level.

Operations *insert* and *delete* are also not difficult. To do *insert*(p), locate the leaf where point p should have been, let p' be the point that was actually there, and replace the leaf with a new node that splits the associated region appropriately. To do *delete*(p), simply locate the leaf that stores p and delete it.

The problem with both insertion and deletion is that the structural properties that we demanded for a kd-tree do not necessarily continue to hold. We demanded that the splitting line is at the median of the points. Inserting or deleting may change the median! A second problem is also that during deletion one node (the parent of the deleted leaf) has only one child left (it has become *unary*). We could remedy this by removing the parent, but then the promise that we alternate between splitting by x and splitting by y no longer holds.

We can still build a data structure much like a kd-tree that has $O(\log n)$ height, with ideas as for the scapegoat trees (Section 4.3.1). However, this means that the children will be less balanced, typically having between $\frac{1}{3}n$ and $\frac{2}{3}n$ of the points. This is *not* balanced enough for the bound on the range-search run-time that we will see below. Summarizing, kd-trees do not adapt well to insertion and deletion.

8.3.2 Range search in a kd-trees

A range search in a kd-tree is almost identical to the range search in a quad-tree; the only difference is that the associated region now gets split into two parts, rather than four parts. Algorithm 8.2 shows the code and Figure 8.9 shows an example.

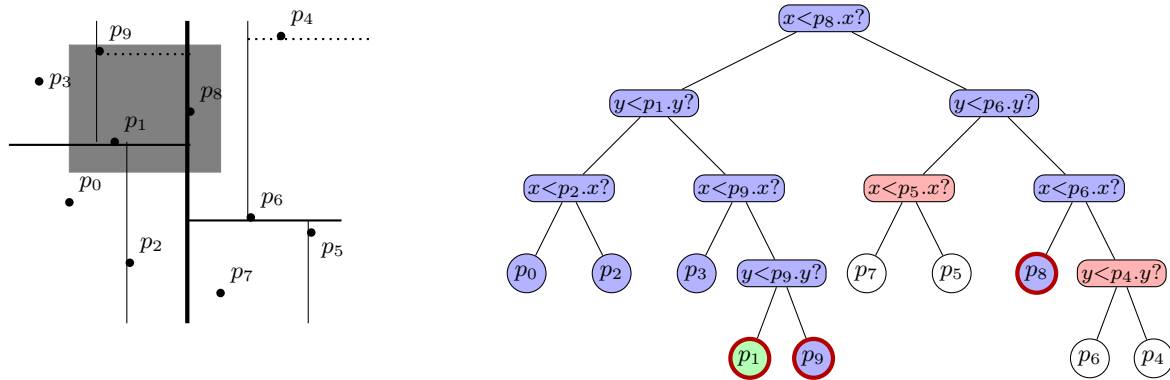


Figure 8.9: Range-search in a kd-tree.

As for quad-trees, we find it convenient to classify the nodes that were visited during a range

Algorithm 8.2: *kdTree::rangeSearch*($r \leftarrow \text{root}, A$)

```

Input : Node  $r$  of a kd-tree. Query-rectangle  $A$ 
1  $R \leftarrow$  region associated with node  $r$ 
2 if  $R \subseteq A$  then                                     // inside node
3 |   report all points in subtree at  $r$  and return
4 else if  $R \cap A$  is empty then                         // outside node
5 |   return
6 else                                                  // boundary node
7 |   if  $r$  is a leaf then
8 | |   if the point  $p$  stored at  $r$  satisfies  $p \in A$  then
9 | | |   return  $p$ 
10 | |   else
11 | | |   return
12 |   else
13 | |   foreach child  $v$  of  $r$  do
14 | | |   kdTree::rangeSearch( $v, A$ )

```

5246 search:

- 5247 • *inside-nodes* (colored green) satisfy $R_v \subseteq A$,
 5248 • *outside-nodes* (colored red) satisfy $R_v \cap A = \emptyset$, and
 5249 • *boundary-nodes* (colored blue) satisfy neither of the above.

5250 Let $Q(n)$ be the number of boundary-nodes that we encountered during the range search.

5251 **Observation 8.1.** *The run-time for a kd-tree range search is $O(Q(n) + s)$, where s is the*
 5252 *output-size.*

5253 *Proof.* We spend $O(1)$ time (not counting the time for the recursion) at each boundary-node.
 5254 At each inside-node v , the time spent is $O(s_v)$, where s_v is the number of points that are stored
 5255 at leaves below v , because we report all these points. Since each reported point is a descendant
 5256 of at most one inside-node, this is $O(s)$ when summed up over all inside-nodes. At each outside-
 5257 node, we spend $O(1)$ time without returning anything. However, since each outside-node has a
 5258 boundary-node as a parent, and each boundary-node has at most two outside-nodes as children,
 5259 the total time spent on all outside-nodes is $O(Q(n))$. \square

5260 The same bound run-time would have held for quad-trees. The reason why we list it here,
 5261 and not for quad-trees, is that for kd-trees we can obtain a bound on $Q(n)$.

5262 **Claim 8.1.** *Let $Q(n)$ be the maximum number of boundary nodes that could happen during a*
 5263 *range search when n points are stored in a kd-tree. Then $Q(n) \in O(\sqrt{n})$.*

Proof. (cs240e) Recall that boundary nodes are all the nodes v where the associated regions R are neither disjoint from the query-rectangle A , nor lie entirely within A . Let us find a different geometric characterization of this. Namely, let $\ell_N, \ell_W, \ell_S, \ell_E$ be the four lines through the boundary of A . These lines define nine cells, see also Figure 8.10. If an associate region R of some node v falls entirely within one of these cells, then it is either disjoint from A or a subset; either way, v is not a boundary node. Put differently

If v is a boundary node, then its associated rectangle R intersects one of the lines $\ell_N, \ell_W, \ell_S, \ell_E$.

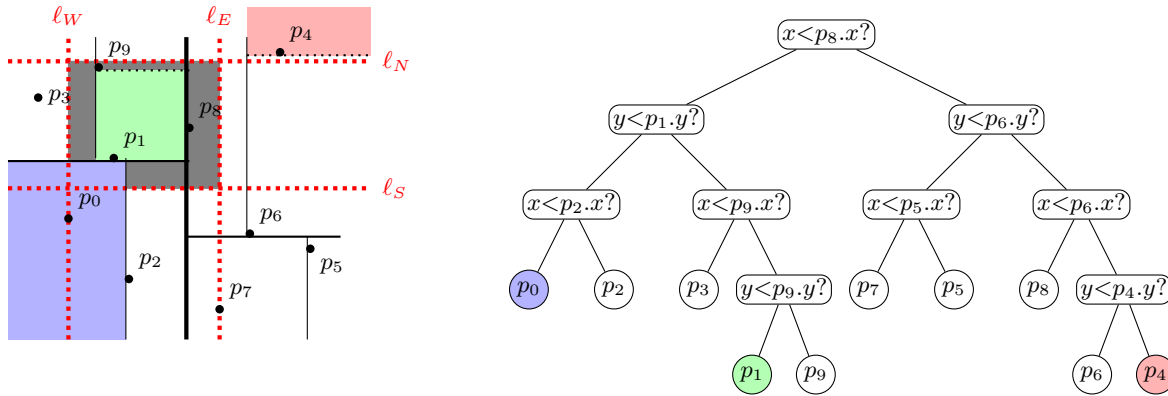


Figure 8.10: The four lines defined by query-rectangle A . If a region R intersects none of them, then R is either an inside or an outside region.

So let us now count how many associated regions could possibly intersect one line. Specifically, for a fixed vertical line ℓ , define

$Q_v(n, \ell)$ = maximum number of nodes (in a kd-tree that stores n points) for which the associated region intersects line ℓ .

It should be clear that the specific line ℓ is quite irrelevant since we are taking the maximum over all kd-trees (we could translate the points), and so sometimes we will simply write $Q_v(n)$. Symmetrically define $Q_h(n, \ell)$ for a horizontal line ℓ , then clearly

$$Q(n) \leq Q_h(n, \ell_N) + Q_h(n, \ell_S) + Q_v(n, \ell_W) + Q_v(n, \ell_E) \leq 2Q_h(n) + 2Q_v(n).$$

Now let us bound $Q_v(n, \ell)$ by inspecting the structure of a kd-tree. Consider Figure 8.11(top), where ℓ is dotted. At the root, we split by some x -coordinate. Line ℓ intersects the region of the root-node, but only *one* of the regions of the children. (In the example in Figure 8.11 this is at the left child.) If this child in turn has children, then each of those grandchildren stores at

most $\lceil \lceil \frac{n}{2} \rceil / 2 \rceil = \lceil \frac{n}{4} \rceil$ points in its kd-tree. Any region intersected by ℓ is hence either the root, or its child, or in the subtree at one of these two grandchildren. Hence

$$Q_v(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 2Q_v(\lceil n/4 \rceil) + 2 & \text{otherwise} \end{cases}$$

5276 One can easily show by induction that therefore $Q_v(n) \in O(\sqrt{n})$. (Specifically, show that
5277 $Q_v(n) \leq 3\sqrt{n} - 2$ if n is a power of 4.)

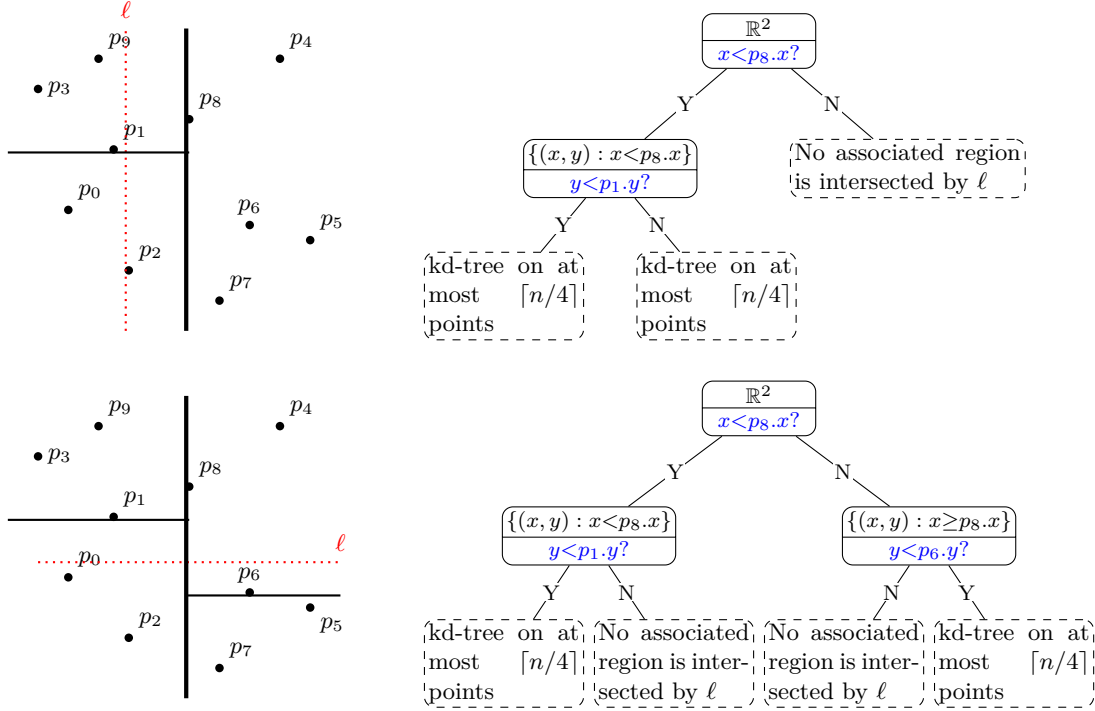


Figure 8.11: For the formulas $Q_v(n) \leq 2Q_v(\lceil n/4 \rceil) + 2$ and $Q_h(n) \leq 2Q_h(\lceil n/4 \rceil) + 3$.

5278 The argument is almost the same for $Q_h(n)$, i.e., when we consider a horizontal line ℓ . See
5279 Figure 8.11(bottom). Here, the region of the root and of both children can be intersected by
5280 ℓ , but at each child only one of the grandchildren can be intersected by ℓ . Therefore $Q_h(n) \leq$
5281 $2Q_h(\lceil n/4 \rceil) + 3$, which again resolves to $O(\sqrt{n})$. Putting it all together therefore $Q(n) \in$
5282 $O(\sqrt{n})$. \square

5283 Combining Observation 8.1 with Claim 8.1, we get that range search in a kd-tree takes
5284 $O(\sqrt{n} + s)$ time. Contrast this to quad-trees, where we had no better bound on the worst-case
5285 run-time of a range search than the size of the entire tree (which in turn cannot be bounded by
5286 n alone). Summarizing, kd-trees are still quite easy to understand and implement, and offer a
5287 guarantee on the worst-case run-time.

8.3.3 kd-trees in higher dimensions

We can easily generalize kd-trees to higher dimension, say for coordinates x_1, \dots, x_d . Then at the root we partition the points by x_1 , at the next level we partition by x_2 , and so on until the d th level where we partition by x_d , then on the next level we partition again by x_1 and so on.

Almost exactly as for $k = 2$ one argues that the height is logarithmic, the construction time is $O(n \log n)$, and the space is linear. The recursion from Claim 8.1 no longer holds, but one can show similar recursions that leads to a run-time bound of $O(s + n^{1-1/d})$ for a range search.

8.4 Range trees

Let us now study a third data structure specifically designed for range searches. This supports *much* faster range searches, at the price of using more than linear space. This is the first time that we see a data structure where the space is asymptotically more than linear (i.e., the space is $\omega(n)$), and yet it is bounded relative to n . A range tree consists of

- the *primary structure*: a binary search tree T that stores all points and where the binary-search-tree order is with respect to x -coordinate, and
- the *associated structures*: each node v of T has a reference to another binary search tree (the *associate structure* $T(v)$).

The points stored in $T(v)$ are defined as follows. Consider the subtree T_v of T that is rooted at v , and let $P(v)$ be all those points that are stored in it (i.e., $P(v)$ are all those points that are at descendants of v (including v itself) in T). Tree $T(v)$ stores the points $P(v)$, and the binary-search-tree order in $T(v)$ is with respect to y -coordinates.

Figure 8.12 illustrates this idea. We assume that both the primary structure and the associated structures are balanced, i.e., have height $O(\log n)$.

A range-tree uses $\omega(n)$ space because nodes are stored repeatedly in the associated structures, and the leaves are in fact stored in *many* associated structures. But we can bound the overall space.

Observation 8.2. *If the primary tree is balanced, then the range tree uses $O(n \log n)$ space.*

Proof. Consider how often one point is stored in the range tree. It is stored once (say at node w) in the primary tree. It is stored in an associated structure $T(v)$ if and only if v is an ancestor of w . Since the primary tree is balanced, each node w has $O(\log n)$ ancestors. Therefore each point is stored $O(\log n)$ times, and the overall space usage is $O(n \log n)$. \square

This bound is tight if the primary tree T is perfectly balanced, since the $\approx n/2$ leaves of T are stored in $\approx \log n$ associated structures each.

8.4.1 Dictionary operations

To search for a point p in a range-tree, simply search for p by its x -coordinate in the primary tree T . Since points are in general position, the x -coordinate uniquely identifies the point, so

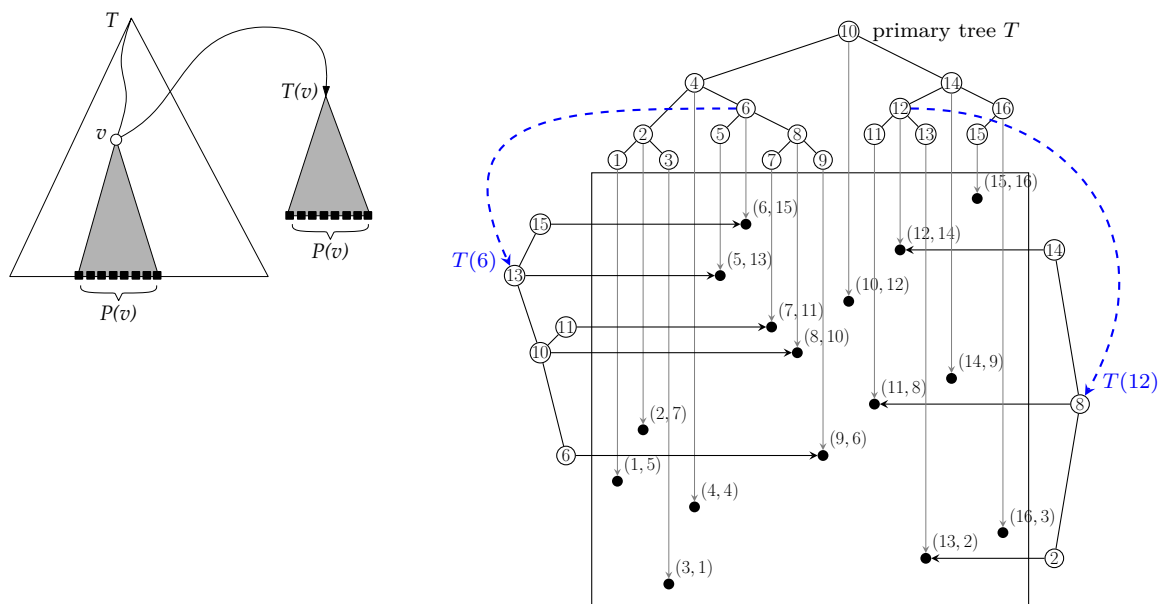


Figure 8.12: A range-tree: the abstract idea (based on a picture of [dBCvKO08]) and a specific example. Not all associated structures are shown.

5323 this will find p if it is stored.

Now let us look at inserting/deleting a point p . (This will also explain how to construct a range-tree, by repeatedly inserting.) We first insert/delete p in the primary structure T by x -coordinates. Now we have two challenges:

- We need to update the associated structures. Say we have done the structural change at T at node z . Now walk back up from z to the root r . Every node v on this path is an ancestor of z , and so we need to insert/delete p in the associated structure $T(v)$, using its y -coordinate as the key.

We do $O(\log n)$ updates (at each ancestor), and each of those takes $O(\log n)$ time (we keep the associated structures balanced), and so the run-time is $O(\log^2 n)$.

- T may now have become imbalanced. We cannot afford to rotate within T to make it balanced, since this would change which nodes are descendants, and we would hence have to update a lot of associate structures which would take too long. But (as opposed to kd-trees) it is not important that we have *exactly* a half-and-half-split among the subtrees; all we need is that T has height $O(\log n)$. Therefore we can use a *scapegoat tree* (Section 4.3.1) for T .

(cs240e) Recall that a scapegoat tree does not use rotations, and instead completely re-
builds a subtree at a node p whenever the subtree at p becomes too unbalanced. This
takes time $\Theta(n_p)$ (where n_p is the size of the subtree at p), but happens rarely enough

so that when averaging over all operations the run-time is still $O(\log n)$. For range-trees, when we rebuild a subtree at p completely, we also rebuild all associated structures. This takes time $O(n_p \log n_p)$, but again happens rarely enough that when averaging over all operations the run-time is still $O(\log^2 n)$. Details are left as an exercise.

In summary, we can insert and delete in $O(\log^2 n)$ amortized time, and build a range tree in $O(n \log^2 n)$ time. It is worth mentioning that there are ways to reduce this run-time by a log-factor using a technique called *fractional cascading*. The idea is to create links between different associated structures so that the search-result that led to the insertion-point in one structure can be used to save search-time in the next one. The details are beyond the scope of cs240.

8.4.2 Range search in a binary search tree

Doing a range search in a range-tree will consists of two parts: First do a range search by x -coordinate in the primary structure, and then, for some of the associate structures, do a range search by y -coordinate.

Correspondingly, we first need to study how to do a range search (say for range (x_1, x_2)) in a binary search tree. This is very simple. Whenever we reach a node (say it stores key k) then we must compare the query-interval to k , and recurse in the children that could possibly have keys in the interval (x_1, x_2) . We also must report k itself, if it falls into the range. The code is given in Algorithm 8.3, with an example in Figure 8.13. We assume here that keys equal to x_1 and x_2 should be included in the returned items; the algorithm is not hard to modify so that they are excluded instead. This code has another other useful property: it reports the points in the query-interval in order of keys, because the order of reporting follows the in-order of the binary search tree.

Algorithm 8.3: *BST::rangeSearch-recursive*($r \leftarrow \text{root}, x_1, x_2$)

Input : Query-interval $[x_1 : x_2]$, node r of a binary search tree

```

1 if  $r = \text{NIL}$  then
2   | return
3 else if  $x_1 \leq r.\text{key} \leq x_2$  then
4   | list  $L \leftarrow \text{BST}::\text{rangeSearch-recursive}(r.\text{left}, x_1, x_2)$ 
5   | list  $R \leftarrow \text{BST}::\text{rangeSearch-recursive}(r.\text{right}, x_1, x_2)$ 
6   | return  $L \cup r.\{\text{key}\} \cup R$ 
7 else if  $r.\text{key} < x_1$  then
8   | return  $\text{BST}::\text{rangeSearch-recursive}(r.\text{right}, x_1, x_2)$ 
9 else //  $r.\text{key} > x_2$ 
10  | return  $\text{BST}::\text{rangeSearch-recursive}(r.\text{left}, x_1, x_2)$ 

```

However, this is *not* the implementation of range search for a binary search tree that we will use later. Instead, we give a much more complicated algorithm for two reasons. First,

we again want to obtain a classification as outside/inside/boundary nodes. The labeling as ‘inside-node’ will be needed when we do range searches in range-trees. Second, based on the classification, it will be much simpler to argue the run-time of the range searches.

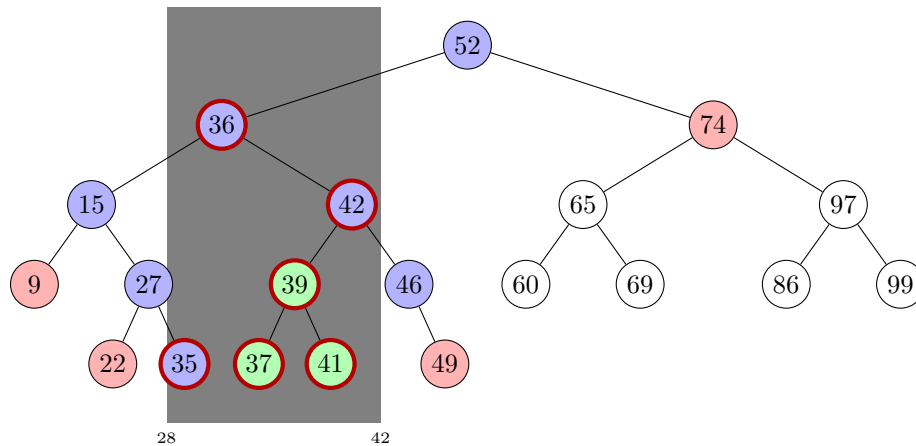


Figure 8.13: The result of $BST::rangeSearch(28, 42)$. Keys in the range are bold. Nodes 9 and 74 are outside nodes.

We proceed as follows (the pseudocode is in Algorithm 8.4):

- Perform $BST::search(x_1)$, i.e., search for the left boundary of the query-interval. This returns the *left boundary path* P_1 .³

We need some (intuitive) notation. Any node v of the tree can be either *on* P_1 , or it can be *strictly left* of P_1 or *strictly right* of P_1 . Formally, a node v is strictly left of P_1 if there exists a node w on P_1 that has v in its left subtree while P_1 either ends at w or continues in the right subtree of w . Being strictly right is defined symmetrically.

All nodes strictly left of P_1 store a key k with $k < x_1$, and so do all their descendants. As soon as we encounter such a node, we can hence declare it an *outside node* and stop the search.

Also observe that any node that is strictly right of P_1 stores a key k with $k \geq x_1$.

- Perform $BST::search(x_2)$, i.e., search for the right boundary of the query-interval. This returns the *right boundary path* P_2 . Any node strictly right of P_2 is an outside node, and any key k that is strictly left of P_2 satisfies $k \leq x_2$.
- The key k of any node that is strictly right of P_1 and strictly left of P_2 hence satisfies

³The code in Algorithm 8.4 does not exactly perform a search, because it does not explicitly check whether the current key equals x_1 , but instead continues the search in the appropriate subtree until we reach an empty subtree. This is quite similar to the difference between our two versions of binary search (see Section 6.1.2) and leads to fewer comparisons overall. It also means that the algorithm works even if the points are not in general position.

Algorithm 8.4: *BST::rangeSearch*(x_1, x_2)

```

1 Initialize empty lists  $L_{ins}, L_{bnd}$ 
2  $v \leftarrow root$  // find node  $v$  where  $P_1$  and  $P_2$  diverge
3 while  $v \neq NIL$  and  $(v.key < x_1$  or  $v.key > x_2)$  do
4    $L_{bnd}.insert(v)$ 
5   if  $v.key < x_1$  then  $v \leftarrow v.right$  else  $v \leftarrow v.left$ 
6 if  $v \neq NIL$  then
7    $L_{bnd}.insert(v)$ 
8    $z \leftarrow v.left$  // find left boundary path  $P_1$ 
9   while  $z \neq NIL$  do
10     $L_{bnd}.insert(z)$ 
11    if  $z.key < x_1$  then  $z \leftarrow z.right$ 
12    else  $L_{ins}.insert(z.right)$  ;  $z \leftarrow z.left$ 
13    $z \leftarrow v.right$  // find right boundary path  $P_2$ 
14   while  $z \neq NIL$  do
15     $L_{bnd}.insert(z)$ 
16    if  $z.key > x_2$  then  $z \leftarrow z.left$ 
17    else  $L_{ins}.insert(z.left)$  ;  $z \leftarrow z.right$ 
18 if we want to report the points then
19   foreach  $v \in L_{bnd}$  do if  $x_1 \leq v.key \leq x_2$  then report point at  $v$  as ‘in range’
20   foreach  $v \in L_{ins}$  do if  $v \neq NIL$  then report points at descendants of  $v$  as ‘in range’
21 else return  $L_{ins}$  and  $L_{bnd}$ 

```

5384 $x_1 < k < x_2$. All descendants of such a node are also strictly right of P_1 and strictly left of
5385 P_2 . So this is an *inside-node*: all its descendants are in the range. We can stop the search
5386 as soon as we encounter the topmost such node and report all its descendants.

5387 • The keys that are on P_1 may be smaller, bigger, or equal to x_1 , and similarly for P_2 . So
5388 the search alone does not tell us whether these are in the range or not. We call these the
5389 *boundary-nodes* and will explicitly test whether they fall in the range or not.

5390 Note that Algorithm 8.4 explicitly finds the boundary nodes (stored in L_{bnd}) and the topmost
5391 inside nodes (stored in L_{ins}), and takes time $O(height)$ to do so. We can choose whether we only
5392 want to return these lists (which will be useful below), or whether actually to print the points in
5393 the range. In the latter case the total time to report the points is $O(height + s)$ since the node
5394 of each reported point is either in L_{bnd} or in exactly one subtree of a node in L_{ins} . Assume that
5395 the binary search tree is balanced, the run-time of the range search is hence $O(\log n + s)$.

Algorithm 8.5: *rangeTree::rangeSearch*(x_1, x_2, y_1, y_2)

Input : Query-rectangle $[x_1 : x_2] \times [y_1 : y_2]$

- 1 $L_{\text{ins}}, L_{\text{bnd}} \leftarrow \text{primaryTree.rangeSearch}(x_1, x_2)$ // without reporting
- 2 **foreach** ($v \in L_{\text{bnd}}$) **do**
- 3 **if** ($x_1 \leq v.x \leq x_2$) and ($y_1 \leq v.y \leq y_2$) **then** report v as ‘in range’
- 4 **foreach** ($v \in L_{\text{ins}}$) **do**
- 5 $v.\text{associatedTree.rangeSearch}(y_1, y_2)$ // with reporting

8.4.3 Range search in a range-tree

With range searches in binary search trees in place, a range search in a range-tree (say for query-rectangle $R = [x_1, x_2] \times [y_1, y_2]$) is now very simple.

- First, perform a range search for $[x_1, x_2]$ in the primary structure, but do not report the point, instead return the lists L_{bnd} and L_{ins} of boundary-nodes and inside-nodes.
- For each boundary node v , explicitly test whether the point at v belongs to R , and report it if it does.
- For each topmost inside node v , we know that all descendants of v in the primary structure T are within the x -range. However, we must test whether they are also in the y -range. To this end, perform a range search in the associated structure $T(v)$ that searches for points in range $[y_1, y_2]$, and report all points in the range. Since $T(v)$ stores only descendants of v , we know that any such point belongs to R .

Figure 8.14 illustrates this process. We do a range search for $[4, 15] \times [9, 11]$. After the range search for $[4, 15]$ in the primary tree, the nodes with x -coordinates 6 and 12 are identified as topmost inside nodes. We therefore do range searches for $[9, 11]$ in the two associated structures $T(6)$ and $T(12)$; this identifies two points in the range in $T(6)$ and none in $T(12)$. One of the boundary-nodes of the range search in T is also in the range and reported.

The run-time can easily be seen to be $O(\log^2 n + s)$. Namely, we have $O(\log n)$ boundary-nodes in T , and hence $O(\log n)$ topmost inside nodes (which are children of boundary-nodes). For each topmost inside node v we spend $O(\log n + s_v)$ time for the range search in $T(v)$, where s_v is the number of points that are reported. Since every reported point belongs to at most one associated structure (the descendants of topmost inside nodes are disjoint), we have $\sum_v s_v \leq s$. Therefore the total run-time is

$$\sum_{\text{topmost inside node } v} (\log n + s_v) \in O(\log^2 n + s).$$

Similar as for *insert*, the run-time can be decrease to $O(\log n + s)$ using fractional cascading, but we will not give any details.

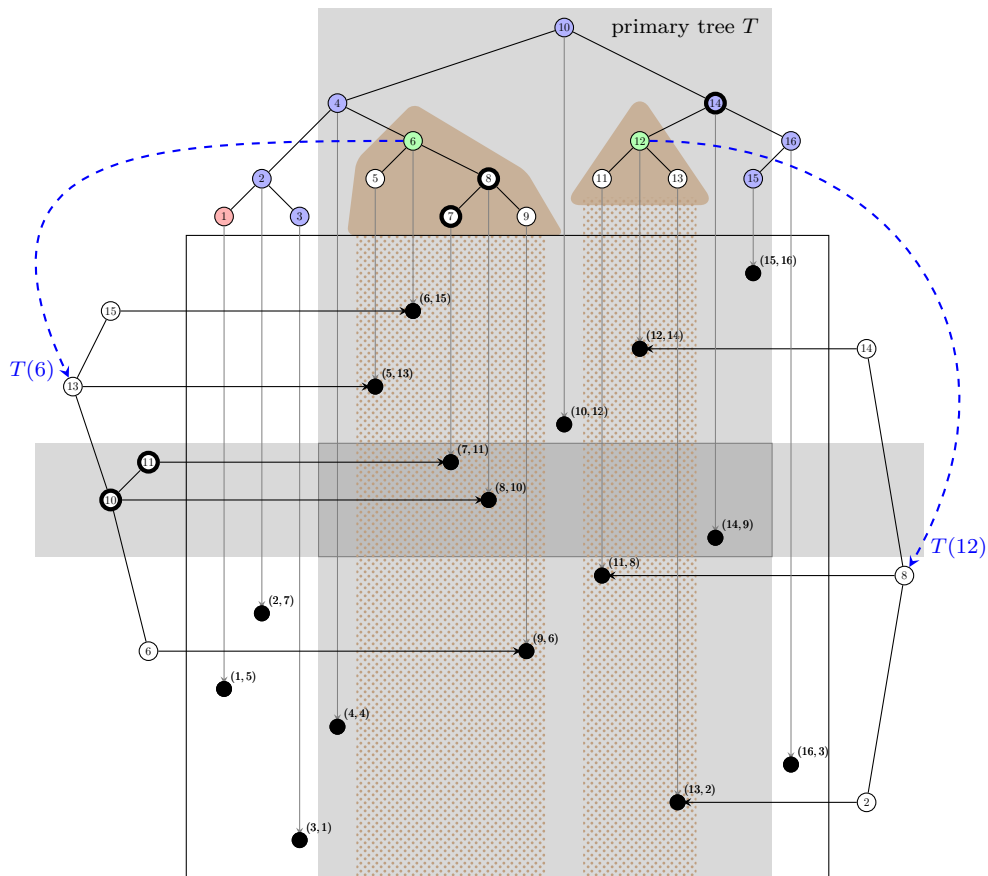
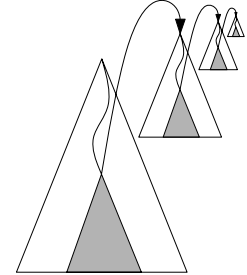


Figure 8.14: Range search in a range-tree.

8.4.4 Range trees in a higher dimension

Range trees are not too difficult to generalize to higher dimensions, though the amount of space used gets increasingly larger. Let us illustrate this for 3-dimensional points. We again store a primary structure T by x -coordinates. Each node v of T stores an associated structure $T(v)$ with all its descendants in T , ordered by y -coordinates. Now each node w in $T(v)$ in turn has an associated structure $T(v, w)$, which stores all descendants of w in $T(v)$, and is ordered by z -coordinates. In general for d dimensions, we have $d - 1$ levels of associated structures. Each node is therefore stored up to $\log^{d-1} n$ times, leading to a total space of $O(n \log^{d-1} n)$. The construction time is easily seen to be $O(n \log^d n)$. Searching is unchanged (it only uses the primary structure), but insertion and deletion again need to update all the associated structures at all nodes on the search-path and take time $O(\log^d n)$ (amortized).



To do a range search in a d -dimensional range-tree, we do one in the primary tree, then in all associated structures of the topmost inside nodes, then in all associated structures of all their topmost inside nodes, etc. In total this takes time $O(\log^d n)$ time, plus the usual $O(s)$ time to report the output.

Comparing d -dimensional range-trees to kd-trees, one notices a tradeoff between the time and the space. kd-trees use only linear space, regardless of the dimensions, but the range search time (which was $O(s + n^{1-1/d})$) gets very close to linear as d gets bigger. On the other hand, the range search time in d -dimensional range-trees is poly-logarithmic (it is $O(s + \log^d n)$), at the price of using super-linear amounts of space.

8.5 3-sided range searches (cs240e)

In this section we study a special case of a range search where the rectangle is unbounded on one side (here the top).

3sidedRangeSearch(x_1, x_2, y'): return (x, y) with $x_1 \leq x \leq x_2$ and $y \geq y'$.

We mostly study this because it is an excellent case study of how to combine previously seen ideas to get increasingly better realizations.

8.5.1 Idea 1: Range trees, except use heaps.

Let us first consider a simple idea that reduces the run-time (but sadly not the space). In range trees, the associated data structure was a binary search tree, so that we could do a 1-dimensional range search in the associated tree. But if we know that the search-interval by y -coordinate will only have a lower bound, then we can use a simpler data structure, namely, a binary heap, for the associated structure. (The primary data structure is a scapegoat tree exactly as before.)

See Figure 8.15. This will make the range search faster because of the following claim (whose proof is left as an exercise):

Lemma 8.4. *Let T be a tree that stores points and satisfies the (max-oriented) heap-order property with respect to y -coordinates. Then we can find all points p in T with $p.y \geq y$ in $O(1 + s)$ time, where s is the output-size.*

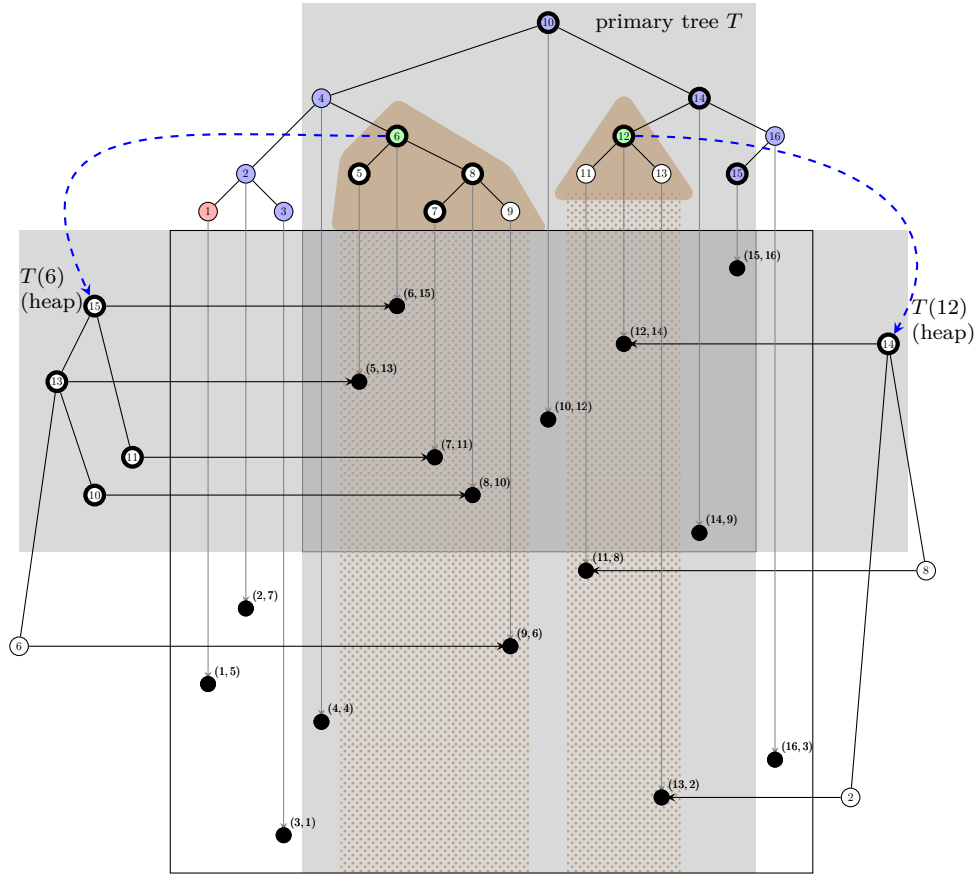


Figure 8.15: $3sidedRangeSearch(3.5, 15.5, 8.5)$ when the associated structures are heaps.

So to do $3sidedRangeSearch(x_1, x_2, y')$ in this data structure, first do $rangeSearch(x_1, x_2)$ in the primary tree to obtain the lists of inside-nodes and boundary-nodes. This takes $O(\log n)$ time. Now explicitly check each boundary-node for whether it is in range; again this takes $O(\log n)$ time. Then for every inside-node v , find the associated structure (now a binary heap), and find all points (x, y) with $y \geq y'$ in it in $O(1 + s_v)$ time, where s_v is the number of reported points. Since every point is reported from at most one inside-node, and there are $O(\log n)$

inside-nodes, the total run-time for this is $O(\log n + s)$. This is a logarithmic factor better than for range trees.

8.5.2 Idea 2: Cartesian trees

However, we can save more space! Recall that we studied treaps (in Section 5.1.1), which are binary search trees where nodes also store priorities, and the tree is a (max-oriented) heap with respect to these priorities. In Section 5.1.1 we chose the priorities randomly to enforce randomization. We now change the approach, and instead use the y -coordinates as priorities while the x -coordinates become the keys. This data structure (illustrated in Figure 8.16) is called a *Cartesian trees*.

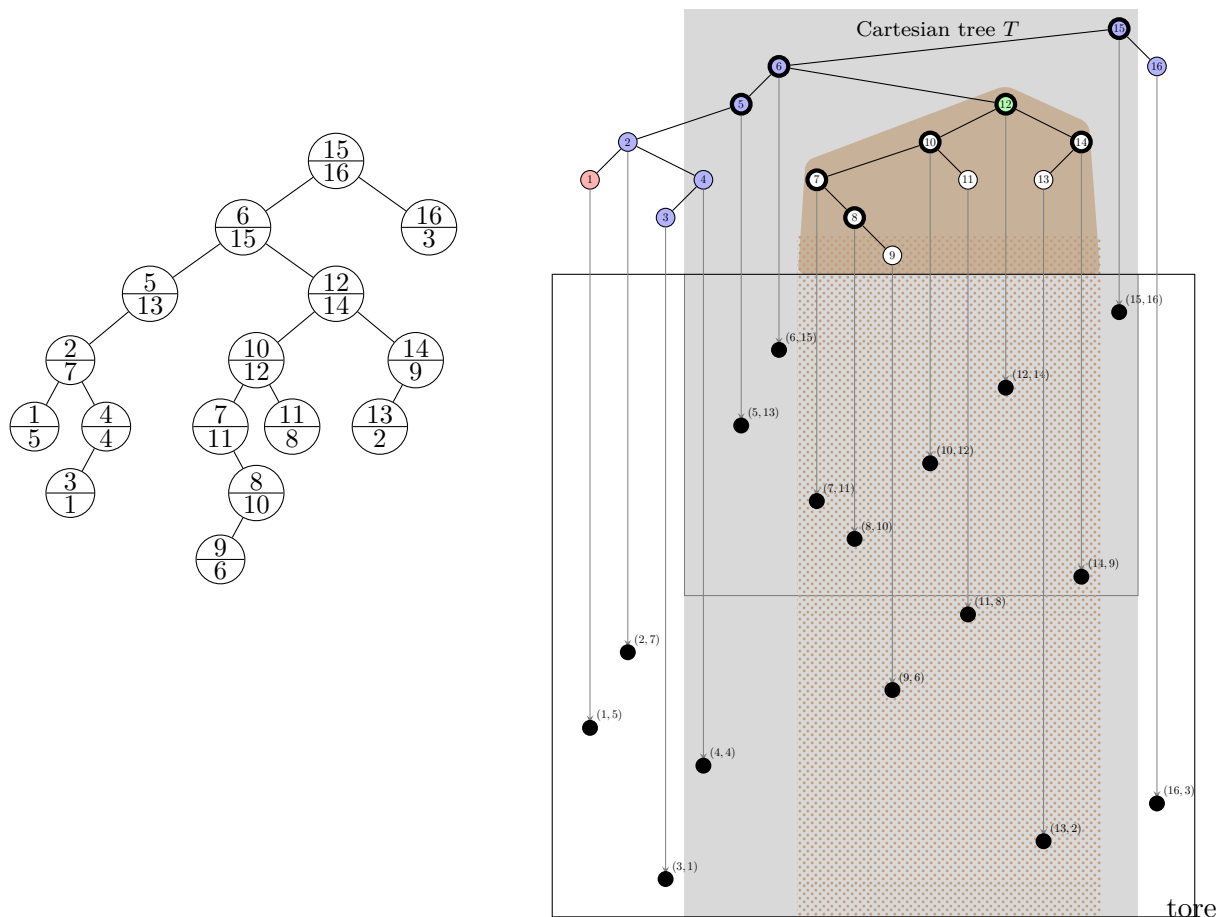


Figure 8.16: Storing points in a Cartesian tree, and performing $3sidedRangeSearch(3.5, 15.5, 8.5)$.

To do a 3-sided range query, we again search for the range $[x_1, x_2]$ in the Cartesian tree, by viewing it as binary search tree with respect to x -coordinates. This identifies boundary and inside

nodes. As before we test each boundary node explicitly. But as opposed to previous approach, we no longer need associated structures for inside nodes! At each inside-node the subtree satisfies the heap-order property (with respect to y -coordinates), and we can immediately perform a search for those nodes with y -coordinates at least y' inside the binary heap and report all those nodes. Hence the run-time is $O(\text{height} + s)$, where height is the height of the Cartesian tree.

The main advantage of using Cartesian tree is that the space-requirement is now $O(n)$. The main disadvantage should be obvious from the example in Figure 8.16: The y -coordinates (hence priorities) force which node must be at the root, and so we no longer have control over the height. So potentially the run-time is linear even if no points are reported.

8.5.3 Idea 3: Priority search tree

The best realization to support 3-sided range queries combines ideas from the Cartesian tree with ideas from kd-trees. In a Cartesian tree, each node stored one point, and they formed a heap with respect to y -coordinates. In kd-trees, interior nodes store a split-line and points are only at leaves. Here we use a 1-dimensional kd-tree, so split-lines are always vertical. In the following data structure, we use both simultaneously, so most nodes store both a point and a split-line. Figure 8.17 illustrates the following definition.

Definition 8.1. A priority search tree is a binary tree where

- every node v stores a point $p_v = (x_v, y_v)$,
- every non-leaf v stores an x -coordinate x'_v (where $x_v \neq x'_v$ is specifically allowed); we call this the split-line coordinate,
- if w is a descendant of v , then $y_w \leq y_v$,
- if w is in the left subtree of v then $x_w < x'_v$, and
- if w is in the right subtree of v then $x_w \geq x'_v$.

Building a priority search tree can easily be done in $O(n \log n)$ expected time, and with some pre-sorting even in $O(n \log n)$ worst-case time. (Details are left as an exercise.) Searching for a point p in the priority search tree is straight-forward: Search for the x -coordinate of p as guided by the split-line-coordinates until we reach a leaf or an empty sub-tree. Point p , if it exists, must be at a node of this search-path, so by comparing to all of them we can test whether p belongs to the tree. This takes $O(\log n)$ time since the height is logarithmic.

Insertion and deletion are feasible in $O(\log n)$ time, but this is non-trivial. As for range-trees, simply inserting by splitting a leaf may destroy the balances. If we used scapegoat trees, then the amortized run-time would be $\Theta(\log^2 n)$, which is too slow. Instead, we restore balances via rotations, but then we need to restore the heap-properties via a process resembling *fix-down*. The details will not be covered here.

Doing a 3-sided range search (say for $[x_1, x_2] \times [y', \infty)$) can easily be done as follows. First, do a range query for $[x_1, x_2]$ by interpreting the tree as a 1-dimensional kd-tree. As before this classifies nodes as outside-nodes, boundary-nodes and inside-nodes; we do not actually report points but only retrieve the lists of boundary-nodes and inside-nodes. (No outside-node can be

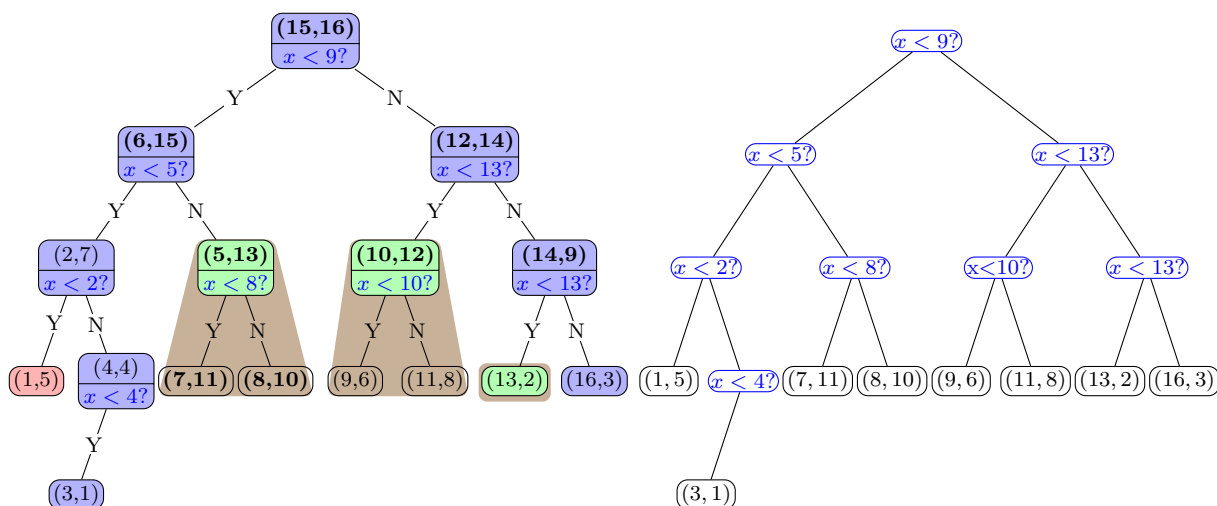


Figure 8.17: A priority-search tree, and the corresponding 1-dimensional kd-tree for the points at the leaves. We also show the node classification when searching for $[3.5, 15.5] \times [8.5, \infty)$; points in bold-face are returned.

in the range.) For any boundary-node, we explicitly check whether it is in the range (note that in contrast to kd-trees the boundary nodes now all store points; we check them explicitly.) For any inside-node v , we interpret the subtree at v as a binary heap. Then, using Lemma 8.4, we can find all points within the subtree at v where the y -coordinates are at least y' and hence fall in the range.

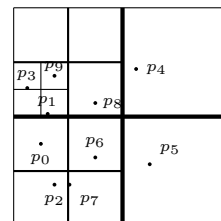
The run-time for this is $O(\log n + s)$, and the space-requirement is $O(n)$, so the priority search tree combines the superior space-requirement of Cartesian trees with the faster search-time since the binary search tree is balanced.

8.6 Take-home messages

We have now seen three very different ways of creating data structures that support (orthogonal) range searches. The following gives a quick overview of the three models:

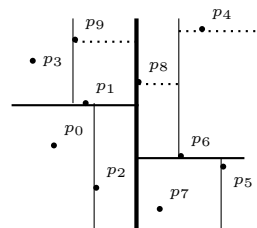
- Quad-trees:

These are very intuitive, easy to implement even for a dynamic set of points, and quite fast if the points are evenly distributed and we implement many operations via bit-shifts. But the worst-case run-time is very bad, and they do not adapt well to higher dimensions.



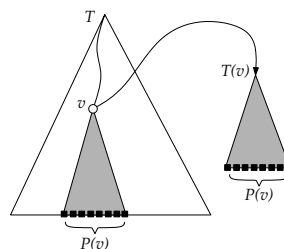
- kd-trees:

These always use linear space, even in higher dimension, while guaranteeing a sub-linear range search time. However, they require points to be in general position and cannot handle inserting or deleting points in an efficient manner. They adapt quite easily to higher dimensions, though the range-search time becomes close to linear.



- Range trees:

These offer by far the fastest range search time, which is poly-logarithmic (beyond the necessary time to print the output). However, the data structure is a lot more complicated and involves a lot of links. Trees can be kept balanced during insertion/deletion by occasionally rebuilding subtrees from scratch. They are somewhat wasteful in space, especially in higher dimensions.



For the enriched section, we also saw that for special kinds of range searches, the run-time and/or space requirements can be improved even further.

Two overall messages to keep in mind before we (finally!) are done with ADT Dictionary:

- Special operations often benefit from special data structures. If you need an operation quite different from the standard *search*, it is worth putting thought into which type of data structure supports this best, and whether the data structure could be enhanced (e.g. with associated structures as was done for range-trees) to enable the operation to be performed faster.
- There are often radically different solutions to a problem. Often they offer a tradeoff between time and space requirements, and the best choice may depend on the type of input data. (This is a theme that we will come back to even more in the next few chapters.)

8.7 Historical remarks

Orthogonal range queries (and more generally the field of computational geometry) became of interest in the 1970s. Quad trees were introduced by Finkel and Bentley in 1974 [FB74], a survey about quad-trees and related structures and applications was given by Samet [Sam84, Sam90]. Bentley improved on (his own!) quad-tree by introducing kd-trees a year later [Ben75]. Originally the splits were done arbitrarily; the idea of using medians came a bit later [FBF77]. The ideas for range trees were discovered in 1979 by (again!) Bentley [Ben79], but also independently around the same time by Lueker [Lue78], Lee and Wong [LW80], and Willard [Wil79].

5534 Research on orthogonal range queries is ongoing and active; see for example a paper from 2011
5535 [CLP11] that shows how to test whether a range is non-empty in time $O(\log \log n)$ under some
5536 assumptions on the model.

5537 As for 3-sided range queries: Cartesian trees were presented here as an adaption of treaps,
5538 but the development actually went the other way around: Vuillemin [Vui80] presented Cartesian
5539 trees in 1980 while their use for simulating randomized binary search trees came almost a decade
5540 later. Priority search tree were designed in 1985 by McCreight [McC85].

5541 For more on orthogonal range search (and the whole wide field of computational geometry),
5542 see the book by de Berg, Cheong, van Kreveld and Overmars [dBCvKO08].

Chapter 9

Pattern Matching

Contents

9.1 Preliminaries	262
9.1.1 Brute-force algorithm	263
9.1.2 Pre-processing	264
9.2 Karp-Rabin fingerprinting	265
9.3 Knuth-Morris-Pratt algorithm	271
9.3.1 Pattern matching with an NFA	271
9.3.2 Pattern matching with a deterministic finite automaton	272
9.3.3 The KMP-automaton	273
9.3.4 Knuth-Morris-Pratt pattern matching	274
9.3.5 Computing the failure array	277
9.3.6 Related finite automata (cs240e)	280
9.4 Boyer-Moore algorithm	282
9.4.1 The last-occurrence heuristic	282
9.4.2 Simplified Boyer-Moore	285
9.4.3 The good-suffix heuristic (cs240e)	286
9.5 Suffix trees and suffix arrays	290
9.5.1 Trie of suffixes	291
9.5.2 Suffix trees	291
9.5.3 Suffix arrays	293
9.6 Take-home messages	295
9.7 Historical remarks	296

In the previous chapters, we looked at searches that were *structured* in the sense that we exactly knew the kind of key that we would be searching for, and therefore could create a data structure specifically tailored to the type of key.

We are now turning to searches where much less is known in advance. We are in particular looking at *string searching* or *pattern matching* which is the following problem:

Given a *text* T and a *pattern* P , does P occur in T ?

As a token example, consider the text $T = \text{“Where is she?”}$. Then “he” does occur in text T (twice, in fact), but “who” does not.

You do pattern matching every time when you hit Ctrl-F (or an equivalent key-combination) to search for a word in a document. It is also a crucial tool within any web search engine that crawls through numerous web pages and executes pattern matching for the keywords you specified. Another application comes from bioinformatics, where your search within a DNA string for the pattern of particular gene. So it is a very commonly used problem, and we will specify here many ways of solving it.

9.1 Preliminaries

Let us clarify a few definitions and conventions:

- The text T (also called the *haystack*) consists of n characters of some alphabet Σ . There is no restriction on the alphabet Σ (we will usually use ASCII), and it can include characters for white-space such as breaks between words, paragraphs or pages. In particular, we could view an entire web page as one text T . In consequence, we should think of n as truly huge, and algorithms with super-linear run-time are probably too slow in practice. We write $T = T[0..n-1]$, i.e., address the individual characters of T as if T were stored in an array.
 - The pattern P (also called the *needle*) consists of m characters of the same alphabet Σ . We write $P = P[0..m-1]$. We know that $m \leq n$ (else P cannot occur within T), and usually m is a *lot* smaller than n . However, m is not necessarily of constant size, and our run-time bounds will depend on both n and m .
 - Pattern P *occurs* in T if there exists an index i such that $T[i..i+m-1] = P[0..m-1]$. The notation “ $T[i..i+m-1] = P[0..m-1]$ ” is a shortcut for “every character of $T[i..i+m-1]$ equals the corresponding character in $P[0..m-1]$ ”.
 - An equivalent way to say this is to say that that $\text{strcmp}(T[i..i+m-1], P[0..m-1]) = 0$ for some i . Operation *strcmp* was defined in Algorithm 6.5 on Page 187, and was specifically designed to handle substrings of the first word. We can hence do this comparison by calling *strcmp*($T, P, i, i + m$); we do *not* need to spend time copying T into a new array. Recall that *strcmp* is *not* a constant-time operations; we need $\Theta(m)$ time in the worst case.
 - We call a substring $T[i..i+m-1]$ of length m a *guess*; when we want to be specific about index i we call it the *i th guess* and denote it by T_i . In our illustrations to come (see e.g. Figure 9.2), we typically create a matrix that has the text T on the top and creates a row for every guess (in gray) that was considered.
- A guess is called *correct* if it corresponds to an occurrence of P , and *incorrect* or a *mis-match* otherwise. We can determine this with a string-compare, but sometimes break the operation down into individual *checks*, i.e., individual comparisons of a character of the guess with the corresponding character of P . To be sure that a guess is correct we must check *all* characters of the guess and so need m checks, but at an incorrect guess fewer

checks may suffice.

- If P does not occur in T , then the answer to the pattern matching question should be **FAIL**. If P does occur in T , then our convention will be to return the *first* occurrence of P in T , i.e., the occurrence that minimizes i . Most algorithms that we will see can easily be adapted to return instead *all* occurrences.

Observe that saying “ P occurs in T ” is the same as saying “ P is a substring of T ”. We clarify here that the set of substrings includes the *empty word* (consisting of no characters at all and usually denoted by Λ); we use the term *non-empty substrings* if we only mean substrings with at least one character.

Recall that a *prefix* of T is any word that equals $T[0..j]$ for some $0 \leq j \leq n-1$, or that is the empty word Λ . Similarly, a *suffix* of T is any string that equals $T[i..n-1]$ for some $0 \leq i \leq n-1$ or is Λ . For example, “bear” has prefixes “bear”, “bea”, “be”, “b”, Λ and suffixes “bear”, “ear”, “ar”, “r”, Λ . The *i*th *prefix* of a word is the prefix with exactly i characters, i.e., for text T the *i*th prefix is $T[0..i-1]$. The following observation (illustrated in Figure 9.1) is trivial but crucial:

Observation 9.1. *The following three statements are equivalent:*

- P is a substring of T .
- P is a prefix of a suffix of T .
- P is a suffix of a prefix of T .

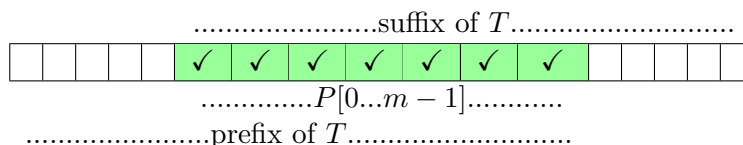


Figure 9.1: If P appears as substring of T , then it is a prefix of a suffix, and a suffix of a prefix.

9.1.1 Brute-force algorithm

The simplest and most obvious approach for pattern matching is to check every possible guess using a string-compare. This is called the *brute-force algorithm*, see Figure 9.2 for an illustration and Algorithm 9.1 for the pseudo-code.

Algorithm 9.1: *bruteForce::patternMatching(T, P)*

Input : Text T of length n , Pattern P of length m

```

1 for  $i \leftarrow 0$  to  $n - m$  do
2   if  $\text{strcmp}(T[i..i+m-1], P) = 0$  then return found at guess  $i$ 
3 return FAIL
```

	a	b	b	b	a	b	a	b	b	a	b
a	a	b	b	a							
		a									
			a								
				a							
					a	b	b				
						a					
							a	b	b	a	

Figure 9.2: The brute-force algorithm on text $T = \text{abbbababbab}$ with pattern $P = \text{abba}$.

At first glance, this may look like a linear-time algorithm, but keep in mind that *strcmp* takes $\Theta(m)$ time in the worst-case. So the worst-case run-time is in $O(m \cdot (n-m+1))$. This is tight. For example, consider the text $T = a^n$ (i.e., n characters of a in a row) and the pattern $P = a^{m-1}b$. On every guess, we have to do the full m checks before realizing at the last character that the guess is a mismatch. There are $n-m+1$ guesses, so the run-time for this input is $\Omega(m(n-m+1))$.

However, brute-force works very well if m is a small constant, or m is very close to n (i.e., $n - m$ is a constant). It also works well if either text T or pattern P looks like random text, i.e., has no discernible patterns, because *strcmp* has $O(1)$ average-case time presuming all input texts are used equally often. Unfortunately, the texts typically do *not* look like random texts but instead follow some language-rules and have many repeating patterns. As such, we should not trust an average-case run-time bound here. (Your thoughts should now immediately be “could we *randomize* pattern matching?” It is really not clear how one can do this, but the algorithm that we will see in Section 9.2 can be viewed as a randomized version of brute-force pattern matching.)

9.1.2 Pre-processing

We have put no constraints on m , and so it need not be a constant or close to n . In particular, we could have $m = n/4$ or $m = n/2$, and then the run-time of the brute-force algorithm would be quadratic in the worst case. This is too large in practice since n is huge. We therefore apply *pre-processing* to speed up pattern matching. The idea of pre-processing applies to many problems (not just pattern matching) and is the following: Before even starting to solve the problem, build some data structures and/or extract other information from the data that will make later operations easier. This may take some time, but typically has to be done only once. With this data structure/information in place, we are then ready to parse a *query*, i.e., do the actual searching, and the data structure/information should help to make the query faster. We have actually already seen this idea in Chapter 8 where we did range searches. We *could* have done a range search on any data structure that stores points. But instead we pre-processed the points to build a new data structure (e.g. a range tree) so that range searches then become faster.

Notice that the same data structure can be used for all later queries, which makes pre-processing especially worthwhile if we expect multiple queries of the same type.

Specifically for pattern matching, we have two possible ways in which we can do pre-processing:

- We could pre-process the pattern P . Storing information about P allows us to eliminate many guesses before we do a single check on them.

Pre-processing the pattern P is especially appropriate if we expect to be searching for the same pattern in many different texts. For example, if you search for keywords in all the files within one directory, then it makes sense to pre-process the keywords (your pattern) because then you search for this pattern in all the files (multiple texts).

- We could pre-process the text T . By Observation 9.1 storing suffixes of T in a suitable way permits us to find matches quickly.

Pre-processing the text T is especially appropriate if we expect to be searching in the same text repeatedly. For example, in bioinformatics, we might have a DNA string and search for many genes (or similar snippets of DNA string) repeatedly; we should then pre-process the DNA string. Also, much of the efficiency of web-search engines is based on the idea of pre-processing a web-page (the text) so that different searches for different keywords (the pattern) can all be very fast.

See Figure 9.3 for the preprocessing methods that we will study later. Amazingly enough, it will turn out that even with only *one* query, pattern matching can be made faster than the brute-force method by preprocessing either the text or the pattern.

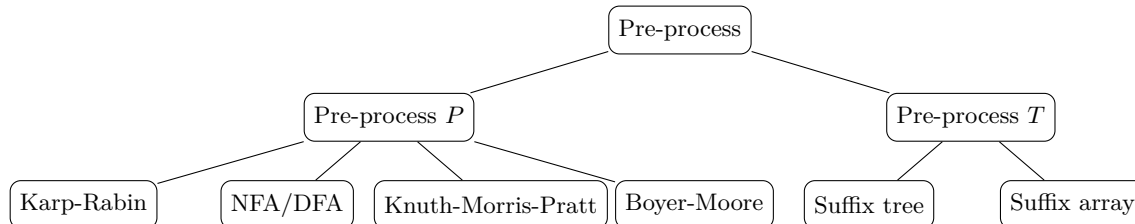


Figure 9.3: Methods of pre-processing for pattern matching.

9.2 Karp-Rabin fingerprinting

The idea for the Karp-Rabin method of pattern matching is to eliminate quickly many guesses that could not possibly be correct. To this end, we fix some *hash-function* $h(\cdot)$ that maps words of length m to integers. (The word ‘hash-function’ is used here only as a convenient shortcut for ‘some function that is easy to compute’; there is no dictionary stored here.)

In particular, hash-function $h(\cdot)$ can be computed for the pattern P (which has length m), and for any guess $T[i..i+m-1]$. These hash-values are called the *fingerprints*. If the fingerprint

5693 of a guess is different from the fingerprint of pattern P , then surely this guess is incorrect and
 5694 we can move on to the next one.

Let us consider an example (illustrated in Figure 9.4) that uses

$$P = 5\ 9\ 2\ 6\ 5, \quad \text{and} \quad T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5.$$

(For illustrative purposes we assume here that $\Sigma = \{0, \dots, 9\}$ so that texts can be viewed as numbers is base-10.) We use as hash-function the standard hash-functions for words, computed with *word-hash-value* (Algorithm 7.5 from Page 226). Recall that this consists of flattening the word to an integer using some radix (here $R = 10$), and then taking the result modulo some value M . (Based on the similarity with hashing, we call M again the “table-size”, though there is no table here.) In this example we use $M = 97$. Thus the hash function (for $m = |P| = 5$) is

$$h(x_0 \dots x_4) = (x_0 x_1 x_2 x_3 x_4)_{10} \bmod 97.$$

5695 Pre-computing this for P , we get $h(P) = 59265 \bmod 97 = 95$. Now parse through all possible
 5696 guesses. For the first four of them, we observe that the hash-value is not 95; we can hence
 5697 immediately eliminate the guess without having to do a string-compare.

	3	1	4	1	5	9	2	6	5	3	5
	hash-value 84										
		hash-value 94									
			hash-value 76								
				hash-value 18							
					hash-value 95						

Figure 9.4: Karp-Rabin example.

Algorithm 9.2: *KarpRabin::patternMatching*(T, P) // naive version

```

1  $h_P \leftarrow \text{word-hash-value}(P, 10, M)$ 
2 for  $i \leftarrow 0$  to  $n - m$  do
3    $h_T \leftarrow \text{word-hash-value}(T[i..i+m-1], 10, M)$ 
4   if  $h_T = h_P$  then
5     if  $\text{strcmp}(T, P, i, i+m-1) = 0$  then
6       return “found at guess  $i$ ”
7 return FAIL

```

5698 Algorithm 9.2 shows the pseudo-code for this idea. Observe that this algorithm is correct,
 5699 because it always finds a correct match if one exists. Namely, if P equals guess $T[i..i+m-1]$,
 5700 then $h(T[i..i+m-1]) = h(P)$, and so the algorithm will execute *strcmp* and find the match.

However, there is a problem with this algorithm. We compute $h(T[i..i+m-1])$ using *word-hash-value*. The run-time for this is $\Theta(m)$, which is no better than doing a string-compare, and so would be no faster overall than the brute-force method. The reason why Karp-Rabin fingerprinting can be made efficient is that we can compute the fingerprint of one guess much faster if we know the fingerprint of the previous guess. Consider the example in Figure 9.4.

- Let us assume that we have pre-computed $10000 \bmod 97 = 9$.
 - Let us also assume that we know the fingerprint of the third guess, which is $41592 \bmod 97 = 76$.
 - We now want to compute the fingerprint of the next guess: $15926 \bmod 97 = ??$
- But observe that

$$15926 = (41592 - 4 \cdot 10\,000) \cdot 10 + 6 \quad \text{and therefore}$$

$$\begin{aligned} 15926 \bmod 97 &= \left(\left(\underbrace{41592 \bmod 97}_{76 \text{ (previous fingerprint)}} - 4 \cdot \underbrace{10000 \bmod 97}_9 \right) \cdot 10 + 6 \right) \bmod 97 \\ &= \left((76 - 4 \cdot 9) \cdot 10 + 6 \right) \bmod 97 = 18 \end{aligned}$$

In general, since the two guesses overlap on all but two digits, we can compute each fingerprint from the previous fingerprint in $O(1)$ time as follows:

$$h(T[i+1..i+m]) = \left(\left(\underbrace{h(T[i..i+m-1])}_{\text{previous fingerprint}} - T[i] \cdot \underbrace{10^{m-1} \bmod M}_{\text{pre-computed}} \right) \cdot 10 + T[i+m] \right) \bmod M$$

We say that a hash-function is a *rolling hash-function* if such a formula exists, i.e., each fingerprint can be computed in constant time given the value of the previous fingerprint. Any rolling hash-function could be used for Karp-Rabin-style pattern matching, but the expected run-time bound requires to use *word-hash-value*.

Algorithm 9.3 gives the final algorithm for Karp-Rabin pattern matching. Note that we *insist* on M being a prime (this is crucial in the analysis below and generally a good idea since it avoids patterns in the text). Also, every time that we had a *false positive* (i.e., two fingerprints that matched, but the corresponding guess was incorrect), we re-set this prime M to a new number and re-compute the corresponding hash-values from scratch. As we will see below, false positives should be very rare and therefore this does not affect the overall run-time.

Now we analyze the expected run-time. We will show the following:

Lemma 9.1. *The probability of at least one false positive is at most $\frac{2c}{n}$ for some constant c and all sufficiently large n .*

Note that as you search in longer and longer strings, the probability of a false positive decreases! So even just having one false positive is very unlikely; having more than one is even less likely.

Algorithm 9.3: *KarpRabin::patternMatching*(T, P)

Input : T and P are texts of length n and m over alphabet $\Sigma = \{0, \dots, R-1\}$

```

1 bool needToReset  $\leftarrow$  TRUE
2 for  $i \leftarrow 0$  to  $n - m$  do
3   if needToReset then           // get random prime and initialize fingerprints
4      $M \leftarrow$  prime number randomly chosen in  $\{1, \dots, mn^2 \log R\}$ 
5      $h_P \leftarrow$  word-hash-value( $P, R, M$ )
6      $h_T \leftarrow$  word-hash-value( $T[i..i + m - 1], R, M$ )
7      $s \leftarrow$  word-hash-value('10...0',  $R, M$ ) // passed word has  $m$  chars
8     needToReset  $\leftarrow$  FALSE;
9   else                           // get next fingerprint from previous
10     $h_T \leftarrow ((h_T - T[i-1] \cdot s) \cdot R + T[i+m-1]) \bmod M$ 
11  if  $h_T = h_P$  then
12    if strcmp( $T, P, i, i+m-1$ ) = 0 then
13      return "found at guess  $i$ "
14    else
15      needToReset  $\leftarrow$  TRUE;
16 return FAIL

```

Now we distinguish two cases. In the first case, there are no false positives at all. Then the run-time is in $O(m + n)$, because we use $O(m)$ time to initialize the fingerprints and $O(1)$ time per index i to check and update the fingerprints. (We are ignoring here the run-time to compute the prime; as discussed below it is insignificant in comparison.) In the second case, there are some false positives. At the very worst, we have n false positives and the run-time hence is $O(n(m + n))$. This second case occurs with probability at most $\frac{2c}{n}$. Let d be the constant hidden by the O -notation, i.e., the run-time is at most $d(n + m)$ in the first case and at most $dn(n + m)$ in the second case. Then

$$T^{\text{exp}}(n) \leq P(\text{no false positives})d(m+n) + P(\text{at least one false positive})dn(m+n)$$

$$d(m+n) + \frac{2c}{n}dn(m+n) = (1 + 2c)d(m+n) \in O(m + n)$$

5729 so the expected run-time of Karp-Rabin pattern matching is $O(m + n)$.

5730 This run-time is not as good as the one of some other algorithms that we will see later,
 5731 because the bound is only for the expected run-time, while other algorithms achieve $O(m + n)$
 5732 worst-case run-time. However Rabin-Karp fingerprinting achieves this run-time with only $O(1)$
 5733 auxiliary space, while later methods will use $\Omega(m)$ auxiliary space.

5734 **Finding a random prime number (cs240e)** In Algorithm 9.3, we must find a prime
 5735 number in the set $\{1, \dots, x\}$ for some x . The simplest method of doing this is to choose a

random number M in the range, to test whether M is a prime, and to repeat until we have found a prime. Testing whether a number is a prime is *not* trivial (in fact, it was proved to be doable in deterministic polynomial time only in 2002 [AKS04], and the algorithm is rather slow in practice). However, testing primality can also be done with a randomized algorithm in $O(\log M) \subset O(\log x)$ time (we will not give details of this, see [Rab80]). While this algorithm might occasionally answer (incorrectly) that M is a prime, this will be good enough.

Now often will we have to repeat until we find a prime? The following definition and result are from number theory and will not be shown here:

Lemma 9.2. *For any integer $x \geq 2$, let $\Pi(x)$ be the set of all primes that are no greater than x , and let $\pi(x) = |\Pi(x)|$. Then $\frac{x}{\ln x} \leq \pi(x) \leq c \frac{x}{\ln x}$ for some constant $c > 1$ and all sufficiently large x .*

Therefore, the probability that our randomly chosen number is a prime is in $\Theta(\frac{1}{\ln x})$, where $x = mn^2 \log R$ is the upper bound that we use for M . It follows that we would expect to have found a prime after we tried $\Theta(\ln x)$ numbers for M . So in total the expected run-time to find M is $O(\log^2 x) = O(\log m + \log n + \log \log R)$. Contrasting this with the run-time of $\Theta(m)$ that we had budgeted for initializing hash-values, this *can* be bigger, but only if m is very small compared to n (or $\log R$, but usually R is a constant). So technically we should use $O(m + \log n + \log \log R)$ as bound for initializing the fingerprints. Practically we would not use Rabin-Karp unless m is fairly big (otherwise the brute-force algorithm works as well) and so we will not bother cluttering our run-time bounds with this.

The proof of Lemma 9.1 (cs240e) Now we show that the probability of a false positive is quite small. We need some notations first. Let us write T_i for the i th guess (i.e., the string $T[i..i+m-1]$). Also write $H(T_i)$ and $H(P)$ for the strings T_i and P interpreted as base- R numbers. (In other words, $h_P = H(P) \% M$.) Define $h_{T_i} = H(T_i) \% M$, i.e., this is the value of h_T when the running index is i . Now we repeatedly reformulate what it means to have a false positive:

$$\begin{aligned}
 \text{false positive} &\Leftrightarrow \text{for some } i, h_P = h_{T_i} \text{ but } P \neq T_i \\
 &\Leftrightarrow \text{for some } i, h_P = h_{T_i} \text{ but } H(P) \neq H(T_i) \\
 &\quad (\text{since } H(\cdot) \text{ defines a 1-1-correspondence}) \\
 &\Leftrightarrow \text{for some } i, H(P) \equiv_M H(T_i) \text{ but } H(P) \neq H(T_i) \\
 &\Leftrightarrow \text{for some } i, |H(P) - H(T_i)| \equiv_M 0 \text{ but } H(P) \neq H(T_i) \\
 &\Leftrightarrow \left(\underbrace{\prod_{i: H(P) \neq H(T_i)} |H(P) - H(T_i)|}_X \right) \equiv_M 0
 \end{aligned}$$

where the last equivalence holds since M is prime, and so multiplying factors that are all not divisible by M cannot give a number that is divisible by M .

Let X be the value of the product in the last row. Also let $\omega(X)$ be the set of distinct primes that are divisors of X . Then

$$P(\text{false positive}) = P(X \equiv_M 0) = P(M \text{ divides } X) \leq \frac{|\omega(X)|}{\pi(mn^2 \log R)}.$$

Here the last inequality holds because we picked prime M randomly and uniformly among $\Pi(mn^2 \log R)$, and we have a false positive only if the picked prime-number M was a prime divider of X . In other words, the number we picked for M was one of at most $|\omega(X)|$ numbers among $\pi(mn^2 \log R)$ possibilities.

Observe that $|H(P) - H(T_i)| < R^m$ for any i , and therefore X (which is the product of at most n such terms, and actually fewer) satisfies

$$X < R^{nm} = 2^{nm \log R}.$$

We can also lower-bound X as

$$X = \prod_{q \in \text{prime factorization of } X} q \geq \prod_{q \in \omega(X)} q \geq \prod_{q \in \text{first } |\omega(X)| \text{ prime numbers}} q.$$

Finally, we need one more result from number theory that we will not prove here, see [RS62]: For any large enough x we have $\prod_{q \in \Pi(x)} q > \frac{e^x}{e^{1/2 \ln x}} > 2^x$. Combining these three inequalities we get

$$\prod_{q \in \Pi(nm \log R)} q > 2^{nm \log R} > X > \prod_{q \in \text{first } |\omega(X)| \text{ prime numbers}} q.$$

This means that there must be at least $|\omega(X)|$ prime numbers in $\Pi(nm \log R)$, or $|\omega(X)| \leq \pi(nm \log R)$. Now we combine this with the bound on $\pi(x)$ from Lemma 9.2 to get that

$$P(\text{false positive}) \leq \frac{|\omega(X)|}{\pi(mn^2 \log R)} \leq \frac{\pi(nm \log R)}{\pi(mn^2 \log R)} \leq \frac{c \frac{nm \log R}{\ln(nm \log R)}}{\frac{mn^2 \log R}{\ln(mn^2 \log R)}} \leq c \cdot \frac{1}{n} \cdot \underbrace{\frac{\ln(n^2 m^2 (\log R)^2)}{\ln(nm \log R)}}_2 = \frac{2c}{n}$$

as desired.

Why do we re-set M ?(cs240e) The astute reader will notice that we did not use in our proof that the prime number M gets re-set after each false positive. Indeed, the expected run-time bound would hold even if we used the same M throughout. However, in practice, re-setting M takes very little extra time (we had just before done a *strcmp* for which we need to budget $\Theta(m)$ time anyway), and significantly decreases the likelihood of further false positives. Specifically, recall that in real-life texts often have repeating substrings. If one of our substrings led to a false positive, then later repetitions would lead to more false positives. Choosing a new M makes it exceedingly unlikely that we will run into the same false positive again.

9.3 Knuth-Morris-Pratt algorithm

We now give a completely different approach to pattern matching, which is based on the idea that if we have a mis-match, then we should shift the guess forward in such a way that the parts that were matched already will again be matched. One can explain this directly (and some instructors may do so and skip to Section 9.3.4), but one especially easy way to understand why it works is to do a detour into how to do pattern matching using finite automata.

9.3.1 Pattern matching with an NFA

You should be familiar with finite automata from cs241. Briefly, a *non-deterministic finite automaton* (NFA) consists of a set of *states* Q , a *start-state* $q_0 \in Q$, a set of *accepting states* $A \subset Q$ and a *transition function* $\delta : Q \times \Sigma \rightarrow 2^Q$. To *parse* a text $T[0..n-1]$ (over alphabet Σ) means to start at the start-state s , and to go (for $i = 0..n-1$) from the current state q to some other state q' for which $q' \in \delta(q, T[i])$. The text is *accepted* if we are at an accepting state after we parsed $T[n-1]$.

It is very easy to create an NFA that accepts a text T if and only if pattern P occurs as a substring of T (see Figure 9.5):

- Create states $0, \dots, m$, where 0 is the start-state, and $m = |P|$ is the accepting state.
- Add a transition from i to $i + 1$ labeled with $P[i]$.
- Finally add transitions within 0 and m that are labeled with all of Σ .

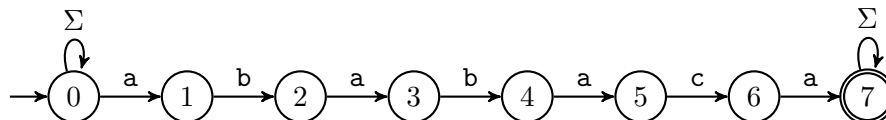


Figure 9.5: The non-deterministic finite automaton for the pattern $P = ababaca$.

To see that this works, assume first that $P = T[i..i+m-1]$ for some index i . Then we can reach an accepting state by staying in state 0 for the first i characters of T , then parsing the characters of $T[i..i+m-1] = P$ to get to state m , and then staying in state m until the end. Vice versa, if we reach state m while parsing T , then we must have had a substring of T that followed the arcs from 0 to m , which happens only if P occurs in T .

For future reference, we would like to observe exactly what holds if we are in a state j . (Later approaches will build on top of this idea.)

Observation 9.2. Assume we have reached state j at some point during the parsing with the NFA (say when we have parsed $T[0..i]$). Then the last j characters of $T[0..i]$ equal the first j characters of P . Equivalently, $T[i-j+1..i] = P[0..j-1]$. Equivalently,

the j th prefix of pattern P is a suffix of what we have parsed

Pattern matching with an NFA is easy to explain, but does not give a fast algorithm since we cannot test quickly whether we can reach the accepting state. Note that at state 0 we have (with each occurrence of $P[0]$ in the text) a choice of whether to stay at state 0 or whether to go to state 1. To do string matching with an NFA, we would have to try out *all* these possibilities. Since there may be $\Theta(n)$ possibilities, this would lead to a run-time of $O(mn)$, no better than brute-force. In fact, pattern matching with an NFA is exactly the same as the brute-force algorithm, only described in a very different way.

9.3.2 Pattern matching with a deterministic finite automaton

You may recall from cs241 that any non-deterministic finite automaton can be converted into an equivalent *deterministic finite automaton* (DFA), i.e., an automaton where $|\delta(q, a)| = 1$ for all $q \in Q, a \in \Sigma$. In other words, there is never any choice as to which transition to use next. In a DFA it is very easy to test in linear time whether a text T is accepted, because there is never any choice about the transition to use, and so we follow the unique transition required by text T and either reach an accepting state at the end or not.

The price to pay is that in general the DFA can be exponentially bigger than the corresponding NFA. However, since the NFA that we built for pattern matching has a special structure (state 0 is the only state where two outgoing transitions have the same character), one can actually build a DFA for pattern matching that is linear in the size of P . See Figure 9.6.

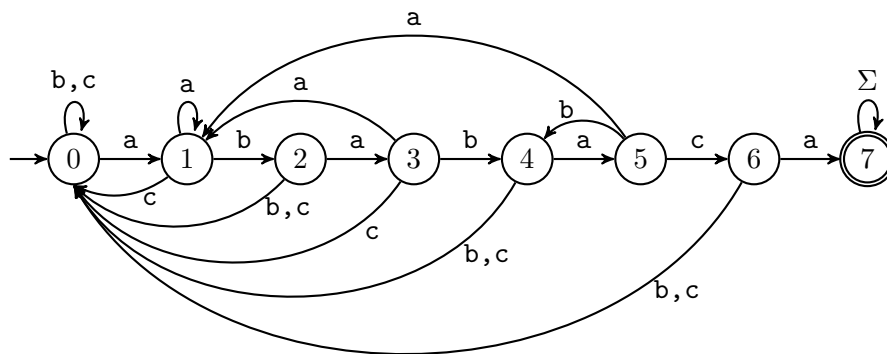


Figure 9.6: An equivalent deterministic finite automaton.

The DFA has been set up such that again that Observation 9.2 holds, i.e., reaching state j means that the j th prefix of P is a suffix of what we have parsed. But the DFA is deterministic, and so we can say more. Namely, the DFA has been set up so that we are always as far right as possible, or put in different words, j is maximal. Thus we have:

Observation 9.3. Assume we have reached state j during the parsing with the DFA. Then j is the length of the longest prefix of P that is a suffix of what we have parsed.

However, there are a few problems with pattern matching using a DFA:

- We need to store the transitions. Since each of the m states has up to $|\Sigma|$ outgoing transitions, we need to use $\Theta(m|\Sigma|)$ auxiliary space for this. This is perhaps acceptable since $|\Sigma|$ is a constant and m is not as huge as n , but we will soon see a method that needs fewer transitions.
- We need to find the transitions, i.e., we need to ask which state j' we need to go to if we find a mismatch while parsing character $T[i]$ and state j . Since we know what must hold at j' by Observation 9.3, it is not hard to compute it in polynomial time, but it is not easy to make this computation fast. (Section 9.3.6 will return to this.)

For these reasons we will not study pattern matching with a DFA further.

9.3.3 The KMP-automaton

We now describe a type of finite automaton that you have not encountered previously, and which blends the best of the NFA (few transitions) with the best of the DFA (uniqueness of parsing). Because this automaton is only used for the Knuth-Morris-Pratt pattern matching algorithm that we are building up to, it is called here a *KMP-automaton*.

The difference to a DFA is that a KMP-automaton allows one special type of transition, called a *failure-arc* (and denoted with symbol \times in the drawings). This acts as follows:

- It does not consume a character. (As such, it acts a lot like an ε -arc that you might have seen in cs241.)
- It may be used *only* if no other transition is possible (hence the name *failure*).
- Every state has at most one failure-arc.

Figure 9.7 shows the KMP-automaton that is equivalent to the earlier DFA in the sense that the meaning of reaching state j (Observation 9.3) is exactly the same.

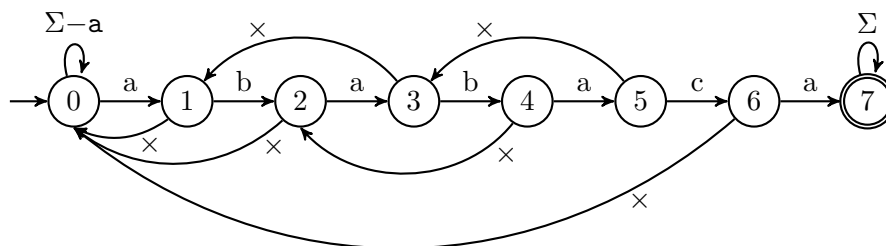


Figure 9.7: The equivalent KMP-automaton.

The KMP-automaton for pattern $P[0..m-1]$ hence consists of the following:

- States $0, \dots, m$, where 0 is the start-state and m is the unique accepting-state.
- *Forward-arcs* $j \rightarrow j+1$ that are labeled with $P[j]$.
- Loops at 0 and m , labeled with $\Sigma \setminus P[0]$ and Σ , respectively.

- One failure-arc for each state $1, \dots, m-1$.

With the exception of the failure-arc, all this information is immediately determined from P , so to describe the KMP-automaton, it suffices to list where the failure-arc of each state goes to. In our example, we have

state j	0	1	2	3	4	5	6
failure-arc to state	NA	0	0	1	2	3	0

The leftmost entry in this table is always NA (“not applicable”), because there is no failure-arc from state 0; all characters that are not $P[0]$ have an explicit transition-arc back to state 0. For this reason (and also because it will fit later computations well), the *failure-array* F that stores the failure-arcs will be shifted by one unit in the indexing:

Definition 9.1. *The failure array is an array F where $F[j]$ stores the target-state of the failure-arc from state $j+1$.*

In the example, we hence have:

state j	0	1	2	3	4	5	6
failure-arc to state	NA	0	0	1	2	3	0
$F[j]$	0	0	1	2	3	0	NA

9.3.4 Knuth-Morris-Pratt pattern matching

We now give the pattern matching algorithm by Knuth, Morris and Pratt, which essentially consists of building the KMP-automaton described above. However, there really is no need to build the automaton and its transition-function explicitly. Instead, the code to do the pattern-matching consists of direct for-loops that express appropriately where the current state is. The pseudo-code for this is in Algorithm 9.4, and assumes that we have the failure-array F . (Section 9.3.5 will discuss how to do find F .)

Figure 9.8 illustrates how this algorithm executes, using the same pattern and KMP-automaton that we have seen before (we repeat it here for convenience).

- On the leftmost guess, we find matching characters until we have reached state 5, i.e., we know that $P[0..4]$ matches $T[0..4]$.
 - The next character is a mismatch, so we follow the failure-arc from state 5, which brings us to state 3.
- In terms of the guess, this means that the guess has been shifted two units rightward (since $5 - 3 = 2$). Presuming that the failure-arc is correctly set up, we *know* that the first three characters of P match in this shift, because the failure-arc brought us to state 3, which promises that the first 3 characters of P are matched.
- We now use forward-arcs again to match further characters and get to state 4.

Algorithm 9.4: *KMP::patternMatching(T, P)*

```

1  $F \leftarrow \text{failureArray}(P)$ 
2  $i \leftarrow 0$                                 // index of current character of  $T$  to parse
3  $j \leftarrow 0$                                 // index of current state
4 while  $i < n$  do
5   if  $P[j] = T[i]$  then
6     if  $j = m - 1$  then                        // all characters were matched
7       return “found at guess  $i - m + 1$ ”
8     else                                    // go along forward-arc
9        $i \leftarrow i + 1$ 
10       $j \leftarrow j + 1$ 
11   else
12     if  $j > 0$  then                        // go along failure-arc from state  $j$ 
13        $j \leftarrow F[j - 1]$ 
14     else                                    // at leftmost state; go along loop
15        $i \leftarrow i + 1$ 
16 return FAIL

```

- Unfortunately the next character is **b** while we need an **a**. So we follow the failure-arc from state 4, which gets us to state 2 (shift the guess two units rightward). We *know* that the first two characters of P are matched.
- Unfortunately, **b** is a mis-match here as well. The failure-arc from state 2 gets us to state 0 (shift the guess two units rightward).
- Unfortunately, **b** is *still* a mis-match here. Since we are now at state 0, there is no failure-arc. We discard character **b** and move on to the next character. This means shifting the guess one unit rightward.
- The next character also doesn't match. Since we are in state 0, we discard it and move to the next character. Again this shifts the guess one unit rightward.
- Now we have a sequence of matches and found the pattern.

Summarizing, a failure-arc that goes k states leftwards means that we shift the guess by k units rightwards. Also, using the loop at state 0 means shifting the guess by one unit rightward.

Notice that we *never* compare a character $T[i]$ again once we have moved past it with our parsing. This suggests that the run-time should be linear in $|T|$. But also observe that we may compare $T[i]$ repeatedly when following failure-arcs, e.g. consider the character **b** in the middle of the example that was compared three times. How often might we compare character $T[i]$? A crude upper bound would be that this can happen at most m times (because we have $m - 1$ failure-arcs, and compare $T[i]$ again only if we have used one of them in the step before). But we can improve this bound with a better analysis.

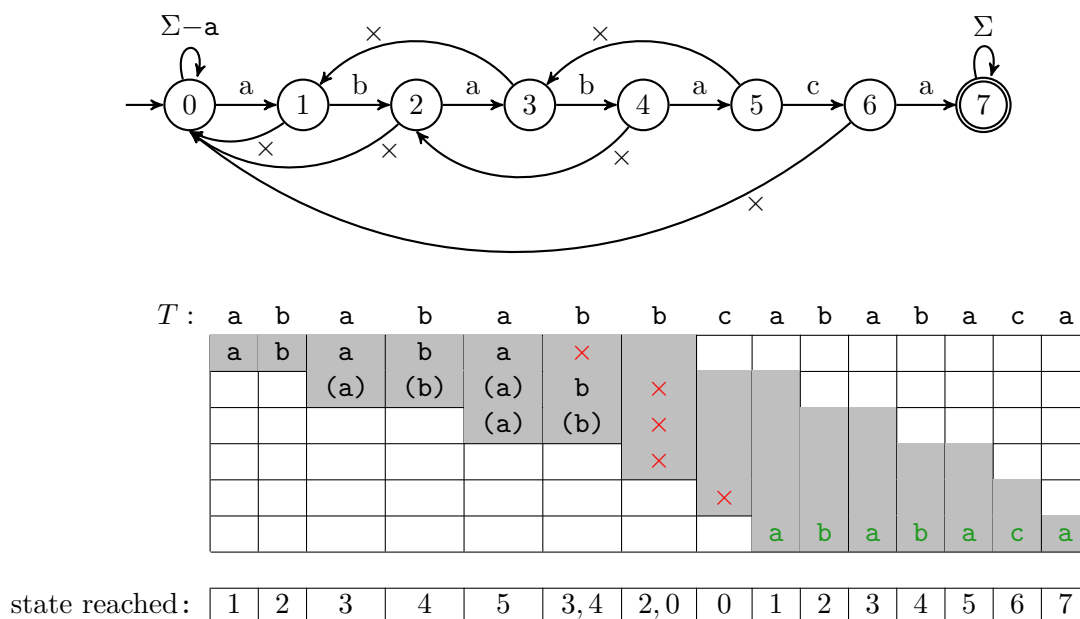


Figure 9.8: Searching for pattern $P = \text{ababaca}$.

Lemma 9.3. *The run-time of `KMP::patternMatching` (not counting the time for sub-routine `failureArray`) is in $O(n)$.*

We give three proofs of this; all three are basically the same idea but express it in different ways. As usual you need to read only the proof that your instructor used in class.

5906 *Proof.* Let us consider how often we walk along an arc during the execution of the algorithm.

- Every time we walk along a forward-arc or along the loop at state 0, we consume one character of T . So this happens at most n times.
- Every time we walk along the failure-arc, there must have been at least one forward-arc that brought us to the state from which we use the failure-arc. So the total number of used failure-arcs cannot be more than the total number of used forward-arcs, which is at most n .

5913 So in total we have followed at most $2n$ arcs and the run-time is $O(n)$.

5914 *Proof.* Define a function $\psi = 2i - j$ and observe how this changes throughout the execution of
5915 the algorithm. Initially we have $i = j = 0$ and hence $\psi = 0$. Now consider the four possible
5916 situations that can happen for each round of the while-loop:

- If we found P , then we are done.
 - If we go along a forward-arc, then both i, j increase, so ψ increases by $2 - 1 = 1$.
 - If we go along a failure-arc, then i is unchanged and j decreases, so ψ increases by at least 1.

- If we go along the loop at the leftmost state, then j is unchanged and i increases, so ψ increases by at least 2.

The algorithm can end in two possible ways. Either we found P , or we reached $i = n$. Either way we have $i \leq n$ and $j \geq 0$, and hence $\psi = 2i - j \leq 2n$.

Summarizing, for every execution of the while-loop ψ increases by at least one, and at the end it is at most $2n$. Therefore there can be at most $2n$ executions of the while-loop. Since each of them takes constant time, the run-time is $O(n)$. \square

Proof. (cs240e) Let us assume that time units are such that going along an arc takes at most one time unit. Define a potential function Φ to be j and observe that $\Phi = 0$ initially and $\Phi \geq 0$. Now compute the amortized cost of each operation:

- If we go along a forward-arc, then j increases, so $\Delta\Phi = \Phi^{\text{after}} - \Phi^{\text{before}} = 1$, and the amortized time is at most $1 + 1 = 2$.
- If we go along a failure-arc, then j decreases, so $\Delta\Phi \leq -1$, and the amortized time is at most $1 + (-1) = 0$.
- If we go along the loop at the leftmost state, then j is unchanged, so $\Delta\Phi = 0$ and the amortized time is at most $1 + 0 = 1$.

So the sum of the amortized run-times is at most $2n$, since going along a forward-arc or a loop consumes a character of T . This upper-bounds the actual run-time by definition of “amortized”, so the run-time is at most $2n \in O(n)$ time units. \square

9.3.5 Computing the failure array

We now fill in the missing piece, which is how to compute the failure-array $F[\cdot]$. We give two methods here; the first one is much more intuitive while the second one is much faster.

Recall that the entry $F[j]$ in the failure-array denotes the target-state of the failure-arc at state $j+1$. If we reached state $j+1$, then we have (by Observation 9.3) just seen $P[0..j]$, i.e., the last $j+1$ characters of the text equal $P[0..j]$.

T :						...matched $P[0..j]$...					
current guess					 $P[0..j]$	×				
shift by 1?					 $P[0..j-1]$					
shift by 2?					 $P[0..j-2]$...					

We would like to shift the guess forward, but also avoid all the guesses that are surely incorrect. Consider shifting the guess forward by one unit. This would align $P[1..j]$ (the characters in T corresponding to the guess that we just worked with) with $P[0..j-1]$ (the guess shifted over). So if $P[1..j] \neq P[0..j-1]$, then we can eliminate the guess obtained by shifting over by one unit. Note that this depends *only* on P and not on the text T ; in particular we can pre-compute this.

Generalizing, if $P[s..j]$ (for some $s \geq 1$) is different from $P[0..j-s]$, then we can eliminate the guess obtained by shifting over by s units. If $P[s..j] = P[0..j-s]$, then this guess might be a match, so then we should not eliminate that guess. We want to shift by the minimal amount

for which the guess could be correct, and hence shift the guess by the minimal amount s for which $P[s..j] = P[0..j-s]$. This corresponds to the failure-arc going left by s states, i.e., to state $j+1-s$.

Let us re-phrase this a bit to obtain something easier to memorize. We want the smallest $s \geq 1$ such that $P[s..j] = P[0..j-s]$. Equivalently

$$\begin{aligned}
 & \text{smallest } s \geq 1 \text{ such that } P[0..j-s] = P[s..j] \\
 \Leftrightarrow & \text{smallest } s \geq 1 \text{ such that } P[0..j-s] \text{ is a suffix of } P[1..j] \\
 \Leftrightarrow & \text{largest } \ell \leq j \text{ such that } \underbrace{P[0..\ell-1]}_{\ell \text{ chars}} \text{ is a suffix of } P[1..j] \\
 \text{substitute } \ell=j-s+1 & \\
 \Leftrightarrow & \text{length of the longest prefix of } P \text{ that is a suffix of } P[1..j]
 \end{aligned}$$

So we have:

Claim 9.1. *The target-state $F[j]$ of the failure-arc from state $j+1$ equals the length of the longest prefix of P that is a suffix of $P[1..j]$.*

With this, computing the failure-array $F[\cdot]$ is very easy in polynomial time: For each $j = 0, \dots, m-1$, compute all pre-fixes of P , compare them to $P[1..j]$, determine the longest prefix that is a suffix, and let $F[j]$ be its length. Table 9.9 illustrates this on the pattern $P = \text{ababaca}$ that has been our running example.¹

j	$P[1..j]$	Prefixes of P	longest prefix that is a suffix of $P[1..j]$	$F[j]$
0	Λ	$\underline{\Lambda}, \text{a}, \text{ab}, \text{aba}, \text{abab}, \text{ababa}, \dots$	Λ	0
1	b	$\underline{\Lambda}, \text{a}, \text{ab}, \text{aba}, \text{abab}, \text{ababa}, \dots$	Λ	0
2	b <u>a</u>	$\underline{\Lambda}, \underline{\text{a}}, \text{ab}, \text{aba}, \text{abab}, \text{ababa}, \dots$	a	1
3	b <u>a</u> b	$\underline{\Lambda}, \text{a}, \underline{\text{ab}}, \text{aba}, \text{abab}, \text{ababa}, \dots$	ab	2
4	b <u>a</u> a	$\underline{\Lambda}, \text{a}, \text{ab}, \underline{\text{aba}}, \text{abab}, \text{ababa}, \dots$	aba	3
5	b <u>a</u> b <u>a</u> c	$\underline{\Lambda}, \text{a}, \text{ab}, \text{aba}, \text{abab}, \text{ababa}, \dots$	Λ	0
6	b <u>a</u> b <u>a</u> c <u>a</u>	$\underline{\Lambda}, \underline{\text{a}}, \text{ab}, \text{aba}, \text{abab}, \text{ababa}, \dots$	a	1

Table 9.9: Computing the failure-function in polynomial time. The largest prefix that is a suffix is underlined.

For small examples (such as you would be expected to do on exams) this method is good enough. But for larger patterns, the run-time (which would be $\Theta(m^3)$) is too big. Luckily enough, there is a much faster method. Recall Observation 9.3 (renaming j to ℓ):

the number ℓ of the current state is the length of the longest prefix of P that is a suffix of what we have parsed

¹Note that we compute here $F[6] = F[m-1]$, which would correspond to a failure-arc from the accepting state (state 7). There is no such failure-arc in the KMP-automaton as presented, but having the value $F[m-1]$ is useful if we wanted to find not just one occurrence, but all occurrences of P in T . Details are left as an exercise.

5972 We can therefore re-phrase our definition of $F[j]$ as follows:

$F[j]$ = length ℓ of the longest prefix of P that is a suffix of $P[1..j]$
 = the number of the state that we reach if we parse $P[1..j]$ on the KMP-automaton for P

5973 This immediately suggests how to compute $F[j]$: simply parse $P[1..j]$. Moreover, if we next
 5974 want to compute $F[j+1]$, we do not need to re-start from scratch, because we have already
 5975 parsed $P[1..j]$ and so only need to parse $P[j+1]$ (from the state where we ended) to find $F[j+1]$.

5976 One thing to worry about: We need to compute the failure-array in order to find the KMP-
 5977 automaton for P , but now we are using this KMP-automaton to compute the failure-array.
 5978 This feels circular, why can we do this? The reason that this does not create trouble is that the
 5979 failure-array computation is “one step behind”. In order to find the failure-arc from state $j+1$,
 5980 we need to parse $P[1..j]$, which has j characters and so (at best) brings us to state j . So we
 5981 do not need to know what the failure-arc from state $j+1$ is in order to compute it, and there is
 5982 hence no circularity.

5983 Algorithm 9.5 gives the pseudo-code for computing the failure-array. Note its extreme simi-
 5984 larity with the ‘main’ Knuth-Morris-Pratt algorithm (Algorithm 9.4); the main difference is that
 5985 we parse P , rather than T , and start at character $P[1]$ rather than $T[0]$. (We also renamed $i \leftarrow j$
 5986 and $j \leftarrow \ell$ in the code to keep the notation corresponding to how we argued the correctness of
 5987 the failure-function above.) Figure 9.10 shows how this executes.

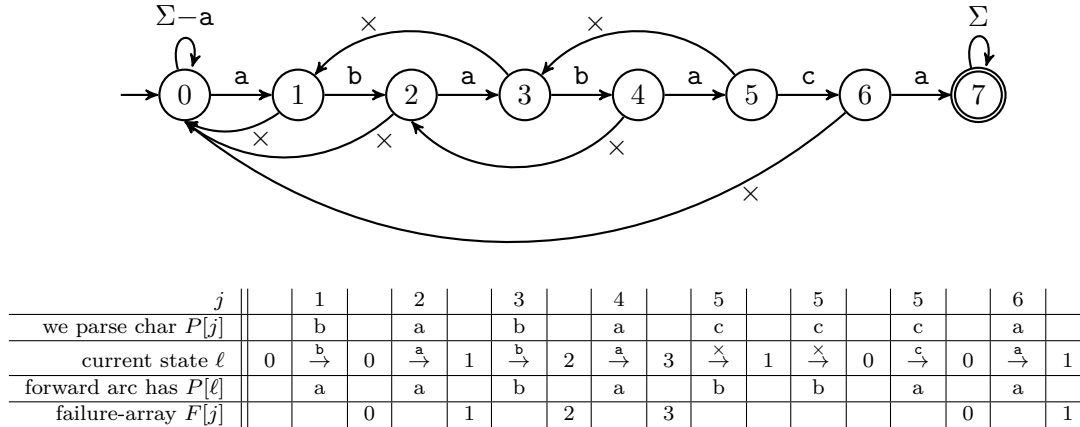
Algorithm 9.5: *KMP::failureArray(P)*

```

1  $F \leftarrow$  array of length  $m$ ; initialize  $F[0] \leftarrow 0$ 
2  $j \leftarrow 1$  // index of current character of  $P[1..m]$  to parse
3  $\ell \leftarrow 0$  // index of current state
4 while  $i < m$  do
5   if  $P[\ell] = P[j]$  then // go along forward-arc
6   |  $\ell \leftarrow \ell + 1$ 
7   |  $F[j] \leftarrow \ell$  // Have parsed  $P[1..j]$  and reached  $\ell$ 
8   |  $j \leftarrow j + 1$ 
9   else
10  | if  $\ell > 0$  then // go along failure-arc
11  | |  $\ell \leftarrow F[\ell - 1]$ 
12  | else // at leftmost state
13  | |  $F[j] \leftarrow 0$ 
14  | |  $j \leftarrow j + 1$ 

```

5988 With exactly the same argument as for the main routine, one shows that computing the
 5989 failure-array with this method takes $O(m)$ time because the parsed text has m characters.
 5990 Summarizing, the Knuth-Morris-Pratt algorithm takes $\Theta(n + m)$ time. Even if we only do a

Figure 9.10: We parse $P[1..m-1] = \text{babaca}$ to compute $F[0..m-1]$.

single execution of pattern-matching, this would be faster than brute-force, and if we do repeated searches for the same pattern, it would be even better since we do not need to re-compute the failure-array.

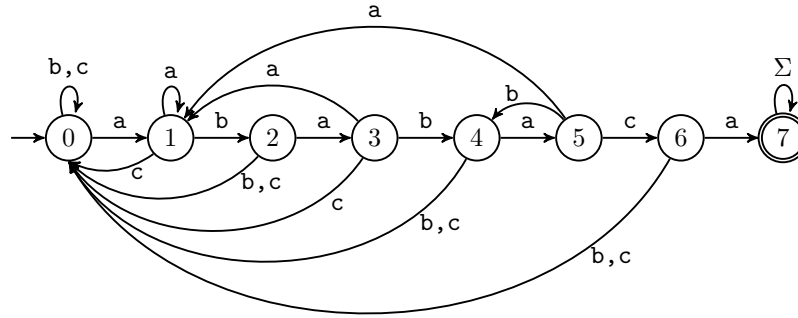
9.3.6 Related finite automata (cs240e)

Recall the deterministic finite automaton from Section 9.3.2, where we did not state how to compute the transition-function. The KMP-automaton implicitly contains all information needed for the DFA, except that it uses failure-arcs rather than explicit arcs for each mismatched character. Specifically, if $\delta(q, a)$ is the transition-function of the DFA, then

$\delta(q, c)$ = the state that we reach if start at q in the KMP-automaton,
 follow failure-arcs whenever the forward-arc is not labeled c ,
 and finally follow a forward-arc labeled c (or the loop at state 0).

With this definition it is clear that the transition-function for the DFA can be computed in polynomial time, given the KMP-failure-function. By organizing the computation from left to right (so that state j already has all arcs $\delta(j, c)$ when we compute them for states to the right of j), the time becomes $O(|\Sigma|m)$. Details are left as an exercise.

Second, one can observe that the KMP-failure-function can be improved! Consider the example of Figure 9.10. The failure-arc from state 4 is used if the next character in the text is *not* a. This failure-arc leads to state 2, but from here the forward-arc is again labeled with a. So if we fail at state 4, we will immediately afterwards fail at state 2, which brings us to state 0. We might as well have gone to state 0 directly and saved ourselves the comparison. Generally,

Figure 9.11: The DFA for `ababaca` revisited.

an improved failure-function $F^+[j]$ would be as follows:

$$F^+[j] = \begin{cases} \text{length } \ell \text{ of the longest prefix of } P \text{ that is a suffix of } P[1..j], \\ \quad \text{and where additionally } P[j+1] \neq P[\ell]. \\ 0 \text{ if there is no such } \ell. \end{cases}$$

6003 A different way to describe this is as follows: To find the F^+ -failure arc from state j , first follow
 6004 the F -failure arc from state j , say this gets us to state q . Check the forward-arcs at state j and
 6005 q . If they have different characters, then the F^+ -failure arc from j leads to q . It also leads to
 6006 q if $q = 0$, i.e., if we were at the leftmost state already. Otherwise ($q \neq 0$ and the two forward
 6007 arcs at j and q have the same character), the F^+ -failure-arc from j should lead to where the
 6008 one from q leads. Presuming we compute the F^+ -failure-arcs from left to right, and we had $F[\cdot]$
 6009 available already, one sees that we can hence compute the F^+ -failure arcs in constant time per
 6010 state, hence $O(m)$ time overall.

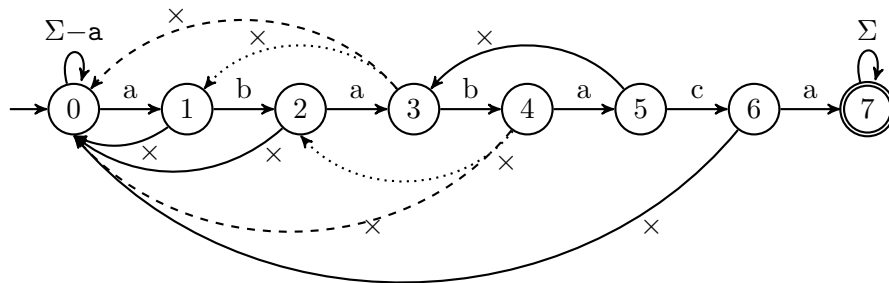


Figure 9.12: The KMP-automaton can be improved by using dashed failure-arcs, rather than dotted ones.

6011

9.4 Boyer-Moore algorithm

6012

6013

6014

6015

6016

6017

6018

6019

6020

The Knuth-Morris-Pratt algorithm achieves pattern matching in $O(m+n)$ time, which seems as good as one can hope for (surely one needs to look at all the characters at least once?) Actually, while it is good, there are *even* better approaches. To see why this is possible, consider searching for pattern `horn` in `cordelia`. If we compare the first guess as usual (from left-to-right), then after one comparison with a mismatch, we can shift the guess over by only one unit, since we know nothing about the text to the right of the mismatch. But if we compare *backwards*, then after one comparison with a mismatch, we can sometimes shift the guess much more! In this example, we saw character `d` in `T`, which doesn't occur anywhere in `P`, and so we can shift the guess all the way past where we compared.

6021

	c	o	r	d	e	l	i	a	
	h								

	c	o	r	d	e	l	i	a	
				n					

6022

6023

This is the crucial idea behind the *Boyer-Moore algorithm*, which has the following three ingredients:

- 6024
- 6025
- 6026
- 6027
- 6028
- 6029
- 6030
- *Reverse-order searching*: When comparing `P` to a guess, compare the characters from right to left, i.e., starting at `P[m−1]`.
 - *Last-occurrence heuristic* (also known as *bad-character heuristic*): When a mismatch occurs, then eliminate guesses based on the current character of `T`, i.e., consider where (if at all) this character occurs in `P`.
 - *Good-suffix heuristic* (also known as *suffix-skip heuristic*): When a mismatch occurs, then use the recently seen suffix of `P` to eliminate guesses.

6031

6032

6033

Both the bad character heuristic and the good suffix heuristic eliminate some guesses. We can hence use as location for the next guess to try the one that moves us forward more; all others have been ruled out by one heuristic or the other or both.

6034

6035

6036

6037

In practice, many characters of the text will therefore not be looked at at all, making the Boyer-Moore heuristic even faster than Knuth-Morris-Pratt in practice. Algorithm 9.6 gives the pseudo-code; the computation of the last-occurrence array and the good-suffix array will be discussed later.

6038

9.4.1 The last-occurrence heuristic

6039

6040

Let us see another example to understand the behavior of the last-occurrence heuristic better. In Figure 9.13, we are searching for the pattern `P = paper`.

- 6041
- 6042
- 6043
- 6044
- 6045
- As usual we start at the leftmost guess. We compare from the right, and immediately find a mismatch. The character in the text here is `a`.
 - We can rule out any guess that intersects this `a`, and where `P` does *not* have `a` in the corresponding place. So we shift the guess until `a` in `P` aligns with the `a` that we just found in text `T`.

Algorithm 9.6: *BoyerMoore::patternMatching*(T, P)

```

1  $L \leftarrow$  last occurrence array computed from  $P$ 
2  $S \leftarrow$  good suffix array computed from  $P$ 
3  $i \leftarrow m - 1$  // currently inspected character of  $T$ 
4  $j \leftarrow m - 1$  // currently inspected character of  $P$ 
5 while  $i < n$  and  $j \geq 0$  do
    // invariant: The current guess begins at  $T[i - j]$ 
6   if  $T[i] = P[j]$  then // match, go one character leftward
7      $i \leftarrow i - 1$ 
8      $j \leftarrow j - 1$ 
9   else // mismatch, find next guess to try
10     $i \leftarrow m - 1 + i - \min\{L[T[i]], j - 1, S[j]\}$ 
11     $j \leftarrow m - 1$ 
12 if  $j = -1$  then
13   return “found at guess  $i + 1$ ”
14 else
15   return FAIL

```

6046 • Again we compare from the right, and immediately find another mismatch. This time the
6047 character in T is **p**. There are two occurrences of **p** in P , giving us two possible guesses
6048 that we cannot rule out yet. Of those, we want to try the guess that is further left. This
6049 corresponds to taking the *rightmost* (or *last*) occurrence of **p** in pattern P and aligning it
6050 with the **t** that we just found in text T .

6051 • Again we compare from the right, and immediately find another mismatch. This time the
6052 character in T is **o**. There is no occurrence of **o** in P . With this, *all* guesses that intersect
6053 this **o** are ruled out, and we shift completely past this character in T .

6054 • Again we compare from the right, have one match and then another mismatch. This time
6055 the character in T is **r**. This occurs in P , but the occurrence is to the right of what we
6056 had already compared. Aligning P to this **r** would shift our guess leftward, but we never
6057 want to shift leftward (we have already ruled out all those guesses). So in this case the
6058 last-occurrence heuristic is not helpful; we simply shift the guess by one unit to the right.²

6059 So we need to know for any character $c \in \Sigma$ whether it occurs in P , and if so, where its
6060 the rightmost occurrence is. We store this information in the so-called *last-occurrence array* $L[\cdot]$
6061 which is indexed by the characters in the alphabet. In our example, it would be

²One could of course try to be smarter, and store more complicated information such as “if we see **r** and we are comparing $P[j]$ for $j < 4$ then shift entirely past this occurrence of **r**”. While this would result in fewer comparisons in practice, it means that the last-occurrence array would instead have to be a matrix with $|\Sigma|m$ entries and the pre-processing would be slower and more complicated.

P : p a p e r
 T : f e e d a l l p o o r p a r r o t s

Figure 9.13: Searching for pattern `paper` in text `feedallpoorparrots`.

chart c	p	a	e	r	all others
$L[c]$	2	1	3	4	-1

6062

6063 because \mathbf{p} occurs at $P[0]$ and $P[2]$ (and we take the larger of the two indices), and similarly for
6064 the other characters. It is not yet clear why we set $L[c] := -1$ if character c does not occur in
6065 P , but as we will see below, this allows us to do the update in multiple cases using one formula.
6066 Algorithm 9.7 gives the (extremely simple) method to compute the last-occurrence
6067 array in $O(|\Sigma| + m)$ time.

Algorithm 9.7: *BoyerMoore::lastOccurrenceArray(P)*

```

1  $L \leftarrow$  array indexed by alphabet  $\Sigma$ 
2 Initialize  $L$  to be  $-1$  everywhere
3 for  $i \leftarrow 0 \dots m-1$  do  $L[P[i]] \leftarrow i$ 
4 return  $L$ 

```

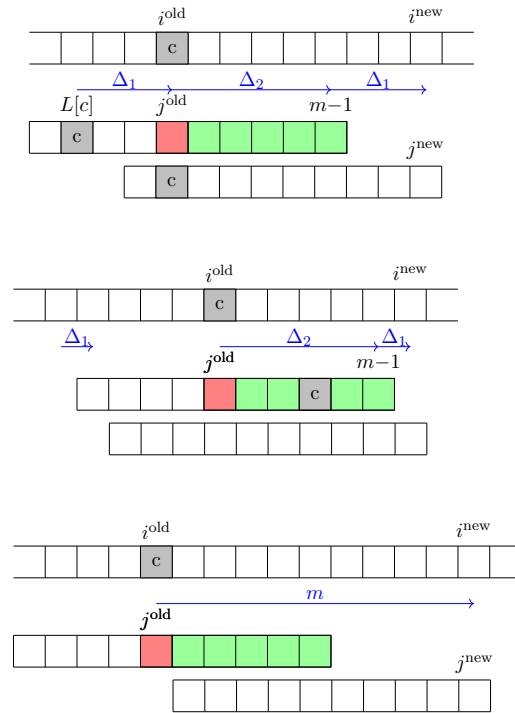
Updating i and j : With this definition of $L[\cdot]$ clarified, let us see why the updates to i and j in lines 10-11 of Algorithm 9.6 are correct. At any time during the algorithm, we compare $T[i]$ to $P[j]$. Consider the values i^{old} and j^{old} as they were when we found a mismatch. What should the values i^{new} and j^{new} be for the next guess?

Setting j^{new} is easy: We again start comparing from the right, and since j refers to the place within the pattern, we should set $j^{\text{new}} := m-1$ so that we are at the rightmost character of P . As for the update for i , we should have i^{new} point to the index in T that corresponds to $P[m-1]$ in the new guess. We have three cases:

6076 **Case 1:** Character $c := T[i^{\text{old}}]$ exists in P , and $L[c] < j^{\text{old}}$, i.e., the character is to the left of where we currently compared. Here we want to shift the guess forward by $\Delta_1 := j^{\text{old}} - L[c]$ units so that the two occurrences of c align. Let $\Delta_2 := m-1-j^{\text{old}}$ be the amount by which we had successfully compared. We then want $i^{\text{new}} = i^{\text{old}} + \Delta_1 + \Delta_2 = i^{\text{old}} + m-1-L[c]$.

Case 2: c exists in P but $L[c] > j^{\text{old}}$. This is the case where the last-occurrence heuristic is not helpful and instead we shift the guess rightward by one unit. So $\Delta_1 = 1$ while $\Delta_2 = m-1-j^{\text{old}}$ as before, and we want $i^{\text{new}} = i^{\text{old}} + m-1 - (j^{\text{old}}-1)$.

Case 3: c does not exist in P . We hence want to shift the guess completely past c . Since $j^{\text{new}} = m-1$, this means that the corresponding character in T is m units forward, i.e., $i^{\text{new}} = i^{\text{old}} + m$. If we set $L[c] := -1$ in this case, then the formula $i^{\text{new}} = i^{\text{old}} + m-1 - L[c]$ from Case 1 applies.



6077 All three cases can be unified into one formula $i^{\text{new}} = i^{\text{old}} + (m-1) - \min\{L[c], j^{\text{old}}-1\}$,
 6078 which is exactly the update that we did in line 10 (disregarding the update that will be done
 6079 from the good-suffix heuristic).

6080 9.4.2 Simplified Boyer-Moore

6081 The main loop for the Boyer-Moore algorithm and the code to compute the last-occurrence array
 6082 are both extremely simple. In contrast to this, the good-suffix array (which we have not yet seen)
 6083 will be quite complicated. (Actually, the code is not particular complicated, but understanding
 6084 why it works and not getting the indices incorrectly is complicated.) For this reason, in practice
 6085 a simplified version of Boyer-Moore is often used that uses *only* the last-occurrence array to
 6086 determine shifts forward.

6087 One can construct examples where the worst-case run-time of this simplified Boyer-Moore
 6088 algorithm is $\Theta(mn)$, i.e., where the algorithm performs worse than Knuth-Morris-Pratt. How-
 6089 ever, in practice even this simplified version performs exceedingly well. On a typical search in
 6090 English text, experiments have shown that only $\approx 25\%$ of the characters of the text are looked
 6091 at, making the method faster than Knuth-Morris-Pratt in practice.

9.4.3 The good-suffix heuristic (cs240e)

The idea for the good-suffix heuristic is very similar to the failure-arcs in the Knuth-Morris-Pratt algorithm: When we have matched part of pattern P , then the shift of the guess should be done in such a way that the new guess matches the characters of P that we have seen. However, the formulation is different because we now match backwards; we have hence seen a suffix of P .

Let us illustrate this on the example of the pattern $P = \text{boobobo}$ in Figure 9.14.

o	n	o	o	b	o	n	o	i	n	b	o	b	o	l	a	n	d
			b	o	b	o											
			(o)	(b)	(o)	b	o										
							(o)		o								
										o							
								(o)	b	o							
								(o)	(b)	(o)	b	o					
											(b)	(o)					

Figure 9.14: The good suffix heuristic explained on pattern $P = \text{boobobo}$.

- We initially try the leftmost guess, and correctly match the last three characters **obo**. But then we obtain a mismatch.
- We therefore must shift the guess over. We now let the amount of shifting be determined by the goal that **obo** must fit the new guess. In this case, shifting the guess two units over would satisfy this.
- Now we try matching the new guess. The rightmost character **o** matches, but then we get a mismatch. We shift over so that that the suffix **o** that we found matches the character **o**, which is two units.
- We immediately get a mismatch. We know nothing correct that would guide us for the good-suffix heuristic, and so shift over by one unit. (We actually could have shifted more using the last-occurrence heuristic, but this example was designed for explanation of the good-suffix heuristic and so we will ignore the last-occurrence heuristic here.)
- The next step gives another mismatch and we shift over another unit.
- In the next guess we match **bo** but get a mismatch at the character **o** before it. Suffix **bo** tell us to shift by two units.
- In the next guess we match **bobo** but get a mismatch at the character **o** before it. Note that this is the *same* **o** where we had the mismatch in the previous guess. It actually turns out that we could have avoided doing this guess, and deduced from the pattern alone that it cannot succeed. We will return to this a bit later.
- It is not possible to shift the guess to match **bobo**. If we can't get a full match, then try to match as much as possible of the suffix of P that we have seen. In our case (in the last row) we can at least match the last two characters of **bobo** with the first two character of

P , and so shift the guess to fit this. This gets us beyond the right end of the text, so we stop.

Observe that all the above operations *only* need to know the pattern P , they do not depend on the actual character seen in the text T . As such, we can pre-compute the *good-suffix array* S of size m that expresses the shift appropriately.

Defining S : To define S precisely, let us assume that we successfully matched $P[j+1..m-1]$ to the right end of the current guess, but had a mismatch at $P[j]$. (We assume here that $j < m-1$, i.e., we had at least one matched character. One can show that $S[m-1]$ should be set to be $m-2$ in order to shift the guess forward by one unit if we had an immediate mismatch.)

Use $Q := P[j+1..m-1]$ as a convenient shortcut for the matched string. We have three situations for determining the next guess (see also Figure 9.15):

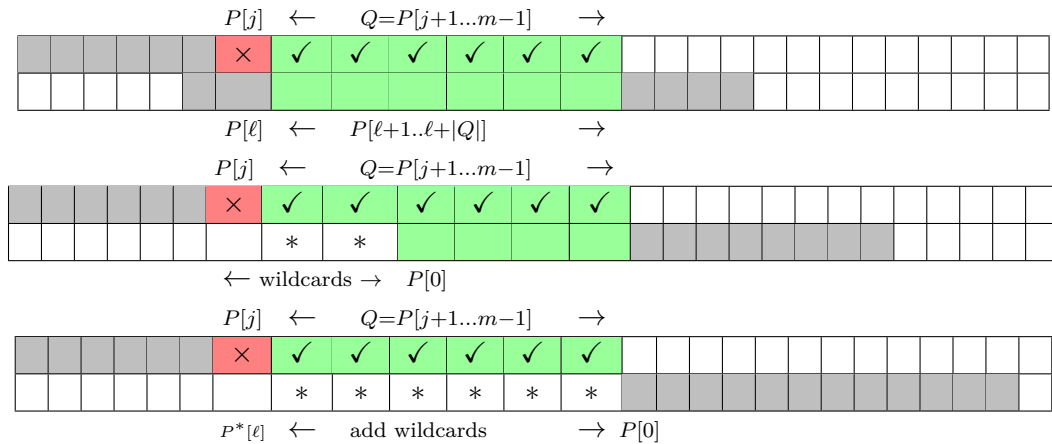


Figure 9.15: Three situations for defining $S[j]$. (Top) Q is a substring of P . (Middle) A suffix of Q is a prefix of P . (Bottom) No character of Q can be matched.

- If possible, we want to match all of Q in the new guess. This is possible if Q is a substring of $P[0..m-2]$. (We are using here $P[0..m-2]$, rather than $P[0..m-1]$, because Q is *always* a substring of $P[0..m-1]$, but using this would not shift us forward.)
- If Q is not a substring of $P[0..m-2]$, then find the longest suffix of Q that is a prefix of P , and shift so that they align. To express this more easily, we think of P as expanded leftward into negative indices. Specifically, define P^* to be an array with indices in $\{-m, -m+1, \dots, 0, \dots, m-1\}$ (exceptionally permitting negative indices) where $P^*[j] = P[j]$ for non-negative indices while $P^*[j] = *$ (a symbol for a *wildcard* that matches anything) if $j < 0$. Then Q is a substring of $P^*[-m..m-2]$.
- Finally it is possible that nothing of Q can be matched. This is really the same situation as the previous case, but the suffix of Q is now Λ . Since P^* was expanded sufficiently far, we again have that Q is a substring of P^* (using only wildcards).

In all three situations, Q is a substring of $P^*[-m..m-2]$, say it is a prefix of $P^*[\ell+1..m-2]$ for some (possibly negative) index ℓ . We want to shift the guess forward such that Q (in the old guess) aligns with $P^*[\ell+1..]$ (in the new guess). Since Q begins at $P[j+1]$, this means that we want to shift the guess forward by $\ell - j$. As for the last occurrence function, one verifies that we shift the guess forward by $\ell - j$ if and only if we set $S[j]$ to be this value ℓ . In other words,

$$S[j] = \max \{ \text{index } \ell: \underbrace{P[j+1..m-1]}_Q \text{ is a prefix of } P^*[\ell+1..m-1]. \}$$

6143 We should explain why ‘max’, i.e., what we should do if more than one possible index ℓ exists.
 6144 (In fact, due to the wildcards, there are always many indices ℓ where Q matches a suffix of P^* .)
 6145 As always, we want to shift by the least amount forward, so that we do not skip any guess that
 6146 might work. So we want to minimize $j - \ell$, and therefore maximize ℓ .

6147 **Computing S :** Figure 9.16 illustrates this definition of S , and also shows how to compute it on
 6148 small examples. Write down P^* (omitting the rightmost character). For each $j \in \{0, \dots, m-2\}$,
 6149 write down the string $P[j+1..m-1]$, and find its rightmost match in P^* . This gives us the
 6150 start-index $\ell + 1$ of the match, and subtracting one from it gives $S[j]$. Clearly this can be done
 6151 in polynomial time.

j	$P[j+1..m-1]$	$P^*[-m..m-2]$													match starts at index	$S[j]$
		-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5		
		*	*	*	*	*	*	*	b	o	o	b	o	b		
5	o												o		4	3
4	bo											b	o		3	2
3	obo										o	b	o		2	1
2	bobo						b	o	b	o					-2	-3
1	obobo					o	b	o	b	o					-3	-4
0	oobobo				o	o	b	o	b	o					-4	-5

Figure 9.16: Finding the good-suffix array S for pattern $P = \text{boobobo}$.

6152 But with a trick similar as for Knuth-Morris-Pratt we can actually compute $S[\cdot]$ in $O(m)$
 6153 time. To explain this, define for any string w the *reverse string* w^{rev} to be the characters of w
 6154 in reverse order. Now observe that

$$\begin{aligned} S[j] &= \max_{\ell} \{ P[j+1..m-1] \text{ is a prefix of } P^*[\ell+1..m-2] \} \\ &= \max_{\ell} \{ (P[j+1..m-1])^{\text{rev}} \text{ is a suffix of } (P^*[\ell+1..m-2])^{\text{rev}} \} \end{aligned}$$

So that we do not have to deal with reversals any longer, let us define R to be $(P^*)^{\text{rev}}$, i.e.,

$$R[j] = P^*[m-j-1] \text{ for } j = 0, \dots, 2m-1.$$

For example, for pattern $P = \text{boobobo}$ we would have $R = \text{oboboo}*****$. In general, R begins with P^{rev} and then has m wildcards attached. We will also use two substitutions to make the formula easier to read. Using $k = m - \ell - 2$, we have

$$\begin{aligned} S[j] &= \max_{\ell} \{ (P[j+1..m-1])^{\text{rev}} \text{ is a suffix of } (P^*[\ell+1..m-2])^{\text{rev}} \} \\ &= \max_{\ell} \{ R[0..m-j-2] \text{ is a suffix of } R[1..m-\ell-2] \} \\ &= m - 2 - \min_k \{ R[0..m-j-2] \text{ is a suffix of } R[1..k] \} \end{aligned}$$

and substituting $q = m - j - 1$ therefore

$$\begin{aligned} S[m-q-1] &= m - 2 - \min_k \{ R[0..q-1] \text{ is a suffix of } R[1..k] \} \\ &= m - 2 - \min_k \{ \text{the } q\text{th prefix of } R \text{ is a suffix of } R[1..k] \} \end{aligned}$$

This should really remind you of the analysis of the Knuth-Morris-Pratt failure function. In particular, we crucially need Observation 9.3, which states that we reach state q of a KMP-automaton (during the parsing of some text) if and only if the q th prefix of the pattern was a suffix of the parsed text. Therefore we can now reformulate this again:

$$\begin{aligned} S[m-q-1] &= m - 2 - \min_k \{ R[0..q-1] \text{ is a suffix of } R[1..k] \} \\ &= m - 2 - \min_k \{ \text{when parsing } R[1..k] \text{ on the KMP-automaton for } R, \text{ we reach state } q \} \end{aligned}$$

This is still not a formula that is easy to memorize, but observe that with it we can clearly compute S in $O(m)$ time. Namely, first compute R and build the KMP-automaton \mathcal{K}_R for it. Then parse $R[1..2m-1]$ on \mathcal{K}_R . (It actually suffices to do the parsing until we reach state $m-1$.) Whenever we reach a state $q \leq m-1$ for the first time, update $S[m-q-1]$ to be $m-2-k$, where k is the index of the last parsed character. Figure 9.17 shows how this operates on an example.

Since $|R| \leq 2m$, this takes $O(m)$ time total. With this we have finally shown:

Lemma 9.4. *The good-suffix array of pattern P can be computed in $O(m)$ time.*

Run-time of Boyer-Moore: So we have now shown how to compute the good-suffix array in $O(m)$ time. We also saw earlier that the last-occurrence array can be computed in $O(|\Sigma| + m)$ time. What is still open is how long the Boyer-Moore algorithm itself takes. This is non-trivial, since it is possible (see e.g. the fifth row in Figure 9.14) that characters of T are checked repeatedly against different shifts of P . One can avoid such repeat-tests with further modifications of Boyer-Moore; the details are fairly complicated and omitted here.

Is the good-suffix heuristic worth it? Maybe. The actual computation of the suffix-array is actually quite straightforward (no more difficult than the computation of the failure-function for Knuth-Morris-Pratt). But understanding why it works is quite complicated, and it is very

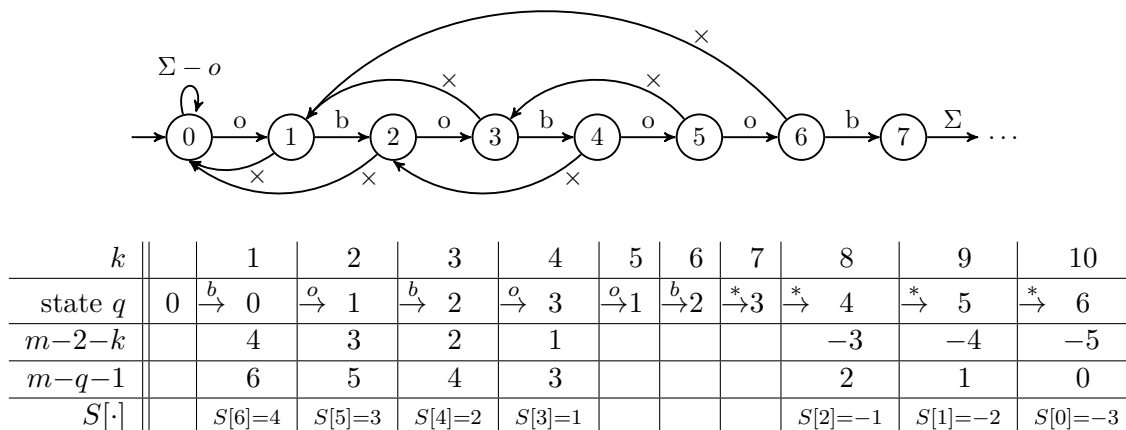


Figure 9.17: To compute the good-suffix array of $P = \text{boobobo}$, we build the KMP-automaton for $R = \text{oboboob *****}$ and then parse $R[1..k]$ for increasing k until state $m-1$ is reached.

easy to get indices wrong. Experiments have shown that adding the good-suffix heuristic on top of the last-occurrence heuristic adds comparatively little in terms of number of comparisons skipped, so in practice the simplified Boyer-Moore might well be good enough.

Further improvements: Recall that we improved the KMP-failure-function further, by redirecting failure-arcs if the state that they reached had the same outgoing forward-arc. In the same spirit, one can improve the good-suffix heuristic. Consider the example in the 5th row of Figure 9.14, where we matched *bobo* but failed to match *o*. This was actually completely predetermined! Consider the previous row. Here we also failed to match *o*, at the same location. We moved to a guess where the pattern P has *o* exactly at the place where we already *knew* that the text does not contain *o*. So this next guess was doomed to failure, and we could have skipped it right away. This can be expressed in the good-suffix array by demanding that in addition to the above conditions on ℓ we have $P[\ell] \neq P[j]$, and checking this during the parsing that computes $S[j]$. Details are left as an exercise.

9.5 Suffix trees and suffix arrays

In all the (non-trivial) pattern matchings algorithms that we saw so far, we pre-processed the pattern P , and then used the stored information to speed up the search. We now change tactics, and pre-process the text T , rather than the pattern. Recall that this is helpful in situations where we want to use the same text T for multiple patterns, such as when gene-searching in a DNA string or when searching for keywords in a (cached) web page. The key idea for pre-processing T is Observation 9.1: P is a substring of T if it is a prefix of some suffix of T . We

hence want to store (and search in) the suffixes of T in some form.

9.5.1 Trie of suffixes

So the simplest idea for preprocessing T is to store a trie with all the suffixes of T . See Figure 9.18.

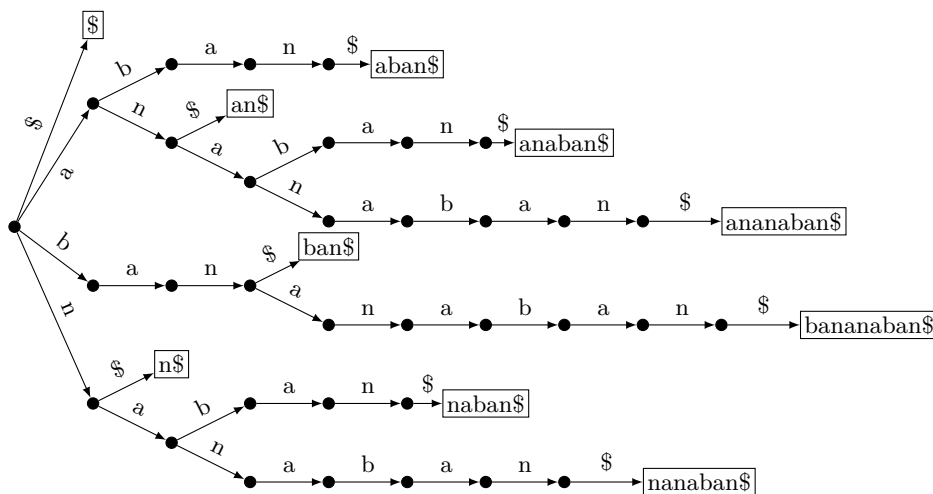


Figure 9.18: A trie that stores all suffixes of text $T = \text{bananaban}$.

Searching for a pattern P , given such a trie of suffixes, is extremely easy. Recall that we discussed (briefly) in Chapter 6 that tries support not only $\text{search}(x)$, but also $\text{prefixSearch}(x)$. This returns a positive answer if and only if x is a prefix of some word stored in the trie. Therefore $\text{prefixSearch}(P)$ will return successfully when applied onto the trie of suffixes if and only if P is a substring of T . Testing this takes $O(|P|)$ time (or perhaps $O(|P| \cdot |\Sigma|)$, depending on how the up to $|\Sigma|$ children at a node are stored).

However, there are some problems with the trie of suffixes. Text T has length n , and it has $n + 1$ suffixes; their total length is $1 + 2 + \dots + n \in \Theta(n^2)$. Therefore the trie of suffixes may use $\Omega(n^2)$ space and we would use $\Omega(n^2)$ time to build it. This is unacceptably large (recall that we assume n to be a *huge* number).

9.5.2 Suffix trees

We can reduce the space taken up by the data structure to store the tries with two tricks:

- First, recall that the words that are stored are all suffixes of one text T . There is no need to store the complete words explicitly at the leaves. It suffices to store some indication as to which suffix is stored here. (In our examples, we will write “ $T[i..n-1]$ ” to indicate the word while keeping the example somewhat more understandable; in real life one would only store the index i that identifies the suffix.)

- Second, recall that we studied *compressed tries* in Section 6.2.4. These omit all nodes that have only one child, at the price of needing to store an index at each node that identifies which character needs to be compared.

The data structure that incorporates both these ideas is called a *suffix tree* and illustrated in Figure 9.19.

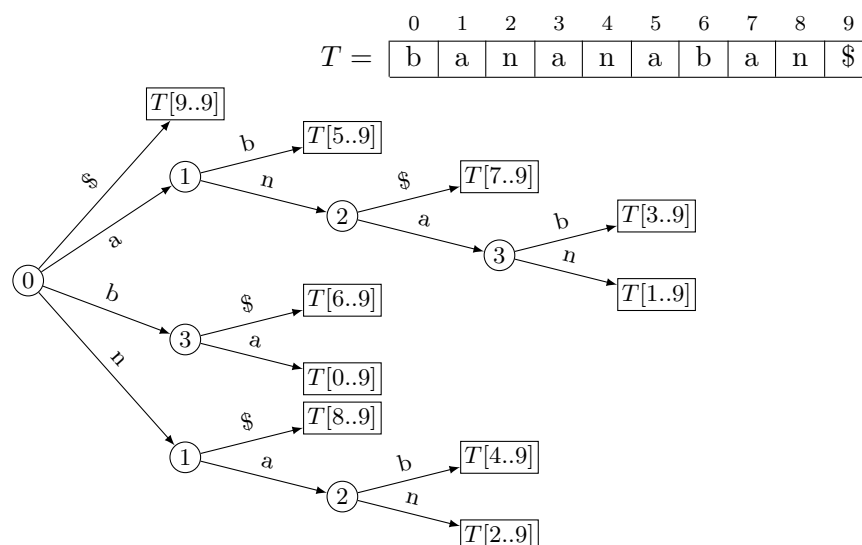


Figure 9.19: A suffix tree.

Observe that a suffix tree has $\Theta(n)$ leaves, one per suffix of text T . Also recall that a compressed trie has fewer interior nodes than leaves, so the entire suffix tree has $O(n)$ nodes. Each leaf takes $O(1)$ space, since it only stores the start-index of the corresponding suffix. Each interior node stores links to the children, which may take space $\Theta(|\Sigma|)$ depending on our choice of how to store these links. Hence the space of a suffix tree is $O(|\Sigma|n)$, and perhaps only $O(n)$.

Construction time: We mentioned (without giving details) that inserting a word w into a compressed trie can be done in $O(|w| + |\Sigma|)$ time. As such, it is straight-forward to build a suffix tree in $O(n(n + |\Sigma|))$ time, by explicitly inserting all suffices.

This is again too slow to be useful in practice, and much research has been spent on improving the run-time. There *are* ways to build suffix trees faster, and an algorithm has been developed that does so in $O(n|\Sigma|)$ time. The details are very complicated and will not be covered in cs240 (not even in the enriched section). Sometimes course cs482 (Computational Techniques in Biological Sequence Analysis) covers some of the ideas for this, so take this course if you are interested in construction (as well as numerous applications of) suffix trees.

Pattern matching: Doing pattern matching with a suffix tree is now the same as doing $prefixSearch(P)$ in a compressed trie. We did not give the details of this in Chapter 6, because they get a bit tedious. (We need to use leaf-references at nodes to get to characters that were ‘skipped’.) Overall, pattern matching with a suffix tree can be done in $O(m)$ time (or perhaps $O(m|\Sigma|)$ time depending on how child-references at each node are stored). While this sounds impressively fast, in practice the slow/complicated construction for suffix trees makes this inferior to the next data structure that we will see.

9.5.3 Suffix arrays

We now turn to a method of storing the suffixes of text T that is a bit slower than suffix trees in theory, but works better in practice both because it is easier to compute and because we use an array and avoid the space-overhead of suffix trees.

Definition 9.2. *The suffix array A^s of a text T is the sorting permutation of the suffixes of T with respect to lexicographic order.*

To illustrate this definition, let us consider the text **bananaban\$**. This has 10 suffixes (the end-of-word symbol is not part of the word, so the empty suffix is represented by \$), and we index each suffix by its start-index in T . See Figure 9.20. Now ³ Now sort these suffixes, and keep track of how the indices get permuted. Put differently, obtain not only the sorted order, but the sorting permutation that creates this order.

i		j	$A^s[j]$	
0	bananaban\$	0	9	\$
1	ananaban\$	1	5	aban\$
2	nanaban\$	2	7	an\$
3	anaban\$	3	3	anaban\$
4	naban\$	4	1	ananaban\$
5	aban\$	5	6	ban\$
6	ban\$	6	0	bananaban\$
7	an\$	7	8	n\$
8	n\$	8	4	naban\$
9	\$	9	2	nanaban\$

Figure 9.20: The suffixes for **bananaban\$** sorted.

In this example, we wrote down the sorting permutation *and* the actual suffixes. In the suffix array we only keep the sorting permutation, so for our example the suffix array is

³For pattern matching it would actually suffice to consider only the non-empty suffices, but suffix arrays are useful for other applications as well and so we include the empty suffix.

0	1	2	3	4	5	6	7	8	9
9	5	7	3	1	6	0	8	4	2

6258

6259 **Construction time:** Following the steps of its definition, it is clear that the suffix array
 6260 can be computed in $O(n^2 \log n)$ time: Compute the $\Theta(n)$ suffixes, and then apply any sorting
 6261 algorithm; since comparing two suffixes takes $O(n)$ time the run-time for this is $O(n^2 \log n)$.

6262 We can improve this run-time easily by remembering radix-sort. Pad the suffixes with
 6263 arbitrary characters at the end until they all have the same length. This does not affect the sorted
 6264 order due to the end-of-word character \$, which is smaller than all other characters. Now we can
 6265 view the suffixes as ‘numbers’ where the ‘digits’ are characters in Σ and all numbers have the
 6266 same number of digits. Applying radix-sort to sort them hence takes time $O(n(n+|\Sigma|)) = O(n^2)$.
 6267 In fact, if we use MSD-radix-sort then the run-time will frequently be better in practice, since
 6268 the suffixes will typically not have many leading characters in common and so not all characters
 6269 of all suffixes need to be compared.

6270 A theoretically faster way to find the suffix array A^s would be to find the suffix tree, and
 6271 to observe that the leaves in the suffix tree are sorted in lexicographic order from left to right
 6272 (presuming children are ordered according to the alphabet). Therefore, one can parse the suffix
 6273 tree, and set $A^s[j]$ to be the starting-index of the suffix at the j th leaf. In theory therefore
 6274 we can find the suffix array in $O(|\Sigma|n)$ time, but this requires the (complicated) algorithm for
 6275 constructing a suffix tree, which we wish to avoid.

6276 The amazing part of suffix arrays is that they can be constructed in linearithmic time *without*
 6277 needing to build the suffix tree. The idea is to modify MSD-radix-sort and exploit the fact that
 6278 the second $m/2$ characters of each suffix also occur as the first $m/2$ characters of some other
 6279 suffix. This makes it possible to cut the number of rounds needed by MSD-radix-sort to $O(\log n)$,
 6280 and therefore leads to a (not-too-complicated) algorithm that constructs a suffix array $O(n \log n)$
 6281 time. Details are typically covered in cs482 and so will not be given here.

6282 **Pattern matching:** It is very easy to do pattern matching in a suffix array. The array stores
 6283 suffixes (implicitly) in sorted order. Therefore we can apply binary search! The only difference
 6284 is that comparing two keys now means comparing two strings (pattern P and the suffix) and so
 6285 takes longer. (Also note that P may actually be longer than the suffix that it is being compared
 6286 to, but our version of *strcmp* (Algorithm 6.5) has been set up to handle this.) Algorithm 9.8
 6287 gives the code and Figure 9.21 gives an example. (In this example, we show the suffixes for the
 6288 ease of the reader doing the comparison, but notice that it is *not* needed to store them explicitly;
 6289 we can read the appropriate substring from T .)

6290 We need $O(\log n)$ comparisons for the search. Each comparison consists of doing *strcmp* and
 6291 hence takes $O(m)$ time. So the run-time for pattern matching is $O(m \log n)$ in the worst case,
 6292 which is a logarithmic factor slower than in suffix trees, but in practice well worth it due to the
 6293 space savings.⁴ Furthermore, often *strcmp* will use fewer than m comparisons, and quite often it

⁴The run-time can be reduced even further, to $O(m + \log n)$ time, by storing a second array. The details are

Algorithm 9.8: *suffixArray::patternMatching*(T, P)

```

1  $A^s \leftarrow$  suffix array of text  $T$  // this has length  $n + 1$ 
2  $\ell \leftarrow 0, r \leftarrow n$ 
3 while ( $\ell < r$ ) do
4    $\nu \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$  // Use  $\nu$  rather than  $m$  since  $m$  is used for  $|P|$ 
5    $i \leftarrow A^s[\nu]$  // index of the suffix to compare
6    $s \leftarrow \text{strcmp}(T, P, i, i+m-1)$ 
7   if ( $s = 0$ ) then return “found at guess  $T[i..i+m-1]$ ”
8   else if ( $s < 0$ ) then  $\ell \leftarrow \nu + 1$ 
9   else  $r \leftarrow \nu - 1$ 
10  $i \leftarrow A^s[\ell]$  // index of the suffix to compare
11 if  $\text{strcmp}(T, P, i, i+m-1) = 0$  then return “found at guess  $T[i..i+m-1]$ ”
12 else return “not found”

```

$\ell \rightarrow$	j	$A^s[j]$	
	0	9	\$
	1	5	aban\$
	2	7	an\$
	3	3	anaban\$
$\nu \rightarrow$	4	1	ananaban\$
	5	6	ban\$
	6	0	bananaban\$
	7	8	n\$
	8	4	naban\$
$r \rightarrow$	9	2	nanaban\$

$\ell \rightarrow$	j	$A^s[j]$	
	0	9	\$
	1	5	aban\$
	2	7	an\$
	3	3	anaban\$
	4	1	ananaban\$
$\ell \rightarrow$	5	6	ban\$
	6	0	bananaban\$
$\nu \rightarrow$	7	8	n\$
	8	4	naban\$
$r \rightarrow$	9	2	nanaban\$

$\nu=\ell \rightarrow$	j	$A^s[j]$	
	0	9	\$
	1	5	aban\$
	2	7	an\$
	3	3	anaban\$
	4	1	ananaban\$
$\nu=\ell \rightarrow$	5	6	ban\$ found
$r \rightarrow$	6	0	bananaban\$
	7	8	n\$
	8	4	naban\$
	9	2	nanaban\$

Figure 9.21: Search for ban using a suffix array.

will only use 1 or 2 comparisons, so in practice the run-time is closer to $O(\log n)$ and extremely fast.

9.6 Take-home messages

In this chapter, we studied a new problem (pattern matching), but many of the techniques that we have seen in previous chapters proved useful to do this efficiently. We have seen multiple methods, and they offer a trade-off between simplicity and run-time. Also, some of them preprocess the pattern while others pre-process the text. Table 9.22 summarizes the results.

A few other comments:

- We stopped our pattern-matching search as soon as we found one occurrence of pattern

non-trivial and beyond the scope of cs240.

	Brute- Force	Karp- Rabin	DFA	Knuth- Morris- Pratt	Boyer- Moore	Suffix Tree	Suffix Array
Preproc.	—	P	P	P	P	T	T
Preproc. time	—	$O(m)$	$O(m \Sigma)$	$O(m)$	$O(m+ \Sigma)$	$O(n^2 \Sigma)$ [$O(n \Sigma)$]	$O(n \log n)$ [$O(n)$]
Search time	$O(nm)$	$O(n+m)$ expected	$O(n)$	$O(n)$	$O(n)$ or better	$O(m)$	$O(m \log n)$ [$O(m + \log n)$]
Extra space	—	$O(1)$	$O(m \Sigma)$	$O(m)$	$O(m+ \Sigma)$	$O(n \Sigma)$	$O(n)$

Table 9.22: Worst-case run-times of methods for pattern-matching. Bounds in brackets are theoretically achievable, but quite complicated in practice.

P . Most algorithms can be adapted to find *all* occurrences within the same run-time.

- The best pattern-matching algorithm to apply depends on the situation. In particular one should consider the length of P , the size of the alphabet, and whether either P or T might get re-used for multiple pattern matching queries.
- There are many other problems that are a bit like pattern matching, and similar techniques can be applied. (Testing whether a word is a palindrome is one example.) In particular suffix trees have found many other applications in biological sequence analysis, for example for finding longest common substrings.
- A general programming principle to keep in mind is that spending some time on pre-processing data can be worth it overall, especially if there are repeated queries later that can re-use the same information from pre-processing.

9.7 Historical remarks

The order in which pattern matching were described here does not reflect the order in which they were discovered. According to the history as told in [KMP77], the earliest development (other than the obvious brute-force algorithm) was in 1969, when Morris developed string matching via finite automaton (he published it as a technical report together with Pratt in 1970). In 1970, Knuth independently came up with using finite automaton, and together with Pratt worked on making its run-time independent of the alphabet-size $|\Sigma|$. The three authors combined their efforts and published the final result in 1977 [KMP77].

As reported in the histories given in [KMP77, BM77], Boyer and Moore (and independently, Gosper) invented the idea of the Boyer-Moore algorithm in 1974, though it underwent some revisions and improvements until it was finally published in 1977 [BM77]. The Rabin-Karp algorithm was not discovered until a decade later by (obviously!) Rabin and Karp in 1987

[KR87].

Using the trie of suffixes for pattern matching was first proposed by Weiner in 1973 [Wei73]. Suffix trees were first introduced by McCreight [McC76], and increasingly faster algorithms and/or (slightly) simpler algorithms to construct it were developed over the next few years [Ukk95, Far97].

Suffix arrays are by far the newest contribution among the ones presented here, they were introduced by Manber and Myers in ([MM93], the conference-version was published in 1990), but apparently also discovered independently in 1987 [GBS92]. Numerous algorithms have been developed for constructing them efficiently in both theory and practice; see [PST07] for a survey.

Chapter 10

Text Compression

Contents

10.0.1 Detour: Streams	301
10.1 Character-by-character encoding	301
10.1.1 Encoding and decoding	303
10.1.2 Huffman’s algorithm to build a prefix-free trie	306
10.2 Multi-character encodings	313
10.2.1 Run-Length Encoding	314
10.2.2 Lempel-Ziv-Welch	318
10.3 bzip2	325
10.3.1 Huffman encoding	325
10.3.2 Modified run-length encoding	325
10.3.3 Move-to-front transform	327
10.3.4 Burrows-Wheeler Transform	328
10.3.5 bzip2 discussion	334
10.4 Take-home messages	334
10.5 Historical remarks	335

We now turn to the topic of *text compression*. You have undoubtedly used this frequently—**gzip**, **bzip2**, and **GIF** are all examples of text compression methods that are used whenever we would like to make files smaller without losing information.

In the text compression problem, we want to do a *compression* or *encoding*, which is a mapping

$$S \longrightarrow C$$

where S is the *source text* (over some alphabet Σ_S) and C is the *code-text* or *compressed text* (over some alphabet Σ_C). We are using ‘text’ here, rather than ‘word’, to emphasize that control-characters such as spaces or periods might well be included. The source-alphabet Σ_S may or may not be the same as the code-alphabet Σ_C but (as for all alphabets) we assume that it has at least two characters. To keep things simple we will often assume that the code-alphabet

Σ_C is $\{0, 1\}$, i.e., we want to convert the source-text into a bit-string.

There are different kinds of compressions:

- **Logical vs. Physical Compression.** In *physical compression*, the algorithm to encode S acts directly on S , without any knowledge what S represents. In particular, it purely considers the bits/digits/characters of S , and works equally whether these are pixels of a picture or video, a sound file or characters of English text. In contrast to this, *logical compression* knows more about what the input string represents, e.g., whether it is a picture, a movie, music, etc., and takes advantage of this to compress better. For example, converting a picture into a quad-tree as in Figure 8.7 is an example of logical compression.
- **Lossy vs. Lossless Compression.** In *lossless compression*, the coded text C must contain *all* the information that was in the input text S . Put differently, we must be able to *decode* text C and recover *exactly* the text S .

In contrast to this, *lossy compression* permits that some information is lost during encoding. There are some situations where lossy compression is good enough. For example, consider the video-stream during e-meetings, which compresses the input from your camera before sending it over the internet to other participants. There is (usually) no need for these videos to be perfect, and so lossy compression is used to achieve a smaller output (hence use less bandwidth).

We study here only physical lossless compression, which is applicable in any situation.

The prime objective of text compression is to make the coded text short (hence the name “compression”). Formally, we measure this by taking the ratio between the length of S and the length of C . However, to make things fair, we must consider the size of the alphabets as well—if Σ_S has 256 characters while $\Sigma_C = \{0, 1\}$, then a single character of Σ_S carries much more information, and so simply comparing the lengths would be comparing apples and oranges. To make things fair, we proceed as follows: If $|\Sigma_S| = 2^k$, then we could convert S to a bitstring by somehow mapping Σ_S to bitstrings of length k , and then mapping each character of S to the corresponding bitstrings. Thus we could represent S as a bitstring of length $|S| \log |\Sigma_S|$.¹ We do the same thing for the coded text C (though usually we assume that $\Sigma_C = \{0, 1\}$, i.e., C is a bitstring right away. Summarizing, we measure the

$$\text{compression ratio} := \frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}.$$

For cs240, achieving a small compression ratio is the main objective, but we occasionally also consider other goals such as the speed of encoding S or decoding C . Unfortunately, we cannot hope to *always* get a good compression ratio.

Lemma 10.1. *For any physical lossless encoding scheme, there exists a text where the compression ratio is at least 1.*

¹If $|\Sigma_S|$ is not a power of 2 then we would need more bits, so perhaps one should have used $|S| \lceil \log |\Sigma_S| \rceil$ to define the compression ratio. But for most alphabets (e.g. ASCII) the size is a power of 2, so to keep things simple we will ignore this rounding.

Proof. (cs240e) We will give the proof only $\Sigma_S = \Sigma_C = \{0, 1\}$ (i.e., both S and C are bit-strings); it can be generalized. Assume for contradiction that there exists an encoding scheme \mathcal{A} that always compresses, so in particular it compresses all bit-strings of length n . Hence

$$\mathcal{A} : \{\text{bit-strings of length } n\} \rightarrow \{\text{bit-strings of length } \leq n - 1\}.$$

But there are 2^n bit-strings of length n , but only $1 + 2 + \dots + 2^{n-1} = 2^n - 1$ bit-strings of length at most $n - 1$. This means that there are two bit-strings w, w' of length n that are mapped by \mathcal{A} to the same code-text x . But then \mathcal{A} is not lossless: Given x we cannot decide whether this stood for w or w' . Contradiction. \square

As such, we cannot give any good theoretical bounds for how well the various compression scheme work, and the evaluations in this chapter are predominantly done by experiments.

One disclaimer: The words ‘encoding’ and ‘coded text’ may make you think that one of our objectives might be to encrypt the text, i.e., to make it difficult to decode C unless one has some additional information. While this is a very interesting and deep topic, it is beyond the scope for cs240; we will not consider here how much a compression method obscures the text.

10.0.1 Detour: Streams

Text that should be compressed is often so huge that it does not fit in the computer’s memory and/or it may not all be available yet when we begin compressing it (e.g., it could still be loading from the internet). For this reason, we will often assume that the input and output to our encoding/decoding algorithms are *streams*. Formally, an *input-stream* is similar to a stack (we can *top* and *pop*) while an output-stream is much like a queue (we can *append*), and both support *isEmpty*. On some occasions we will also assume that we can *reset* a stream, i.e., return it to the state that it was when we originally got it.

We use the word stream (rather than a stack/queue) to emphasize the transient nature of the input/output: We can look at the current item, but we *not* look at any items that we saw some time ago (unless we store them in a separate variable), and we can *not* look at any items that will come in the future. As we will see in Chapter 11, any algorithm that uses streams for the input and output can deal well with data that does not fit into the computer memory. Most (but not all) algorithms for text compression will be able to use streams.

10.1 Character-by-character encoding

The simplest method to encode a text S is *character-by-character encoding* (or *single-character encoding*) defined as follows. We have some fixed method of mapping each character in the source-alphabet to a *code-word*, which is a string of characters in Σ_C . Recall that Σ_C^* denotes the set of all words over alphabet Σ_C ; we can hence describe a character-by-character encoding as a function $E : \Sigma_S \rightarrow \Sigma_C^*$. See Figure 10.2.

Let us consider some historical examples:

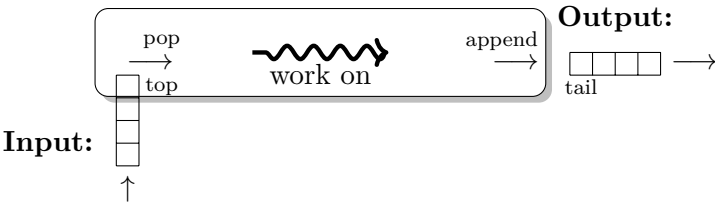


Figure 10.1: When using streams, we can only look at the topmost element of the input.

$c \in \Sigma_S$	\square	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

Figure 10.2: A simple example of character-by-character encoding.

- **Example 1:** Caesar code. This method, supposedly used by Julius Caesar, was one of the earliest method used to obscure a text (though it is actually very easy to decipher). The idea is to replace each letter by the letter that comes k units later in the alphabet (cyclically shifting at the end of the alphabet). So the offset-1 Caesar code would map $A \rightarrow B, B \rightarrow C, \dots, Z \rightarrow A$ and for example

$$APPLE \rightarrow BQQMF$$

6420 This does not compress at all (i.e., the compression ratio is exactly 1) because we use
6421 exactly as many characters in the coded text as we had in the input text and the alphabets
6422 are the same.

- **Example 2:** Baudot code. In the 1870s, a code was invented that maps the English upper-case characters (and some punctuation and control characters) to 5-bit strings:

$$A \rightarrow +---- \quad B \rightarrow ---+-, \quad \text{etc.}$$

6423 The output was then hole-punched into width-5 tapes that could easily be processed and
6424 stored—the earliest precursors of punch cards.

6425 This does not compress the text (we use 5 bits for the $26 < 2^5$ English characters), but
6426 it was a popular means of transmitting information well into the 1960s and 1970s, and
6427 inspired ASCII (which was invented in 1963).

6428 One could also consider ASCII itself to be a character-by-character encoding, since it maps
6429 an alphabet of 128 characters and control-characters onto length-7 bit-strings. However,
6430 how would you describe “characters”, what alphabet is this? ASCII (its name stands
6431 for ‘American Standard Code for Information Interchange’) is by definition the way to
6432 describe “character” in a computer. Therefore we usually treat ASCII as alphabet, rather
6433 than as an encoding scheme.

- **Example 3:** Morse code. Even older than Baudot codes is the Morse code, designed
6434 when electrical telegraphs were first invented (in the 1830s). It maps common characters
6435

(and also numbers and punctuation, not shown here) into a sequence of dots and dashes.
See Figure 10.3.

A	• —	H	• • • •	N	— •	U	• • —
B	— • • •	I	• •	O	— — —	V	• • • —
C	— • — •	J	• — — —	P	• — — •	W	• — —
D	— • •	K	— • —	Q	— — • —	X	— • • —
E	•	L	— • •	R	— • •	Y	— • — —
F	• • — •	M	— —	S	• • •	Z	— — • •
G	— — •			T	—		

Figure 10.3: Morse code as a table.

The Caesar-shift method, Baudot-code and ASCII are all *fixed-length encodings*: all code-words have same length. In contrast to this, Morse code uses shorter codewords for frequently used characters (such as ‘e’ and ‘t’, see also Table 10.4). This is called a *variable-length encoding* and should lead to much shorter encodings (i.e., a better compression ratio).

e	12.70%	d	4.25%	p	1.93%
t	9.06%	l	4.03%	b	1.49%
a	8.17%	c	2.78%	v	0.98%
o	7.51%	u	2.76%	k	0.77%
i	6.97%	m	2.41%	j	0.15%
n	6.75%	w	2.36%	x	0.15%
s	6.33%	f	2.23%	q	0.10%
h	6.09%	g	2.02%	z	0.07%
r	5.99%	y	1.97%		

Table 10.4: Character frequencies in typical English text.

There are other examples of variable-length code in real life; for example UTF-8 is a method of encoding over a million characters (the so-called Unicode characters) that extends ASCII. Each code-word is between 1 and 4 bytes long (here 1 byte is 8 bit). The 1-byte code-words are the ASCII characters, and more bytes are used for other (less frequently used) characters. Virtually all characters in common use can be expressed with three bytes; the fourth byte is used for mathematical symbols, emojis and other rarities.

10.1.1 Encoding and decoding

For any character-by-character encoding scheme $E : \Sigma_S \rightarrow \Sigma_C^*$, doing the encoding is very simple, because all we need is a dictionary where the keys are characters of Σ_S and the values are the code-words. (If Σ_S naturally maps to integers, then we can store this dictionary in an array E indexed by Σ_S .) The encoding-algorithm then simply consists of looking up each character’s code-word in turn.

Algorithm 10.1: *charByChar::encoding(S, C)*

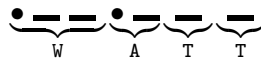
Input : input-stream S , output-stream C
 // We assume the class stores an encoding dictionary E

```

1 while  $S$  is not empty do
2    $c \leftarrow S.pop()$ 
3    $x \leftarrow E.search(c)$ 
4    $C.append(x)$ 

```

To see an example, the text WATT would be encoded with Morse code² as



Now we turn to the problem of decoding. This is very easy if we store the encoding in an *encoding trie*: a multi-way trie where links are labeled with the characters of the alphabet and each node v store the character whose codeword is the string on the path from the root to v . See Figure 10.5.

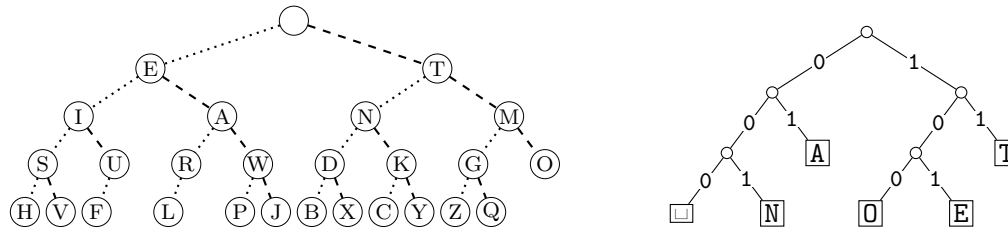


Figure 10.5: Encoding tries for the Morse code and for the example from Figure 10.2.

Decoding is extremely easy once we have the encoding trie: Parse the coded text C in the encoding trie, output the corresponding character and repeat. For example, if we decoded 111000001010111 in the trie in Figure 10.5, we would get

$$\underbrace{11}_T \underbrace{100}_O \underbrace{000}_\square \underbrace{101}_E \underbrace{01}_A \underbrace{11}_T .$$

However, this does not work for Morse code! For example:

$$\bullet \text{---} \bullet \text{---} \text{---} = \underbrace{\bullet \text{---}}_W \underbrace{\bullet \text{---}}_A \underbrace{\text{---}}_T \underbrace{\text{---}}_T = \underbrace{\bullet \text{---}}_A \underbrace{\bullet \text{---}}_N \underbrace{\text{---}}_O$$

We do not know (while parsing the input string) whether we should output the current character

²This is not quite true as we will see in a moment.

or continue parsing! Put differently, the decoding is *ambiguous* and therefore Morse code as presented is lossy.³

To ensure that our character-by-character encoding is lossless, characters should *only* be at leaves of the encoding trie. Put differently, the set of codewords should be *prefix-free*, i.e., no code-word $E(c)$ is a prefix of another code-word $E(c')$. (Confusingly, this is sometimes called a *prefix-code* in the literature, though it should really be called a *prefix-free code*.) Conveniently a prefix-free code guarantees one of our conditions for binary tries (“no stored word is a prefix of another”), even without end-of-word symbols.

A prefix-free code guarantees unambiguous decoding, because all characters are at leaves, and so we output a character (during the decoding) whenever we reach a leaf. Algorithm 10.2 gives the pseudo-code for decoding (again using streams). The run-time for decoding is $O(|C|)$, presuming alphabet Σ_C has constant length.

Algorithm 10.2: *prefixFree::decoding(C, S)*

Input : input-stream C , output-stream S
 // We assume the class stores a prefix-free encoding trie T

```

1 while  $C$  is not empty do
2    $r \leftarrow T.root$ 
3   while  $r$  is not a leaf do
4     if  $C$  is empty or  $r$  has no child labeled with  $C.top()$  then
5       return “invalid encoding”
6     else
7        $r \leftarrow$  child of  $r$  that is labeled with  $C.pop()$ 
8    $S.append(\text{character stored at } r)$ 
```

Let us briefly return to encoding, where we did not analyze the run-time yet. If the code is stored in a table, then we spend $O(1)$ per character c of the source-text S to look up the table-entry, and then $O(|E(c)|)$ time (the length of the code-word) to write the encoding of c to the output. As such, the run-time for encoding is $O(|S| + |C|)$, which is in $O(|C|)$ since $|S| \leq |C|$.

It actually turns out that it is better *not* to store the code-words in an array directly. The reason is that some of the code-words may be quite long. Instead it is better to store the code-words as references to leaves in the encoding trie, see Figure 10.6. The space used is then asymptotically no more than the space for the encoding trie, which we may assume to be $O(|\Sigma_S|)$ since we should never use an encoding trie where a node has exactly one child. The encoding algorithm gets marginally more complicated because first we must create the links to the leaves (this takes $O(|\Sigma_S|)$ time). Then, in order to obtain a code-word, we must go back up from the

³Actually, Morse code is *not* lossy, but the way it is used is more complicated than what we have described here. Morse code also uses a ‘pause’ to signal the end of a code-word, and a double-pause to indicate the end of a word. Effectively, Morse code has a three-character alphabet, dot, dash and pause.

6482 corresponding leaf until the root; this gives the code-word in reverse order. But the run-time is
6483 still $O(|S| + |C|) = O(|C|)$ for the decoding. See Algorithm 10.3 for the code.

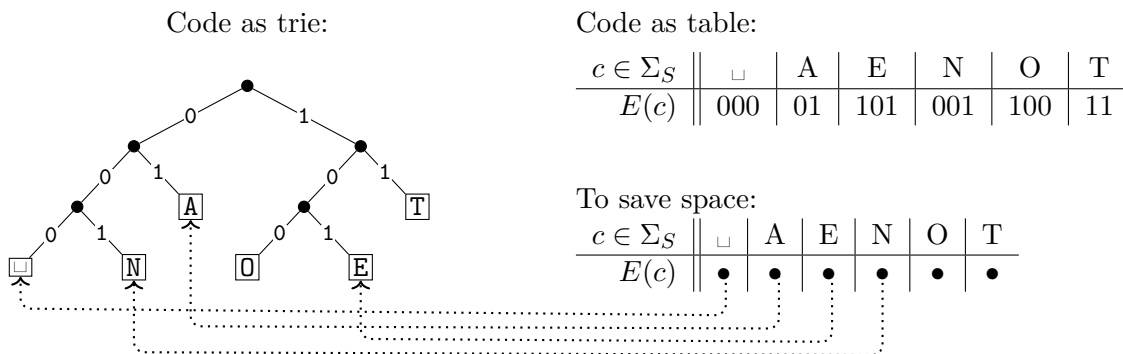


Figure 10.6: The encoding trie with corresponding encoding table, and how to save space by storing links to leaves instead (not all links are shown).

Algorithm 10.3: *prefixFree::encoding(S, C)*

```

// We assume the class stores a prefix-free encoding trie  $T$ 
1 Initialize encoding-array  $E$  indexed by characters of  $\Sigma_S$ 
2 forall leaves  $\ell$  in  $T$  do  $E[\text{character at } \ell] \leftarrow \ell$ 
3 while  $S$  is non-empty do
4    $v \leftarrow E[S.pop()]$  // Find code-word  $w$  by going up from leaf  $v$ .
5    $w \leftarrow$  empty string
6   while  $v$  is not the root do
7      $w.prepend(\text{character on link from } v \text{ to its parent})$ 
8      $v \leftarrow$  parent of  $v$ 
9    $C.append(w)$ 

```

6484 **10.1.2** Huffman's algorithm to build a prefix-free trie

Any prefix-free code can be used for lossless encoding, but as discussed earlier, the idea is to use short code-words for frequently used characters in the hopes of achieving a good compression ratio.

Let us first study what the length of the coded text C is, given a source text S and an encoding-trie T . For each $c \in \Sigma_S$, let $f(c)$ be the *frequency* with which character c occurs in source text S . Let $d(c)$ be the depth of the leaf in the encoding-trie T that stores character c . Note that the code-word for c has length exactly $d(c)$, since it has one character for each link

on the path to the root. Since the coded text C is obtained by concatenating the code-words, we hence have ⁴

$$|C| = \sum_{c \in \Sigma_S} f(c) \cdot d(c) = \sum_{c \in \Sigma_S} (\text{frequency of } c) \cdot (\text{depth of } c \text{ in encoding trie } T).$$

We have no control over the frequencies (they are fixed since the source text S is given), but we have control over the depth of the leaves assigned to characters. As such, the length of the coded text directly depends on the choice of T ; we also write $\text{cost}(T)$ for the length of the coded text obtained for some trie T .

As the formula suggests, characters with high frequency should have small depth to keep the cost down. This insight led Shannon and Fano to try to build such a code by assigning short code-words to frequent characters. However, it turns out (and this was discovered by Huffman when he was an undergraduate student) that a better idea is to turn the viewpoint around: characters with small frequency should have larger depth.

The algorithm developed by Huffman hence finds characters of *small* frequency first. The goal is that these get pushed towards the bottom of the encoding trie. However, we will not explicitly find a place for the low-frequency items. Instead we slowly build up the encoding trie by adding the interior nodes (we know that the leaves store the characters).

Formally, we keep at all times a set of tries that all should become sub-tries of the final encoding trie. Initially each character forms a single-node trie (it is stored in the unique leaf). Each trie T has an associated frequency $f(T)$, which denotes how often the root of T would get accessed during the encoding of the source text. Therefore for a single-node trie T_c that stores character c , the associated frequency $f(T_c)$ should be $f(c)$, because we will access this leaf $f(c)$ times.

Let us consider an example, illustrated in Figure 10.7. We use here the source text **GREENENERGY** with source alphabet $\Sigma_S = \{G, R, E, N, Y\}$. We can easily compute the frequency of each character in S , and therefore initialize our set of tries with five tries (one per character) and the corresponding frequencies.

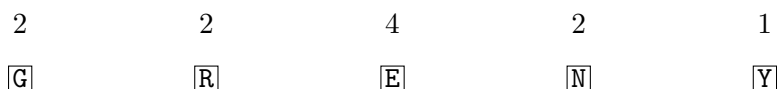


Figure 10.7: Setup to find the Huffman trie.

Now as illustrated in Figure 10.8, repeatedly take the two tries T, T' of smallest frequencies, and make them sub-tries of a newly created interior node z . What frequency should we assign

⁴(cs240e) This formula should remind you a bit of of the one for optimal static binary search trees (Section 5.2.2). But there is a noticeable difference: in a binary search tree the choice of the root determines which other values are left/right, whereas in the encoding trie there is no particular order among the characters and we can freely choose where to put them. As such, the two algorithms to find ‘trees that are optimal in some sense’ are not related.

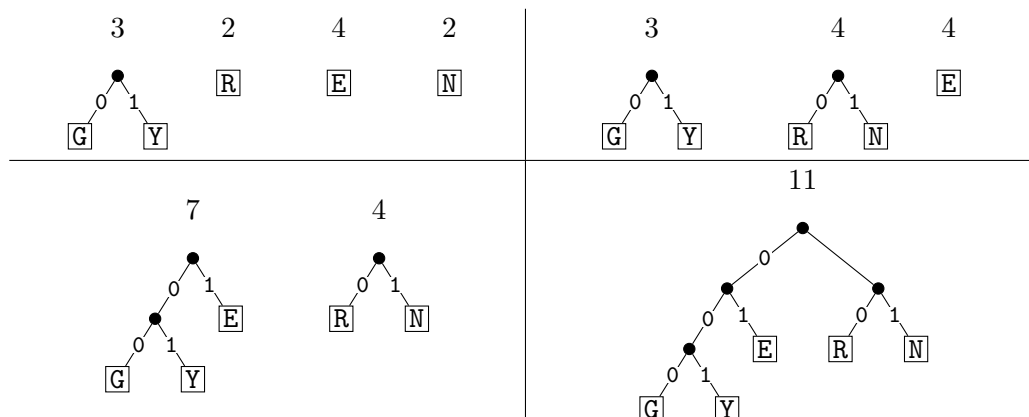


Figure 10.8: Combining tries to build the Huffman trie.

6513 to z (the root of the new trie), i.e., how often will z get accessed during encoding? We will go
 6514 through z whenever we need to access a character that is in either T or T' . In other words, node
 6515 z should have frequency $f(T) + f(T')$.

6516 Every time we do this combination step, we decrease the number of tries; repeat until only
 6517 one trie is left. The result is called the *Huffman trie*. A few details should be noted:

- 6518 • There often will be ties in determining which sub-tries to use. In the example, we must
 6519 use character Y in the first step because it has the minimum frequency. But the other
 6520 character could be any of G , R , N , because all those have frequency 2. It turns out that
 6521 for the quality of the compression it does not matter how ties are broken.

6522 Likewise, there is no rule as to which of the two tries goes into the subtree where the link
 6523 is labeled '0'. Again, either way will give the same compression ratio.

6524 In particular, the Huffman trie is not unique.

- 6525 • In the example, we assumed that the input-alphabet Σ_S is $\{G, R, E, N, Y\}$, i.e., exactly
 6526 the characters that appear in the input-string S . But usually the alphabet Σ_S is much
 6527 more general, e.g. all of ASCII. How should we handle characters of Σ_S that do not appear
 6528 in S ?

6529 One option is to ignore these characters (this is what was done above). This keeps the
 6530 encoding trie slightly smaller (hence results in a slightly shorter compressed string). The
 6531 other option is to include all characters of Σ_S , assigning frequency 0 to all of them. (This
 6532 would lump them all together into one big sub-trie, since all those characters would be
 6533 processed first.) The latter option is preferred in situations where we might want to reuse
 6534 the encoding trie for other texts.

6535 In our examples here we will *not* include the frequency-zero characters, though assignments
 6536 may ask you to include them.

- 6537 • Consider the source-text **AAAAA**. Here we have only one character in S . (This does not

violate our assumption that $|\Sigma_S| \geq 2$, because we do not necessarily have to use all characters that were permitted for S .) If we do not include frequency-0 characters, what would the Huffman trie be here? If we followed the above instructions exactly, then we would create one trie that stores **A** and then finish immediately. But this is not a valid encoding trie: the codeword for **A** would be the empty word, and this is not lossless.

So in the case where S has only one character we cannot follow the above strategy and must use some alternative. Rather than dealing with this in all the statements below, we instead “define the problem away” and make the general assumption that for Huffman-encoding, there are always at least two distinct characters in the input.

- Huffman’s algorithm was motivated by the goal to minimize (for a given source text S) the length of the coded text. This cost is indeed minimized, as long as we only consider the kinds of encodings seen here, i.e., those that use a prefix-free character-by-character encoding and for which the coded text uses alphabet $\Sigma_C = \{0, 1\}$.⁵ A proof will be given (for the enriched section) below.

Algorithm to find the Huffman-trie: To find the Huffman-tree, we need a data structure that permits us to find quickly the trie that has the smallest frequency. We have seen this in Chapter 2: we should use priority queues! However, we want to find the *minimum* element, so we need a *min-oriented* priority queue. Algorithm 10.4 gives the code to find the Huffman-code, given the frequencies.

Algorithm 10.4: *Huffman::buildTrie*(Σ_S, f)

```

Input : Source alphabet  $\Sigma_S$  with  $|\Sigma_S| \geq 2$ , frequencies  $f$  of these characters
// pre: At least two characters have non-zero frequency
1  $Q \leftarrow$  min-oriented priority queue           // PQ to store tries
2 forall  $c \in \Sigma_S$  with  $f[c] > 0$  do
3    $Q.insert(\text{single-node trie for } c \text{ with weight } f[c])$ 
4 while  $Q.size > 1$  do                           // build encoding trie
5    $T_1 \leftarrow Q.deleteMin()$ 
6    $f_1 \leftarrow$  weight of  $T_1$ 
7    $T_2 \leftarrow Q.deleteMin()$ 
8    $f_2 \leftarrow$  weight of  $T_2$ 
9    $T \leftarrow$  trie with  $T_1, T_2$  as sub-tries and weight  $f_1 + f_2$ 
10   $Q.insert(T)$ 
11 return  $Q.deleteMin()$ 
```

The run-time of this algorithm is $O(|\Sigma_S| \log |\Sigma_S|)$. This holds because the priority queue Q

⁵There *are* methods to beat Huffman, not only in practice (as we will see in later sections) but also in theory. These use different code alphabets and/or are not character-by-character encodings.

6558 is initialized with $|\Sigma_S|$ items and never gets bigger (we always remove two items before we insert
 6559 a new one). Thus any call to *deleteMin* takes $O(\log |\Sigma_S|)$ time, and there are $|\Sigma_S| - 1$ of them
 6560 since the size of Q decreases every time.

6561 **Full Huffman-encoding:** We use the term *Huffman encoding* for prefix-free encoding that
 6562 uses the Huffman-trie. This encoding hence has three parts: find the frequencies, compute the
 6563 Huffman-trie, and do the actual encoding. Algorithm 10.5 gives the full algorithm for doing
 6564 Huffman encoding. The run-time for this is hence $O(|S| + |\Sigma_S| \log |\Sigma_S| + |C|)$. This can be
 6565 simplified to $O(|\Sigma_S| \log |\Sigma_S| + |C|)$ since $|S| \leq |C|$ (every character of S gives rise to a codeword
 6566 with at least one character).

Algorithm 10.5: *Huffman::encoding(S, C)*

Input : Input-stream S where S contains ≥ 2 distinct characters, output-stream C

```

1  $f \leftarrow$  array indexed by  $\Sigma_S$ , initially all-0           // compute frequencies
2 while  $S$  is non-empty do increase  $f[S.pop()]$  by 1
3  $T \leftarrow \text{Huffman::buildTrie}(\Sigma_S, f)$                 // find encoding trie
4 Store  $T$  in the class
5  $C.append(\text{trie } T \text{ encoded suitably})$ 
6 reset input-stream  $S$ 
7 prefixFree::encoding( $S, C$ )                             // do actual encoding
```

6567 **Discussion:** The Huffman-trie and the corresponding character-by-character encoding is very
 6568 appealing in theory, because we provably get the trie with the smallest possible cost (in this
 6569 model) and can therefore hope for good compression ratios. However, in practice there are
 6570 various issues around Huffman-encoding, and it is not the most popular (at least not when
 6571 applied directly). Here are various discussion items:

- 6572 • As we mentioned earlier, the Huffman encoding-trie is not unique, not even for a given
 6573 set of frequencies. As such, it is not possible for the decoder to do the decoding unless
 6574 we send the trie along with the coded text. (We did not say how exactly to convert the
 6575 trie into a text that can be sent, but for example one could send the nodes twice, once in
 6576 pre-order and once in level-order.) Thus while the actual coded text should be short, the
 6577 total amount of information that must sent may be much longer. This may make coded
 6578 text obtained with Huffman's algorithm *longer* than sending plain ASCII, especially in
 6579 situations where all characters have roughly the same frequency.
- 6580 • Note that Huffman-encoding needs to go through the source-text twice, once to compute
 6581 the frequencies and once to do the actual encoding. Therefore Huffman encoding cannot
 6582 use streams unless we have a way to reset the input-stream.
- 6583 • Various attempts have been made to overcome the above limitations. For example, we

could define tie-breaking rules, such as ‘among characters with the same frequency, take the one that comes earlier in the alphabet’ or ‘put the character that comes earlier in the sub-trie with 0’. If done right, then it suffices to send along only the frequencies (rather than the whole trie T) with the coded text, making it shorter. The decoding algorithm can then re-construct the Huffman-trie, based on the frequencies and the tie-breaking rules alone.

- Another variation is that we do not use the frequencies as determined by the specific source text S , but instead use some pre-determined frequencies. For example, we know the frequencies of letters in typical English text (see Table 10.4). If the encoder and the decoder agree on tie-breaking rules and on a fixed table of frequencies, then only the encoded text C has to be sent along. On the flip-side, if we use general frequencies rather than the specific frequencies within S , then optimality no longer holds, so we are potentially sending a coded text C that is longer than necessary.
- There are also hybrid methods that start with a general set of frequencies, but adapt the frequencies occasionally by keeping track of the frequencies of characters that appeared in the text encoded thus far. This can be imitated by the decoder, giving a method to send only the encoded text while still being (somewhat) sensitive to the frequencies within the source text and therefore coming closer to the shortest-possible encoding.

Summarizing, there are numerous ways of implementing the details of Huffman-coding, offering a tradeoff between the length of the actual coded text and the length of what actually needs to be transmitted to the decoder. None of these methods turn out to be the best in practice.

Proof of Optimality: (cs240e) We now show that Huffman’s algorithm gives the binary prefix-free code that has the minimum possible cost. Recall that the cost is

$$\text{cost}(T) = |C| = \sum_{c \in \Sigma_S} f(c) \cdot d_T(c) = \sum_{c \in \Sigma_S} (\text{frequency of } c) \cdot (\text{depth of } c \text{ in encoding trie } T).$$

Lemma 10.2. *Fix a source-text S (and the corresponding alphabet Σ_S and frequencies $f(\cdot)$). Let T_0 be an arbitrary encoding trie for Σ_S , and let the T_H be the Huffman-trie. Then $\text{cost}(T_H) \leq \text{cost}(T_0)$.*

Proof. We prove this by induction on $|\Sigma_S|$, with an inner induction on the size of T_0 . Recall that we made the assumption that there are at least two characters in S , so $|\Sigma_S| \geq 2$ and the base case is $|\Sigma_S| = 2$. In the base case, we have two characters $c, c' \in \Sigma_S$, and the Huffman-trie assigns single-bit code-words to both. Hence $\text{cost}(T_H) = f(c_1) + f(c_2)$. Any other trie T_0 must have at least this cost since the codeword of any character must use at least one bit.

Now assume $|\Sigma_S| > 2$ and let a, a' be the first two characters that were extracted from the priority queue Q when building the Huffman-trie. In particular, these two characters have the smallest possible frequencies. Also, a and a' become siblings in the Huffman-trie; let their parent be p_H . The goal is now to modify the other trie T_0 until we are in a situation where we can apply induction; see Figure 10.9 for an illustration.

- Assume first that some leaf b has no sibling. (We will in the following use characters and leaves that store them interchangeably.) Then the parent p_0 of b is an interior node that has only one child. This is wasteful. Create a new encoding-trie T_1 where b has been moved up to take the place of p_0 . Thus the depth of b has decreased, all other depths are unchanged, and so $\text{cost}(T_1) \leq \text{cost}(T_0)$.
- Repeat the above transformation until in the resulting trie T_1 every leaf has a sibling. Let b be a lowest leaf (breaking ties arbitrarily). Let b' be its sibling (which is also a leaf, else it would be lower).
- Up to renaming we may assume $f(b) \leq f(b')$. Note that $f(a) \leq f(b)$ and $f(a') \leq f(b')$ by choice of a, a' . We also know that $d_{T_1}(b') = d_{T_1}(b)$ is no smaller than both $d_{T_1}(a)$ and $d_{T_1}(a')$ by choice of b, b' . Let T_2 be the encoding trie obtained by exchanging a with b and exchanging a' with b' . Since only a, a', b, b' have a different depth in T_2 then in T_1 , we have

$$\begin{aligned}
 \text{cost}(T_2) - \text{cost}(T_1) &= \underbrace{(d_{T_2}(a) - d_{T_1}(a))}_{d_{T_1}(b)} f(a) + \underbrace{(d_{T_2}(a') - d_{T_1}(a'))}_{d_{T_1}(b')} f(a') \\
 &\quad + \underbrace{(d_{T_2}(b) - d_{T_1}(b))}_{d_{T_1}(a)} f(b) + \underbrace{(d_{T_2}(b') - d_{T_1}(b'))}_{d_{T_1}(a')} f(b') \\
 &= \underbrace{(d_{T_1}(b) - d_{T_1}(a))}_{\geq 0} \underbrace{(f(a) - f(b))}_{\leq 0} + \underbrace{(d_{T_1}(b') - d_{T_1}(a'))}_{\geq 0} \underbrace{(f(a') - f(b'))}_{\leq 0} \leq 0
 \end{aligned}$$

and therefore $\text{cost}(T_2) \leq \text{cost}(T_1)$.

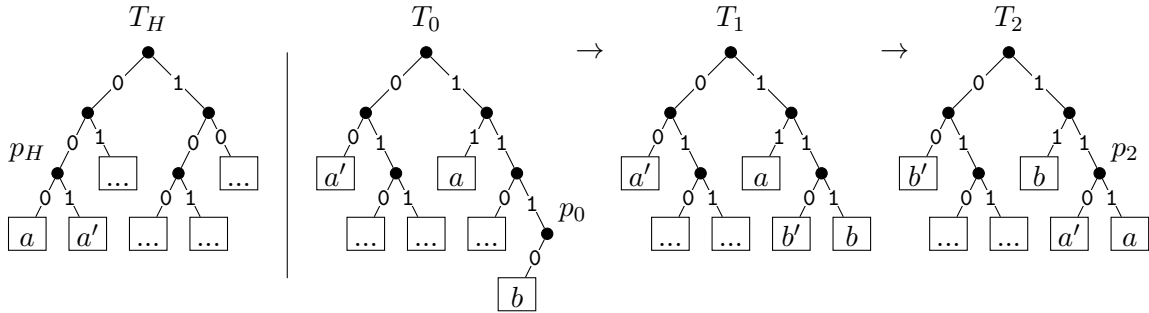


Figure 10.9: Modifying any other encoding trie T_0 until we can apply induction.

So now the two characters a, a' are siblings in both T_H and T_2 ; let their parents be p_H and p_2 , respectively. We are now finally in a position to apply induction.

Namely, let q be some character that does *not* appear in Σ_S . Let S' be the source text that we'd get if we replaced any occurrence of either a or a' by q . Thus S' has alphabet $\Sigma' := \Sigma_S \setminus \{a, a'\} \cup \{q\}$, and characters have the same frequencies except that $f(q) := f(a) + f(a')$. Observe that $T'_2 := T_2 \setminus \{a, a'\}$ and $T'_H := T_H \setminus \{a, a'\}$ are encoding tries for Σ' if we assign q to leaves p and p_H , respectively. See Figure 10.10. Also note that T'_H is exactly the Huffman-tree

that we would have built for Σ' by choice of a, a' , since they were the first two characters whose tries were removed from priority queue Q .



Figure 10.10: Creating encoding tries for the smaller alphabet $\Sigma \setminus \{a, a'\} \cup \{q\}$.

By induction, we hence have $\text{cost}(T'_2) \geq \text{cost}(T'_H)$. Observe that all characters other than a, a', q have the same depth in T_2 and T'_2 . Therefore

$$\begin{aligned}
 \text{cost}(T_2) - \text{cost}(T'_2) &= \sum_{c \in \Sigma_S} f(c) \cdot d_{T_2}(c) - \sum_{c \in \Sigma'} f(c) \cdot d_{T'_2}(c) \\
 &= \left(f(a) \cdot \underbrace{d_{T_2}(a)}_{d_{T'_2}(q)+1} + f(a') \cdot \underbrace{d_{T_2}(a')}_{d_{T'_2}(q)+1} \right) - \underbrace{f(q)}_{f(a)+f(a')} \cdot d_{T'_2}(q') \\
 &= f(a) + f(a')
 \end{aligned}$$

Likewise one shows that $\text{cost}(T_H) - \text{cost}(T'_H) = f(a) + f(a')$. Therefore

$$\text{cost}(T_H) = \text{cost}(T'_H) + f(a) + f(a') \leq \text{cost}(T'_2) + f(a) + f(a') = \text{cost}(T_2) \leq \text{cost}(T_0)$$

as desired. □

10.2 Multi-character encodings

So far, we have studied *single-character encodings*, which consist of an assignment of code-words to characters. However, it makes sense to encode longer strings with one code-word if there are some strings that repeat frequently. For example, in English text we will frequently have **th**, **an**, or longer substrings such as **the**, **then** and **and**. Likewise, in HTML we will frequently have **<a href**", **<img src**", **
", and in C++ we will frequently have **for, **while**, etc. It makes sense to assign a single code-word to such strings, hence allowing multiple characters per code-word.

This raises, however, two problems:

- How many code-words should we use? The more code-words we use, the longer the individual code-words must be. But on the other hand, more code-words mean that more strings can be assigned a code-word.

- How should we determine *which* strings should obtain a code-word? In particular, this is highly dependent on the language of the encoded text. Ideally, the encoding scheme would determine this by itself, without user help.

We will see two examples multiple-character encodings here. The first one (run-length-encoding) is very limited, and really only suitable for special situations. The second one (Lempel-Ziv-Welch encoding) is very versatile and performs very well on English text.

10.2.1 Run-Length Encoding

We assume throughout this section that the source text is a bit-string, i.e., $\Sigma_S = \{0, 1\}$. The main idea of *run-length encoding* is that we can describe any bit-string by listing its first character, and listing the lengths of its runs (where a *run* is a maximal set of consecutive characters that are the same). Consider for example the source text

$$S = \underbrace{00000}_{5} \underbrace{111}_{3} \underbrace{0000}_{4}.$$

We can describe S as “0,5,3,4”, which means the following:

- The first bit of S is 0.
- The first run of S has length 5, i.e., S begins with 00000.
- The next run of S has length 3. Since runs were defined to be *maximal* subsequences of identical characters, and since the alphabet is $\{0, 1\}$, we know that this run necessarily consists of 1s. Therefore, the next 3 characters of S are 111.
- The next run of S has length 4. By a symmetric argument therefore the next 4 characters of S are 0000.

Thus one can easily deduce from the coded ‘text’ 0, 5, 3, 4 that the source text is 000001110000, and this is a lossless compression.

Encoding sequences of integers: There is only one hitch: Currently, we describe the output as a list of integers, but our goal was to have a bit-string as coded text. This therefore raises the question: how to encode an integer as a bit-string? Any one integer is naturally encoded as bit-string by using its base-2 representation, but if we want a sequence of integers, then we either must allow a third character (a comma or some similar separator), or find a way to express a comma through a special bit-sequence.

We describe here the *Elias-gamma-code* that can be used for a lossless encoding of a sequence of positive integers. First, encode one integer $k > 0$ as follows:

- Write $\lfloor \log k \rfloor$ copies of 0.
- Then write the binary representation of k (using the minimal number of bits, which in particular implies that it always starts with 1).

See Table 10.11 for the first few encodings. If we use $E(k)$ for the encoding of integer k , then we can encode a sequence of positive integers by concatenating their encodings, i.e., k_1, k_2, k_3, k_4

Algorithm 10.6: *RLE::encoding(S, C)*

```

Input : Non-empty input-stream  $S$  of bits, output-stream  $C$ 
1  $b \leftarrow S.top()$  // bit-value for the current run
2  $C.append(b)$ 
3 while  $S$  is not empty do
4    $k \leftarrow 1$  // length of run
5   while  $S$  is not empty and  $S.pop() = b$  do  $k++$ 
6    $K \leftarrow$  empty string // binary encoding of  $k$ 
7   while  $k > 1$  do
8      $C.append(0)$ 
9      $K.prepend(k \bmod 2)$ 
10     $k \leftarrow \lfloor k/2 \rfloor$ 
11   $K.prepend(1)$ 
12   $C.append(K)$ 
13   $b \leftarrow 1 - b$  // flip bit for next run

```

Run-length decoding: Recall that (for this section) the binary encoding of a positive integer k *always* starts with a 1; we omit all leading 0s. This is exactly what makes decoding feasible: as long as we see 0s, we know that this is the “counting” part of the Elias-gamma-code that tells us how long the binary string is going to be. So to decode a run-length encoding, we proceed as follows:

- Read the initial bit.
- For as long as there are 0s, keep increasing a counter ℓ .
- Get the next $\ell + 1$ bits to get the binary encoding of the run-length.
- Create the appropriate run (we know whether it consists of 0s or 1s from the initial bit, which we flip with every round).
- Append the run to the output and repeat.

Algorithm 10.7 gives the pseudocode for this procedure. The run-time of this algorithm is $O(|S| + |C|)$, because with every step we either advance one bit in C , or we append one or more bits to S . Note that this algorithm fails (in fact, crashes) if the input was not appropriate, i.e., not a run-length encoding of a string. We could buffer for this by checking in line 4 and line 7 whether C has become empty, but to keep the code simpler we leave this to the reader.

Let us see an example of decoding. When given the bit-string $C = 00001101001001010$, we

- first extract the initial bit (‘0’),
- now read three more 0s, which tells us that the binary encoding of the next run has 4 bits,
- now we extract the next 4 bits so that we get the first Elias-gamma-code:

$$C = 0\underbrace{0001}_{E(k)}101001001010$$

Algorithm 10.7: *RLE::decoding*(C, S)

Input : Non-empty input-stream C of bits, output-stream S
// pre: C is a valid run-length encoding

```

1  $b \leftarrow C.pop()$  // bit-value for the current run
2 while  $C$  is not empty do
3    $\ell \leftarrow 0$  // length of binary encoding, minus 1
4   while  $C.pop() = 0$  do  $\ell++$ 
   // The last pop() removed the leading bit of the binary encoding
5    $k \leftarrow 1$ 
6   for  $j \leftarrow 1$  to  $\ell$  do // get binary encoding and convert
7      $k \leftarrow k * 2 + C.pop()$ 
8   for  $j \leftarrow 1$  to  $k$  do  $S.append(b)$  // append the run
9    $b \leftarrow 1 - b$  // flip bit for next run

```

6719 This decodes to 13, so we know that the output S starts with 13 0s.

- Repeating this, we get three more encodings of runs:

$$C = \underbrace{00001101}_{E(13)} \underbrace{00100}_{E(4)} \underbrace{1}_{E(1)} \underbrace{010}_{E(2)}$$

- So the run-lengths are 13,4,1,2, and since we start with 0 and alternate the bits, we have

$$S = \underbrace{00000000000000}_{13} \underbrace{1111}_{4} \underbrace{0}_{1} \underbrace{11}_{2}$$

6720 In summary, run-length encoding is a very simple and very fast method of converting a bit-
6721 string into a bit-string. The problem with it is that its compression ratio depends very much on
6722 the input and the run-lengths that it has. A run of k bits is compressed to $2 \log k + 1 \in o(k)$ bits.
6723 This is great if k is big, but if k is small then this is not good. Indeed, we have no compression
6724 unless $k \geq 7$, and for $k = 2$ or 4 the Elias-gamma-code uses *more* than k bits. In consequence,
6725 if there are lots of runs of length up to 6 and few longer runs, then the run-length-encoding is
6726 *longer* than the original!

6727 One place where run-length encoding has proved useful is in transmitting pictures, especially
6728 black-and-white pictures. Imagine a picture represented by using 0 for white and 1 for black.
6729 Most pictures have large patches of all-white or all-black, which corresponds to long runs, so for
6730 encoding pictures the compression ratio achieved with run-length encoding should be good. This
6731 is especially true if the picture is of a piece of paper with a few words on them, where nearly
6732 everything is white. In particular, run-length encoding was very popular with fax-machines
6733 (back when we still used such things). These days it is used in some image formats (TIFF), and
6734 as part of bzip2 as we will see soon.

One drawback of run-length encoding is that it crucially *requires* the input to be a bit-string. One can of course convert any input into a bit-string, for example by converting every ASCII-character into its 7-bit representation. However, this would destroy all the runs. Therefore, to use run-length encoding for arbitrary alphabets, we must send not only the length of the run but also which character it encodes. For example, the word *BBBAACC* could be described by *B, 3, A, 2, C, 2* (which then in turn we must encode somehow, e.g. by replacing characters by ASCII-codes and numbers by Elias-gamma-codes). But the overhead for sending the characters makes the coded text long, and run-length encoding is rarely used except for bit-strings.

10.2.2 Lempel-Ziv-Welch

We now turn to the Lempel-Ziv-Welch (or LZW) encoding, which also represents longer strings of characters with one code-word, but differs from run-length encoding in multiple ways:

- It does *not* need to know what substring should get encoded by one code-word. (This is in contrast to run-length encoding, where we fixed that only long runs of repeating characters get encoded by one code-word.)
- It uses an *adaptive* dictionary, which means that the dictionary changes during the encoding. (This is in contrast to all previous encoding schemes, where the dictionary was *static*: The dictionary was fully determined before encoding starts and never changed afterwards.) Allowing an adaptive dictionary may sound a little scary—wouldn't we have to send the dictionary to the decoder then so that they know how to decode? The trick for using adaptive dictionaries is to create a rigid set of rules of *how* the dictionary is changed. Furthermore, the rules must be chosen in such a way that the decoder can *deduce* how the dictionary is changed. Put differently, we do not send the dictionary along, but there is a fixed way of how it was created, and the decoder can re-create the exact same dictionary during the decoding process. Needless to say, this makes decoding less obvious than it has been in the past!

LZW encoding: The input to LZW is an ASCII string S , i.e., $\Sigma_S = \text{ASCII}$. For now, we will assume that the output-alphabet is the infinite set of non-negative integers, i.e., we want to convert S into a sequence of numbers. We will discuss later how to convert this sequence of numbers into a bit-string.

So assume we have a stream S of ASCII characters. Since we have an adaptive encoding scheme, we also maintain a dictionary D that maps strings to their code-words (i.e., non-negative integers). Initially, D is simply ASCII itself, i.e., the mapping from ASCII-character to integers. The LZW-encoding now consists of only two steps, repeated over and over until the entire input S is processed:

- “Encode as much as possible”: Parse the next few characters of S for as long as the corresponding substring w is *already* in D . Output the code-number of w .
- “Add something useful to D ”: Let K be the next character after w in S . (If there is none then we are done with encoding.) It would have been useful to have $w \mathbin{++} K$ in dictionary

6773 D , because then we could have mapped an even longer substring to one code-number. But
 6774 $w \mathrel{++} K$ was not in D (else we would have made w longer). So add $w \mathrel{++} K$ to D , using the
 6775 next code-number.

6776 Figure 10.12 shows an example of the encoding. This can be read as follows: The vertical line
 6777 segments denote the end of the successfully read string w . The larger number in the interval
 6778 between two such segments is the code-number that corresponds to w . The smaller number
 6779 above the segment is the code-number that has been assigned in this round. The string that is
 6780 assigned to it consists of the entire string on the left, and the first character of the string on the
 6781 right. (Sometimes we will indicate this string with a dotted box.) The sequence of code-numbers
 6782 that encodes this string hence is 65,78,128,65,83,128,129.

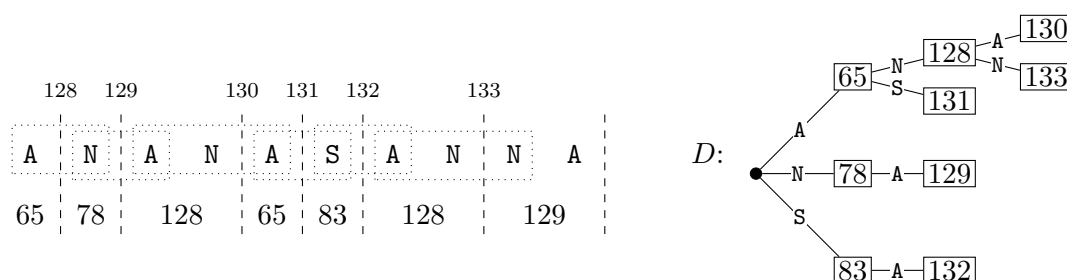


Figure 10.12: Example of an LZW encoding. We omit unused parts of dictionary D (i.e., all ASCII characters except A, N, S).

6783 Based on what we need to do with D , we see that the best way to store it is in a trie. *Every*
 6784 node of the trie (except the root) will store a code-number, and the code-number at node v
 6785 corresponds to the string of characters on the edges to v . Finding the string w is then very easy:
 6786 we simply parse the characters of S , starting from the root, until we reach “no such child” at
 6787 some node v for some character K ; the code-number to use is then simply the one stored at v .
 6788 Adding the new entry in D is also very easy—simply add the child at v that we would have liked
 6789 to have (i.e., with character K) and give it the next available code-number, which we maintain
 6790 with a global counter. Algorithm 10.8 gives the details. Note that we spend $O(1)$ time per
 6791 character that was removed from S (assuming the trie stores children so that the appropriate
 6792 child can be found in constant time). So the run-time of this algorithm is $O(|S|)$.

6793 Figure 10.13 gives another example with source text **barbarabarbarbaren**.⁷ We omit the
 6794 trie here; for small examples like these it suffices to write the encoded string with the code-
 6795 numbers and scan them all to find the longest that fits the next part of the source text. Notice
 6796 how quickly LZW realized that the string **bar** is very important here and hence assigns it a
 6797 code-number.

⁷That’s part of a German tongue twister: rhabarberbarbarabarbarbarenbartbarbierbier. Generally the texts that we use for compression get sillier and sillier, because they have to be chosen with lots of repetitions for the compression to do its magic.

Algorithm 10.8: *LZW::encoding(S, C)*

Input : Input-stream S of ASCII-characters, output-stream C

```

1 Initialize dictionary  $D$  as a trie that maps ASCII to  $\{0, \dots, 127\}$ 
2  $idx \leftarrow 128$  // global counter for first free code-number
3 while  $S$  is not empty do
4    $v \leftarrow$  root of trie  $D$ 
5   while  $S$  is non-empty and  $v$  has a child  $c$  labeled with  $S.top()$  do
6      $v \leftarrow c$ 
7      $S.pop()$ 
8    $C.append(\text{code-number stored at } v)$ 
9   if  $S$  is non-empty then
10    create child of  $v$  labeled  $S.top()$  with code-number  $idx$ 
11     $idx++$ 

```

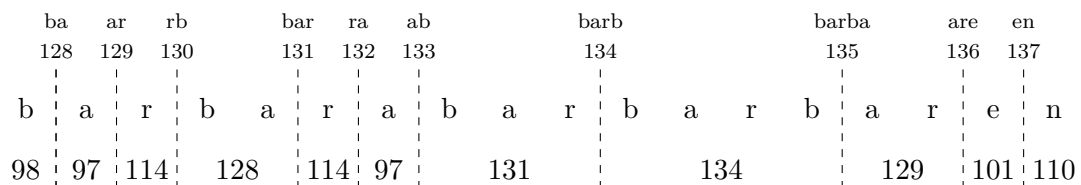


Figure 10.13: Second example for LZW encoding.

6798 To see how well LZW can compress strings in the best case, let us also consider the string a^n ,
6799 i.e., with n copies of the same character (in this case a). In the first round, we write code-number
6800 97 for ‘a’ and assign 128 to ‘aa’. In the next round, we immediately use 128, and assign 129
6801 to ‘aaa’. In the next round, we immediately use 129 and assign 130 to ‘aaaa’. Generally in the
6802 i th round (where initially $i = 1$) we write the code-number for a^i . This means that we encoded
6803 in total the string a^I where $I = \sum_{j=1}^i j = \frac{i(i+1)}{2}$. This means that after roughly $\sqrt{2n}$ rounds,
6804 we have encoded the entire string. So in this case LZW uses $O(\sqrt{n})$ code-numbers. One can
6805 also argue that it cannot use fewer code-numbers, because the code-number used in round i can
6806 encode at most i characters. So the best case for LZW is worse than the best-case for run-length
6807 encoding. However, in practice LZW far beats it since it detects suitable repeats, rather than
6808 relying on runs to exist in the text.

LZW decoding: Now let us turn to the problem of decoding a given sequence of integers that was obtained using LZW encoding. This is a lot harder than for previous encoding schemes, because LZW uses an adaptive dictionary and so we (the decoders) must also build this dictionary

while decoding. Let us use as running example the following sequence of code-numbers:

$$C = 67, 65, 78, 32, 66, 129, 133, 83$$

A few of the code-numbers are ASCII (i.e., in the range 0-127), and for those we know what they stand for: the unknown source text S begins with CAN_B . But to know what code-number 129 stands for, we really need to build the dictionary D . But we know how this was done: in each step, take the string that was encoded, and append the first character of the next string; the combination gets assigned to the next code-number. So we have

$$\text{C} \mathbin{++} \text{A} \rightarrow 128 \quad \text{A} \mathbin{++} \text{N} \rightarrow 129 \quad \text{N} \mathbin{++} _ \rightarrow 130 \quad _ \mathbin{++} \text{B} \rightarrow 131 \quad \text{B} \mathbin{++} \text{A} \rightarrow 132$$

6809 which tells us that 129 decodes to AN. To be more systematic, let us put the decoding-steps into
6810 a table, listing in each row the current code-number, decoded string, and what was added to D .

input	decodes to	Code #	String
67	C		
65	A	128	CA
78	N	129	AN
32	_	130	N_
66	B	131	_B
129	AN	132	BA
133	???	133	

6812 In the next step, however, we have a problem. We encounter code-number 133, but 133 is
6813 not yet in the dictionary! This happens because the decoder is “one step behind” in building
6814 the dictionary—it needs to know the first character of the next string to add to the dictionary,
6815 so can do this only once the next string is known, i.e., one step later.

6816 One could fix this by changing how we encode, by not permitting the use of a code-number
6817 until at least one more step has gone by. But it turns out that this is not needed—by studying
6818 what happened during the encoding process, we can actually deduce what the string encoded
6819 by 133 is, even though it is not in the dictionary yet. Let us look at what happened in the
6820 encoding S using x_1, x_2, \dots for characters of S that we do not know yet.

	CA	AN	N_	_B	BA	AN x_1						
	128	129	130	131	132	133						
	C	A	N	_	B	A	N	x_1	x_2	x_3	x_4	...
	67	65	78	33	66	129						
									133			

6822 So we know that 133 encodes a length-3 string ANx_1 where x_1 is unknown. However, because
 6823 we are in the special situation where 133 is used *immediately* after it has been added to the
 6824 dictionary, we actually do know what x_1 is. Look at the picture of encoding again. Because 133
 6825 was used in the next step, its corresponding string is $x_1x_2x_3$. But we also know that 133 encodes
 6826 ANx_1 . This means that x_1 must be A. More generally, whenever we encounter a code-number
 6827 that is not yet in D but about to be added, then it encodes the previous string, plus the first
 6828 character of the previous string repeated. If we encounter a code-number that is even bigger,
 6829 then the encoding was not valid. With this, we are ready for the pseudocode for LZW decoding,
 6830 which is in Algorithm 10.9. Also, Figures 10.14 and 10.15 show the above example continued,
 6831 and also the decoding of the example from Figure 10.13.

Algorithm 10.9: $LZW::decoding(C, S)$

Input : Input-stream C of integers, output-stream S

```

1 Initialize  $D$  as a dictionary that maps  $\{0, \dots, 127\}$  to ASCII
2  $idx \leftarrow 128$  // global counter for first free code-number
3  $code \leftarrow C.pop()$  // first number creates no entry in  $D$ 
4  $s \leftarrow LZW::dictionaryLookup(D, code)$ 
5  $S.append(s)$ 
6 while  $C$  is not empty do
7    $s_{prev} \leftarrow s$ 
8    $code \leftarrow C.pop()$ 
9   if  $code < idx$  then  $s \leftarrow LZW::dictionaryLookup(D, code)$ 
10  else if  $code = idx$  then  $s \leftarrow s_{prev} \mathbin{++} s_{prev}[0]$  // special situation
11  else return “invalid encoding”
12   $S.append(s)$ 
13  insert  $s_{prev} + s[0]$  into  $D$  with code-word  $idx$ 
14   $idx++$ 
```

6832 To analyze the run-time for Algorithm 10.9, observe that the while-loop executes $|C|$ times.
 6833 Within one execution, we spend $O(|s|)$ time to append s to S , making the total run-time
 6834 $O(|C| + |S|)$, not counting the time that it takes to look up $code$ in dictionary D . This run-
 6835 time bound can be simplified to $O(|S|)$ since $|C| \leq |S|$ (each codeword represents at least one
 6836 character of S).

6837 The time to look up $code$ in D depends on how D is stored. The straightforward method
 6838 of storing D is to use an array that stores strings indexed by the code-number, as suggested in
 6839 Figure 10.14. Then the lookup takes $O(1)$ time and returns a word.

6840 However, just as for prefix-free codes, observe that the words stored in the dictionary might
 6841 get quite long; even on our small example in Figure 10.15 we needed to store **barba**, a word of
 6842 length 5. Moreover, it is unnecessary to store the entire word **barba**: We *know* that the word

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	ANA	133	ANA	129, A
83	S	134	ANAS	133, S

Figure 10.14: LZW decoding example completed.

barb was also in D , because the only way to add a new word to D is to take an existing word and append a character. So a much more space-efficient way to store a word in D is to store where its prefix was, and what character c was appended. This is really the same idea as we used for prefix-free codes, where we stored a reference to the parent in the trie and the character that was on the link. But we do not even need to go through the trie, we can do it directly in the array: The prefix had a code-number, so refer to it via its code-number. With this, we use $O(1)$ space per entry in D (hence $O(|C|)$ space total) yet still can look up the word in D recursively. See the rightmost column of Figures 10.14 and 10.15 for how dictionary D is stored, and Algorithm 10.10 for how to look up a word in D .

Algorithm 10.10: *LZW::dictionaryLookup(D , $code$)*

Input : Dictionary D as array of (int,char)-pairs, integer $code$

```

1  $s \leftarrow$  empty word
2 while  $code \geq 128$  do
3    $(code, c) \leftarrow D[code]$ 
4    $s.prepend(c)$ 
5  $s.prepend(D[code])$                                 //  $code < 128 \Rightarrow$  ASCII character
6 return  $s$ 
```

This lookup takes more than constant time, but the time is proportional to $|s|$, where s is the returned code-word. Since we need this time to append s to the output anyway, this is only a constant overhead; the run-time for LZW decoding remains $O(|S|)$.

LZW discussion and variants: In our description of LZW thus far, we have used positive integers as encoding. This is problematic, partially because we wanted bit-strings for encodings, but more crucially because we are allowing an infinite alphabet Σ_C for the encoding (which is

input	decodes to	Code #	String (human)	String (computer)
98	b			
97	a	128	ba	98, a
114	r	129	ar	97, r
128	ba	130	rb	114, b
114	r	131	bar	128, r
97	a	132	ra	114, a
131	bar	133	ab	97, b
134	barb	134	barb	131, b
129	ar	135	barba	134, a
101	e	136	are	129, e
110	n	137	en	101, n

Figure 10.15: A second LZW decoding example. The special situation occurred when decoding 134.

unfair since this has much more expressive power).

There are numerous ways of how to convert the LZW encoding into a bit-string. The simplest (and original) approach was to use a fixed-length encoding for the integers, for example encode every integer as a length-12 bit-string. To give just one example, the output from Figure 10.12 would become

$$\underbrace{000001000001}_{65} \underbrace{000001001110}_{78} \underbrace{000010000000}_{128} \underbrace{000001000001}_{65} \underbrace{000001010011}_{83} \underbrace{000010000000}_{128} \underbrace{000010000001}_{129}$$

The main disadvantage here is that we then have “only” 4096 code-numbers at our disposal (or actually only 3968 since the first 128 are reserved for ASCII). The original approach was to simply stop adding further strings to the dictionary, making the assumption that 4096 code-numbers are enough to capture most commonly used patterns, and adding more code-numbers (which would come at the price of adding more bits per code-number) is not worth it.

Of course there are other options how to handle such a *dictionary-overflow*. We could agree that numbers always use as many bits as required, i.e., initially numbers are encoded using 8 bits, and the length of the bit-strings increases whenever *idx* (the index of the next free entry of the dictionary) reaches the next power of 2. Another option would be to encode the integers using Elias-gamma-codes. (This is not a good idea; Elias-gamma-codes are typically much too long!)

There are lots of other ways in which one could modify LZW. We currently start with dictionary *D* initialized as ASCII, but one could instead allow a bigger dictionary, such as ISO-8859. We allowed LZW to immediately re-use a code-number that it has assigned; some other methods disallow this for ease of decoding.

The most important part is that the encoder and decoder must *agree* on what exactly the rules are. The intent of LZW is that it does *not* send the dictionary along (in contrast to Huffman!), and instead it only sends the minimal information needed for the decoder to know what variant has been used to build D .

LZW encoding can be very bad in theory. If the input text has no repeated substrings, then an ASCII text (with 7 bits per character) gets effectively mapped to the same ASCII text (but now using 12 bits per character), so it may get longer! However, in practice there are lots of repeated substrings, and on English text a compression ratio of $\approx 50\%$ has been reported for the 12-bit-fixed-length encoding.

10.3 bzip2

One crucial insight to achieve even better compression ratios is to use a *text transform*: Before compressing the text, modify the text first (without changing its length) in the hope that the resulting text is somehow easier to compress. In particular, can we transform text so that there will likely be many long runs? (At first glance this may sound impossible, but as we will see, the Burrows-Wheeler transform (BWT) will do just that.) Can we transform text so that characters have very uneven frequencies and hence Huffman should perform well? (Again, this may seem impossible, but can be done.)

The original **bzip2** has multiple steps; we will not give the full details of all of them but list here only the ones that are the most interesting from a data structure point of view. Figure 10.16 gives an overview of all the steps, and each of the subsections below discusses each briefly (in reverse order).

10.3.1 Huffman encoding

The last step of **bzip2** is a Huffman encoding. Its input is a text that (as we will see) has lots of ‘special’ characters A' and B' , and the rest of the source consists of numbers (in the range $\{0, \dots, 127\}$) with smaller numbers more common than larger ones. As such, the distribution of characters is quite uneven and Huffman should perform quite well.

10.3.2 Modified run-length encoding

In the next-to-last step, **bzip2** receives a text T_2 that (typically) has long runs of zeroes. To compress this, it makes sense to perform run-length encoding with a few modifications:

- We *only* encode runs of zeroes, because it is not very likely that there are many other long runs, and this way we can use fewer bits to encode runs.
- Because we only encode runs of 0s, the alphabet of the output string will be quite big anyway. As such, we can afford to add two more characters A' and B' to the alphabet,

Transformation name	Properties	Alphabet
Text T_0 : barbarabarbarbaren\$		ASCII
↓ Burrows-Wheeler transform	If T_0 has repeated longer substrings, then T_1 has long runs of characters.	
Text T_1 : nrbbbbbr\$arrreaaaaa		ASCII
↓ Move-to-front transform	If T_1 has long runs of characters, then T_2 has long runs of zeros.	
Text T_2 : 110,114,100,0,0,0,0,1,6,100,2,0,0,103,2,0,0,0,0		$\{0, \dots, 127\}$
↓ Modified RLE	If T_2 has long runs of zeroes, then T_3 has chars A' and B' very frequently	
Text T_3 : 110,114,100, A',B' ,1,6,100,2, B' ,103,2, A',B'		$\{1, \dots, 127\} \cup \{A', B'\}$
↓ Huffman encoding	Compresses well since input-chars are unevenly distributed	
Text T_4 : 00011101111100010000000111110101110010110001		$\{0,1\}$

Figure 10.16: The main steps of `bzip2`.

6912 and to use these to encode the length of the run.⁸ With that, the boundary of the encoding
6913 of each run-length is obvious: it ends whenever a “normal” character is encountered. As
6914 such, there is no need to use Elias-gamma-codes, which saves almost half of the length of
6915 the encoding of the run-length.
6916 • We can use a slightly different encoding called *binary bijective numeration*, which works
6917 as follows:

6918

k	1	2	3	4	5	6	7	8	...
$E(k)$	A'	B'	$A'A'$	$A'B'$	$B'A'$	$B'B'$	$A'A'A'$	$A'A'B'$...

6919 (You have encountered this kind of enumeration if you have ever used a spreadsheet: The
6920 columns are enumerated the same way, except that they use 26 characters rather than
6921 2.) This uses $\lfloor \log(k+1) \rfloor$ characters, rather than $\lfloor \log k \rfloor + 1$ characters that are used by
6922 regular binary encoding. Therefore it is never worse, and usually better by one bit.

To see just a small example, if the input-alphabet consists of the number $\{0, \dots, 127\}$, then we transform

110, 114, 100, 0, 0, 0, 0, 1, 6, 100, 2, 0, 0, 103, 2, 0, 0, 0, 0 \rightarrow 110, 114, 100, A', B' , 1, 6, 100, 2, B' , 103, 2, A', B'

6923 because there are two runs of length 4 and one run of length 2.

⁸One of the characters A' and B' can actually be 0, since we will remove all 0s from T_2 , but we write A' and B' here for ease of understanding how the encoding of the run-length works.

If the input to this was a text T_2 that had lots of runs of zeroes, then the resulting text T_3 therefore should have lots of A' and B' , making it suitable for Huffman encoding as desired. Also note that for any characters other than 0, A' and B' , the frequency is the same in T_2 and T_3 , so an uneven distribution of frequencies in T_2 will be “passed on” to T_3 .

10.3.3 Move-to-front transform

Recall that we had the MTF-heuristic when storing a dictionary as an unsorted array, which makes searches faster if the search-requests are biased. We can use the same idea for converting a text into a different text that has skewed frequencies and (under some assumptions) many long runs of 0.

Specifically, assume that we are given a text T_1 with ASCII characters. Create a dictionary D that stores the ASCII characters as an array. Now view the text T_1 as a sequence of requests for characters in D , and implement D as an unsorted array with the MTF heuristic, using $D[0]$ as the front. (Recall that in the original MTF-heuristic, we need to use the maximum index in D as the front, else *insert* would take too long. But in our situation here, we will never insert into D ; it will always store exactly ASCII. Hence we use $D[0]$ as the front, which is more intuitive.)

The actually MTF-transform now consists of converting T_1 into a new text T_2 (with characters that integers in $\{0, \dots, 127\}$) by writing, for every requested character, the index where it was found in D . Let us illustrate this on an example with the following (bizarre) text:

l l a a a t a

We have characters $\{l, a, t\}$ in this text, and for ease of description assume that these are the only characters in our alphabet. The dictionary originally stores these characters in ASCII-order, but the transforms the dictionary with every character as follows:

$\begin{array}{cccccccccccc} \text{0} & \text{1} & \text{2} & & \text{0} & \text{1} & \text{2} & & \text{0} & \text{1} & \text{2} & & \text{a} & \text{0} & \text{1} & \text{2} & & \text{a} & \text{0} & \text{1} & \text{2} & & \text{a} & \text{0} & \text{1} & \text{2} & & \text{t} & \text{0} & \text{1} & \text{2} & & \text{a} & \text{0} & \text{1} & \text{2} \\ \boxed{\text{a}} & \boxed{\text{l}} & \boxed{\text{t}} & \xrightarrow{\text{l}} & \boxed{\text{l}} & \boxed{\text{a}} & \boxed{\text{t}} & \xrightarrow{\text{l}} & \boxed{\text{l}} & \boxed{\text{a}} & \boxed{\text{t}} & \xrightarrow{\text{a}} & \boxed{\text{a}} & \boxed{\text{l}} & \boxed{\text{t}} & \xrightarrow{\text{a}} & \boxed{\text{a}} & \boxed{\text{l}} & \boxed{\text{t}} & \xrightarrow{\text{a}} & \boxed{\text{a}} & \boxed{\text{l}} & \boxed{\text{t}} & \xrightarrow{\text{t}} & \boxed{\text{t}} & \boxed{\text{a}} & \boxed{\text{l}} & \xrightarrow{\text{a}} & \boxed{\text{a}} & \boxed{\text{t}} & \boxed{\text{l}} \\ & & & & \text{1} & & & & \text{0} & & & & \text{1} & & \text{0} & & \text{0} & & \text{0} & & \text{2} & & \text{1} & & & & & & & & & & & \end{array}$

Note that the output 1010021 has quite a few zeroes. In general, if T_1 has a character repeating k times, then T_2 will have a run of $k - 1$ zeroes. Since the input T_1 to the MTF transform is the output of the Burrows-Wheeler transform, it should have (as we will argue below) long runs of characters, and so T_2 will have long runs of zeroes.

Notice another useful property of the output T_2 : small indices are much more likely than large indices. We would typically start the dictionary with ASCII. Text T_1 will be a permutation of the original source text, hence have the same characters. So if the original source text is English, then T_1 will have lots of occurrences of ‘e’ and ‘t’ and exceedingly few occurrences of control-characters. The characters of T_1 will be moved to the front in dictionary D , and hence receive small numbers when they repeat. Therefore the distribution of characters in T_2 (where “character” now means “number in $\{0, \dots, 127\}$ ”) is quite uneven, making it suitable for Huffman-compression. As noted above, the modified RLE-encoding preserves this property, except for character 0.

The MTF transform is another example of a text compression/transform that uses an adaptive dictionary, since the dictionary changes with every step. However, the change happens in a well-defined manner that only depends on the currently encoded character. Therefore the encoder can easily emulate this, and no special tricks are needed for decoding. Algorithm 10.11 and 10.12 shows the corresponding algorithms. Note that the run-time is proportional to the total time that it takes to find the characters in the dictionary; this is $O(|\Sigma_S| \cdot |S|)$ in the worst case, but should be much better in practice since frequently used characters should be found quickly.

Algorithm 10.11: $MTF::encoding(S,C)$

```

Input : Input-stream  $S$  of characters in alphabet  $\Sigma_S$ , output-stream  $C$ 
1 Initialize  $D$  as an array that stores  $\Sigma_S$  in some pre-agreed, fixed order (typically ASCII)
2 while  $S$  is not empty do
3    $c \leftarrow S.pop()$ 
4   for  $i = 0, 1, \dots$  do if  $D[i] = c$  then break
5    $C.append(i)$ 
6   for  $j = i - 1$  down to 0 do                                     // move character to the front
7   | swap  $D[j]$  and  $D[j + 1]$ 

```

Algorithm 10.12: $MTF::decoding(C, S)$

Input : Input-stream C of indices in $\{0, \dots, |\Sigma_S| - 1\}$, output-stream S

- 1 Initialize D as an array that stores Σ_S in some pre-agreed, fixed order (typically ASCII)
- 2 **while** C is not empty **do**
- 3 $i \leftarrow C.pop()$
- 4 $S.append(D[i])$
- 5 **for** $j = i - 1$ down to 0 **do** // move character to the front
- 6 swap $D[j]$ and $D[j + 1]$

6964 10.3.4 Burrows-Wheeler Transform

Now we turn to the ingredient of `bzip2` that makes all the others work: How to transform a text T_0 into a text T_1 that has lots of runs of characters? We will do this in such a way that T_1 is a permutation of the characters of T_0 . In particular, T_1 is no shorter, but also no longer and it has other useful properties. It would be easy to generate a permutation that has lots of runs of characters (e.g., we could simply sort the characters of T_0), but we also must be *lossless*, i.e., given the permutation, we must be able to recover the original. It is not at all obvious that

this can be done. For example, simply sorting the characters is not lossless; the words **good** and **odog** would give the same sorted list of characters.

We will give the Burrows-Wheeler transform here only under the assumption that the text ends with an end-of-text character **\$** that is smaller than all other characters and that occurs nowhere else. To define how it operates, we need the concept of a *cyclic shift*: Given a string S (say as an array $S[0..n-1]$), the i th cyclic shift of S is the string $S[i..n-1] \mathbin{\text{++}} S[0..i-1]$, i.e., the i th prefix of S has been removed from the front and appended at the rear. For example, for the string $S = \text{alf_eats_alfalfa\$}$ the cyclic shifts are shown in the rows of the matrix on the left side of Figure 10.17. (This matrix is also called the *matrix of cyclic shifts*.)

alf_eats_alfalfa\$		\$alf_eats_alfalfa
lf_eats_alfalfa\$a		_alfalfa\$alf_eats
f_eats_alfalfa\$a		_eats_alfalfa\$alf
_eats_alfalfa\$alf		a\$alf_eats_alfalf
eats_alfalfa\$alf_		alf_eats_alfalfa\$
ats_alfalfa\$alf_e		alfa\$alf_eats_alf
ts_alfalfa\$alf_ea		alfalfa\$alf_eats_
s_alfalfa\$alf_eat		ats_alfalfa\$alf_e
_alfalfa\$alf_eats	→	eats_alfalfa\$alf_
alfalfa\$alf_eats_		f_eats_alfalfa\$a
lfalfa\$alf_eats_a		fa\$alf_eats_alfal
falfa\$alf_eats_al		falfa\$alf_eats_al
alfa\$alf_eats_alf		lf_eats_alfalfa\$a
lfa\$alf_eats_alfa		lfa\$alf_eats_alfa
fa\$alf_eats_alfal		lfalfa\$alf_eats_a
a\$alf_eats_alfalf		s_alfalfa\$alf_eat
\$alf_eats_alfalfa		ts_alfalfa\$alf_ea

Figure 10.17: Example of the Burrows-Wheeler encoding. The output-string is bold.

Burrows-Wheeler encoding: The Burrows-Wheeler encoding is now very easily obtained: Take the n cyclic shifts of source text S , sort them lexicographically, and then output the *last* characters of the sorted cyclic shifts. Figure 10.17 (right) shows the cyclic shifts sorted in lexicographic order; the resulting encoded string C is hence

a s f f \$ f _ e _ l l l a a a t a

Why is this useful for **bzip2**? Recall that the objective was to obtain a string where there are lots of long runs of characters. In our example, there are indeed such runs; in particular we have a run of three **l**'s, and a run of three **a**'s. We claim that this is not a coincidence.

Recall that typically texts have lots of repeated substrings; our example had substring **alf** occur three times. This means that there are 3 cyclic shifts of S that begin with **lf** and end with **a**. So these cyclic shifts begin in the same way, and should end up near each other in the sorted order of cyclic shifts. Note that if they end up being consecutive (as they are in our example) then this results in three consecutive **a**'s in the output, hence a run of characters. It

is not guaranteed that repeated substrings lead to repeated characters in the output, because there could be some other cyclic shift that begins with `lf` and ends with something other than `a`. But it is quite likely.

Efficient BWT encoding: It is clear that we can do the Burrows-Wheeler transform in polynomial time: We could explicitly write down the cyclic shifts, sort them (MSD radix sort seems best), and then extract the last characters. The problem is that there are n^2 characters in the n cyclic shifts together, so this would take time $\Theta(n^2)$, which is too slow for practical purposes. A few tricks will help to make this faster.

First, we said to sort the cyclic shifts, but actually for extracting the encoding, it suffices to describe what the sorting would have been, or put differently, to give the *sorting permutation*. So we want a permutation π such that for all $i < j$ the cyclic shift with index $\pi(i)$ is lexicographically smaller than the cyclic shift with index $\pi(j)$. See Figure 10.18 for the sorting permutation π of our example.

start-index of cyclic shift		k	$\pi(k)$	
0	alf_eats_alfalfa\$	0	16	\$alf_eats_alfalfa
1	lf_eats_alfalfa\$a	1	8	_alfalfa\$alf_eats
2	f_eats_alfalfa\$al	2	3	_eats_alfalfa\$alf
3	_eats_alfalfa\$alf	3	15	a\$alf_eats_alfalf
4	eats_alfalfa\$alf_	4	0	alf_eats_alfalfa\$
5	ats_alfalfa\$alf_e	5	12	alfa\$alf_eats_alf
6	ts_alfalfa\$alf_ea	6	9	alfalfa\$alf_eats_
7	s_alfalfa\$alf_eat	7	5	ats_alfalfa\$alf_e
8	_alfalfa\$alf_eats	8	4	eats_alfalfa\$alf_
9	alfalfa\$alf_eats_	9	2	f_eats_alfalfa\$al
10	lfalfa\$alf_eats_a	10	14	fa\$alf_eats_alfal
11	falfa\$alf_eats_al	11	11	falfa\$alf_eats_al
12	alfa\$alf_eats_alf	12	1	lf_eats_alfalfa\$a
13	lfa\$alf_eats_alfa	13	13	lfa\$alf_eats_alfa
14	fa\$alf_eats_alfal	14	10	lfalfa\$alf_eats_a
15	a\$alf_eats_alfalf	15	7	s_alfalfa\$alf_eat
16	\$alf_eats_alfalfa	16	6	ts_alfalfa\$alf_ea

Figure 10.18: The sorting permutation of the cyclic shifts.

How do we find the sorting permutation quickly? The end-of-word character comes in handy here, because it is smaller than all other characters, and therefore the sorting permutation of the cyclic shifts is the *same* as the sorting permutation of the suffixes. (Proving this is left as an exercise.) Therefore all we need to compute is the suffix array, which by definition is the sorting permutation of the suffixes. This takes time $O(n \log n)$ (or even less if we use advanced methods).

With this, we can find the k th character of the Burrows-Wheeler encoding easily, even without ever explicitly writing the matrix of cyclic shifts. Namely, we know that row k of the sorted matrix would hold the string $S[\pi(k)...n-1] + S[0...\pi(k)-1]$. We want the last character

7010 of this cyclic shift, which is hence $S[\pi(k)-1]$. The only exception occurs if $\pi(k) = 0$; then the
 7011 character is $S[n-1] = \$$. Algorithm 10.13 shows how to extract the Burrows-Wheeler encoding
 7012 from the sorting permutation.

Algorithm 10.13: *BWT::encoding(S, C)*

Input : String S as array (not stream), output-stream C . S ends with \$.

```

1  $\pi \leftarrow$  compute the suffix array of  $S$ 
2 for  $k = 0 \dots |S| - 1$  do
3   if  $\pi[k] = 0$  then  $C.append(\$)$ 
4   else  $C.append(S[\pi[k]-1])$ 
```

7013 **Burrows-Wheeler Decoding:** Is the Burrows-Wheeler transform really lossless? It does not
 7014 seem obvious at all that we can recover the original string from the encoded string. It turns
 7015 out to be true (and even quite easy to do, though understanding the correctness is not so easy).
 7016 We will illustrate this in an example first, and then give the algorithm. So let us try to recover
 7017 the source string S from encoding $C = \mathbf{a\ n\ n\ b\ \$\ a\ a}$. For this purpose, let us create the
 7018 matrix M of the cyclic shifts in lexicographic order. (As we will see later, it is not actually
 7019 necessary to create this matrix, everything can be done in an array, but having it will be helpful
 7020 for understanding why the algorithm works correctly.)

- 7021 • We know that the rightmost column of M is C , because that is how the Burrows-Wheeler
 7022 encoding is created. See Figure 10.19(a).
- 7023 • We can also quite easily re-create the leftmost column of M : We know that this contains
 7024 all the leftmost characters of all the cyclic shifts, and they are in alphabetic order (since
 7025 the rows are sorted lexicographically). But the i th cyclic shift starts with $S[i]$, so the
 7026 leftmost column of M contains exactly all characters of S . And we also know that C is a
 7027 permutation of S . So we can obtain the leftmost column of M by putting the characters
 7028 of C in sorted order. See Figure 10.19(b).
- 7029 • Now we can actually figure out $S[0]$. (We will write this next to the $\$$ in the topmost row).
 7030 Study row 4. Here, the rightmost character is $\$$ and the leftmost character is \mathbf{b} . Since
 7031 these are cyclic shifts, and $\$$ is the last character of S , hence $S[0] = b$. In a similar manner,
 7032 studying row 4 and observing that b occurs only once, we can figure out that $S[1] = a$.
 7033 See Figure 10.19(c).
- 7034 • Now, however, we are seemingly stuck. There are three rows where the rightmost character
 7035 is an \mathbf{a} —which of those should we use to obtain $S[2]$?
- 7036 • To break this impasse, let us disambiguate the characters in the rightmost column by
 7037 attaching their row-index, see Figure 10.19(d). Also, for $i = 0, \dots, n - 1$, let w_i be the
 7038 string of the first $n - 1$ characters in row i of M , as illustrated in Figure 10.19(e). For
 7039 example, we have $w_0 = \mathbf{\$ba\dots}$, where we do not know the last three characters yet.
- 7040 • Consider the three strings w_0, w_5 and w_6 that are adjacent to the three copies of \mathbf{a} . Bringing

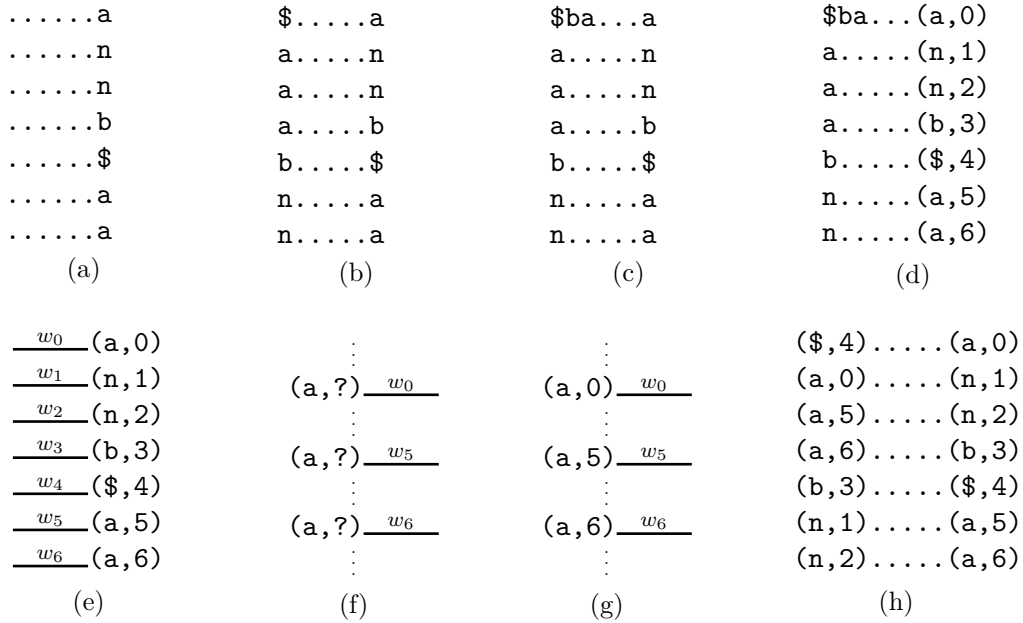


Figure 10.19: An example of Burrows-Wheeler decoding.

a to the front, we get three other cyclic shifts of S : aw_0 , aw_5 and aw_6 . But we know that the **a** in front of w_i has row-index i , so these are actually $(a,0)w_0$, $(a,5)w_5$ and $(a,6)w_6$. See Figure 10.19(f).

- Here is the crucial insight: We *know* that $w_0 <_{\text{lex}} w_5 <_{\text{lex}} w_6$, because the three cyclic shifts w_0a , w_5a and w_6a appear in this order among the lexicographically sorted cyclic shifts. Therefore $aw_0 <_{\text{lex}} aw_5 <_{\text{lex}} aw_6$. But recall that the **a** in front of w_i is (a,i) , so we have

$$(a,0)w_0 <_{\text{lex}} (a,5)w_5 <_{\text{lex}} (a,6)w_6.$$

In particular this tells us exactly *which* **a** on the left side corresponds to which **a** on the right side: they appear in the same order. See Figure 10.19(g). The same holds for all other characters, so we now know exactly the correspondence of characters between the rightmost and leftmost column. See Figure 10.19(h).

- Now we can easily figure out the entire string, because we can read (starting at **\$**) which character follows which. Specifically

$$(\$,4) \rightarrow (b,3) \rightarrow (a,6) \rightarrow (n,2) \rightarrow (a,5) \rightarrow (n,1) \rightarrow (a,0) \rightarrow (\$,4)$$

so the entire string decodes to **banana\$**.

Burrows-Wheeler decoding is one of those algorithms where the algorithm itself is actually very simple; what made the above steps so complicated is arguing that it works correctly. Here are the steps of the decoding summarized (and the pseudocode is in Algorithm 10.14):

- Re-create the first column of matrix M , with characters disambiguated. To do so, copy characters of C and their indices to a new array A , and sort A with a stable sorting method.⁹
- Find the position of the end-of-word character $\$$ in C . The character at the corresponding index in A is the first character of the source text.
- Since each entry in A knows where the character occurred in C , we can repeatedly get the next character. Repeat the process until we again reach character $\$$, and we have then read S in order.

Algorithm 10.14: $BWT::decoding(C, S)$

Input : Coded text C as array (not stream), output-stream S

```

1  $A \leftarrow$  array of size  $n \leftarrow C.size$ 
2 for  $i = 0$  to  $n - 1$  do  $A[i] \leftarrow (C[i], i)$ 
3 Stably sort  $A$  by first aspect
4 for  $j = 0$  to  $n - 1$  do if  $C[j] = \$$  then break           // find $-char
5 repeat                                                     // actual decoding
6    $S.append(\text{first entry of } A[j])$ 
7    $j \leftarrow \text{second entry of } A[j]$ 
8 until last appended character was  $\$$ 
```

Let us see one more example where we do the Burrows-Wheeler decoding directly with this algorithm. Let the coded text C be

$C:$

0	1	2	3	4	5	6	7	8	9	10	11
a	r	d	\$	r	c	a	a	a	a	b	b

After attaching indices this becomes

$A:$

0	1	2	3	4	5	6	7	8	9	10	11
a, 0	r, 1	d, 2	\$, 3	r, 4	c, 5	a, 6	a, 7	a, 8	a, 9	b, 10	b, 11

Now sort A stably to get the the following:

$A:$

0	1	2	3	4	5	6	7	8	9	10	11
\$, 3	a, 0	a, 6	a, 7	a, 8	a, 9	b, 10	b, 11	c, 5	d, 2	r, 1	r, 4

We find $\$$ at index 3 in C , look up that $A[3]$ is (a,7), look up that $A[7]$ is (b,11), and so on until we have the entire string **abacadabra\$**. (It does feel like magic, doesn't it?)

(a, 7) \rightarrow (b, 11) \rightarrow (r, 4) \rightarrow (a, 8) \rightarrow (c, 5) \rightarrow (a, 9) \rightarrow (d, 2) \rightarrow (a, 6) \rightarrow (b, 10) \rightarrow (r, 1) \rightarrow (a, 0) \rightarrow (\$, 3)

What is the run-time? We can sort the characters using bucket-sort; this takes $O(n + |\Sigma_C|)$ time. Everything else takes $O(n)$ time, so the run-time for Burrows-Wheeler decoding is $O(n + |\Sigma_C|)$. The auxiliary space is $O(n)$.

⁹Copying the characters of C to A is a convenience, not a necessity: If $A[j] = (c, k)$ then $c = C[k]$, so it would suffice to store only k and look up the character in C when needed.

10.3.5 bzip2 discussion

bzip2 typically compresses much better than LZW encoding. Consider for example again the text `barbarabarbararen`, where Figure 10.16 shows the transformed (and then compressed text). This achieves 44 bits, in contrast to the $19 \cdot 7 = 133$ bits needed to encode the 19 input characters with 7 bits each in ASCII, a compression ratio of roughly 33%. In Figure 10.13 we saw that the LZW encoding uses 11 code-numbers for the string, and it would be 12 code-numbers if we added the end-of-word character. Since LZW uses (typically) 12 bits per code-number, the LZW-encoding would use 144 bits, even worse than the input and much worse than the result of **bzip2**. (Normally LZW is not quite as bad, but it does need much longer texts to work its magic.)

The price to pay for the good compression is that **bzip2** is quite slow. The run-length encoding takes time $O(n)$ (where we use as n here to denote the maximum length among the texts T_0, T_1, T_2, T_3). The Huffman-encoding takes time $O(n + |\Sigma| \log |\Sigma|)$. The MTF-transform takes time $O(n|\Sigma|)$ in the worst case since the lookup for each character can take $\Theta(|\Sigma|)$ time. (In real life, the frequently-used characters are soon in the front, and so this should often be faster.) The time for the Burrows-Wheeler transform is $O(n \log n)$ if done via suffix arrays. In total the run-time hence is $O(n(|\Sigma| + \log n))$, with a large constant since many algorithms need to be run, and some of them (especially the suffix array construction) are fairly complex. Especially for larger alphabets, this can be quite slow.

Interestingly enough, decoding is faster in **bzip2** than encoding: The Burrows-Wheeler-transform can be decoded in linear time without need to anything more complicated than bucket-sort. So the run-time is $O(n|\Sigma|)$ (and often better, since the bottleneck is the MTF-transform decoding), and all algorithms are straightforward to implement.

One drawback of **bzip2** is that it needs the entire source text at once, because to encode we must look at various characters of the cyclic shifts to sort them. Thus, in contrast to previous schemes, it is *not* possible to implement **bzip2** such that both input and output are streams.

Summarizing, while **bzip2** compresses very well, it could still be improved further, and research in compression techniques is ongoing.

10.4 Take-home messages

- There is no method that can compress *all* texts. But for typical English text, or any other text that should have many repeating substrings, significant compression can be achieved in practice.
- We can compute the optimum encoding that uses prefix-free binary encodings. But the resulting compression does not work particularly well in practice.
- Among the algorithms presented here, **bzip2** (which incorporates many different compression-ideas) achieves the best compression ratio, but is somewhat slow to compute and not suitable for streaming.

10.5 Historical remarks

The problem of data compression became of interest as soon as any sort of data storage was feasible. Huffman compression was invented by David Huffman and first published in 1952 [Huf52]. It is the simplest case of *entropy encoding* where the codewords are based on the frequency of characters and come close (at least in some situations) to the entropy lower bound proved by Shannon [Sha48]. There are further improvements to Huffman compression, in particular so-called arithmetic coding. See for example the book by MacKay [Mac03] for further reading.

The idea of compression by listing runs was developed in 1967 by Robinson and Cherry for the purposes of compression television signals [RC67]. Elias- γ codes were developed by Elias in 1975 [Eli75]; he also developed a number of related codes (called Elias- δ and Elias- ω codes).

LZW encoding was developed originally by Ziv and Lempel in 1978 [ZL78] and then further improved by Welch in 1984 [Wel84]. In its original form it did not use tries for storing the dictionary, instead it stores the dictionary in a table, using hashing. Experiments have shown that in practice hashing appears to be faster [FK17], especially if combined with other tricks such as rolling fingerprints.

LZW is also an interesting case study of the history of development of programs and the debate of patents for computer software. Welch's improvement was widely picked up by computer scientists and implemented in numerous standard compression algorithms, such as **compress** and **GIF**. Unbeknownst to many of these users, multiple patents had been filed both on the Lempel-Ziv algorithms and on LZW. Major controversy erupted when Unisys (the holder of the patent on LZW) tried to enforce the patent in the 1990s and asked users to pay a license fee. Some users (such as Adobe, which used LZW for compression of postscript files) complied. But many others did not, and either used LZW-encoded files such as **GIF** illegally, or abandoned those file-formats and used others such as **PNG** (which has been developed precisely in response to the patent-issues). Nowadays, using **GIF** is no longer illegal (the patents expired in the early 2000s), but it has lost the "race" for predominance in picture-storing-techniques even though it was the better compression at the time.

The Burrows-Wheeler transform was (apparently) invented by David Wheeler in 1983, and published jointly with Michael Burrows in 1994 as a technical report at DEC. Using it to achieve high compression via bzip2 was proposed by Julian Seward in 1996 according to its web site, www.bzip.org. It was specifically developed to avoid any patent-issues, which is one of the reasons why it uses Huffman encoding rather than the (already-known-at-the-time, but also patented) arithmetic encoding.

Many comparative studies have been done between various compression methods, evaluating them based on the achieved compression ratios as well as the encoding and decoding time. See for example [BEHW15] for a relatively recent study.

Chapter 11

External Memory

Contents

11.1 Introduction	337
11.1.1 The external memory model	338
11.1.2 Stream-based algorithms	340
11.2 External sorting (cs240e)	341
11.2.1 d -way merge	342
11.2.2 d -way merge-sort	343
11.3 External dictionaries	345
11.3.1 The overall idea	346
11.3.2 2-4-trees	347
11.3.3 Red-black trees (cs240e)	355
11.3.4 a - b -trees	356
11.3.5 B -trees	361
11.3.6 B -tree variations (cs240e)	363
11.4 Extendible hashing	366
11.4.1 Tries of blocks	367
11.4.2 A few discussion items	369
11.4.3 Avoiding the trie	370
11.5 Take-home messages	371
11.6 Historical remarks	371

11.1 Introduction

So far in these notes, we have been working in the RAM model, which assumes that a computer has infinite memory, and that all its memory cells are equal. The first assumption is not too unrealistic, because while a computer has a finite amount of memory, we can make it near-infinite by considering external memory devices, such as flash drives or cloud storage (decades ago, tapes were used for this, and much of the names used in this chapter comes from the days

of tape drives). But access to such external memory devices is orders of magnitudes slower than access to internal memory, so the second assumption is very unrealistic.

We will hence study in this chapter a different kind of computer model where we consider different speeds for accessing memory. If we wanted to be realistic, then we should have a whole hierarchy of speeds, because there is not only the gap between ‘memory is in the computer’ and ‘memory is on flash drives / cloud storage’, but there are also different levels within a computer. In particular, access to memory cells in the cache is much faster than accessing those in RAM, which in turn are much faster accessed than those in the internal disk drive, which in turn is faster than accessing an external disk drive or the cloud. (There are even more levels within a computer, but you get the idea.)

The closer the memory cell is to the CPU of the computer, the faster it is accessed, but also the more expensive it is to add more such memory cells. Table 11.1 gives very rough estimates of the speed and cost of various levels of memory.¹ In particular note that the CPU can load 1G from RAM literally in the blink of an eye (which is around 100-150ms), while loading it from cloud storage means you might as well get up and take a stretch. The speed-difference is vast!

	RAM	internal drive		cloud storage
		SSD	hard drive	
Typical speed	30Gbps	4Gbps	150Mbps	50Mbps
Wait for 1GB	33ms	250ms	7s	20s
Typical Size	4-16GB	128-512GB	1-8TB	∞
Typical price	\$10/GB	\$1/GB	\$40/TB	\$15/TB/year

Table 11.1: Typical access-speeds, sizes and prices of various levels of memory.

11.1.1 The external memory model

Rather than trying to model all these levels of memory-access-speed in a computer, we define a model that has two levels; this can then be used to model any one of the level-gaps (e.g. RAM to internal drive, or internal drive to cloud) listed above. The naming that we use here is mostly inspired by the gap between the internal drive and the external storage. Our *external memory model* is illustrated in Figure 11.2 and defined as follows:

- We have *external memory* which for all practical purposes is unlimited. (The name for this is not at all standardized in the literature, and other names for ‘external memory’ are ‘tape’, ‘cloud’, ‘file’, or ‘disk’, depending on which kind of gap is modeled. We will stick with ‘external memory’ throughout.)

¹This is based on a random sample of devices/services taken in 2020, and ignoring numerous other factors that might impact the loading speed. So it is a crude estimate, but the orders of magnitude of difference should be approximately correct.

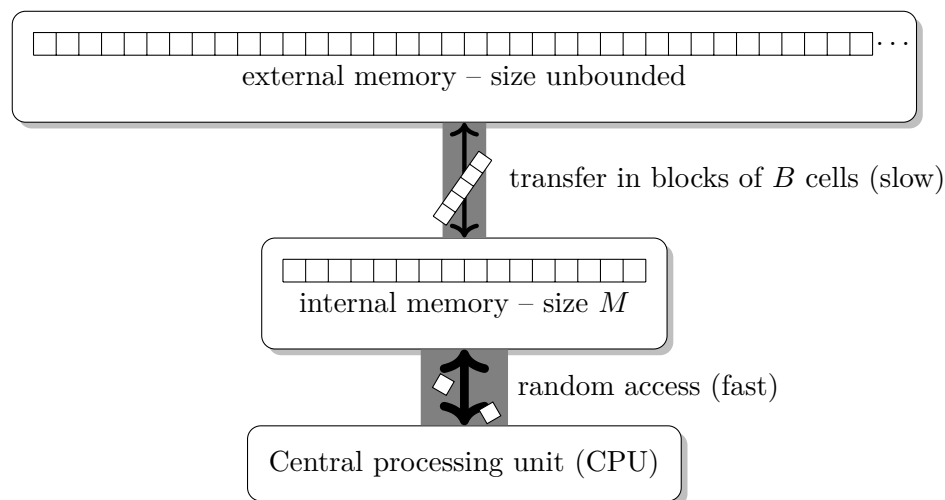


Figure 11.2: The external memory model.

- We have a (relatively small amount of) *internal memory*. (Other names for ‘internal memory’ are ‘main memory’, ‘CPU’, or ‘cache’.) The size of the internal memory is denoted by M .
- Transferring memory from external to internal memory or vice versa is very slow. Our overall goal is to minimize the number of such *block transfers* (also called ‘page load’, ‘page fault’, ‘disk I/O’, ‘disk transfer’, or ‘I/O transfer’).
- One block transfer copies not only one memory cell, but actually copies a whole consecutive *block* of memory cells. (Other names for ‘block’ are ‘page’, ‘line’ or ‘packet’.) There are historical reasons why transfers happened in whole blocks (e.g., on magnetic tapes the tape-head can read a whole chunk at once). But it is also appropriate in today’s cloud computing, because traffic over the internet is sent in packets of information. This *blocking* is a good idea even if it is not required for technical reasons. The *principle of spatial locality* says that if we need data, then it is quite likely that we will need nearby data in the near future. So by transferring a whole block, we may be saving future block transfers. The size of a block (i.e., the amount of memory cells transferred at once) is denoted by B . Its value depends on which gap within the computer architecture is considered: When transferring from RAM to CPU, the block-size is typically in the range of a few dozen bits, whereas from internal drive to the cloud it is closer to a few thousands or millions. The number B technically measures the number of memory cells, which means that when we store more complicated objects such as nodes, we will be able to have $\Theta(B)$ nodes per block, but the actual number of nodes that fit into a block is smaller. We sometimes use b for this number.
- As before, we use n to denote the total size of the data that we wish to store or manipulate.

This is now a truly humongous numbers, typically a few terabytes or even petabytes and beyond.

- Note that the data needs roughly n/B blocks to be stored. We will generally not be concerned with rounding issues, i.e., we will write n/B rather than the more accurate $\lceil n/B \rceil$; since n is so large it does not hurt much to make it a little larger so that it is divisible by B .

Designing and analyzing algorithms in the external memory model. Accessing internal memory is blazingly fast compared to block transfers. In fact, it is so much faster than we will *completely* disregard the amount of time for computation in internal memory. This is clearly not realistic, but can serve as a useful model for designing data structures and algorithms that deal well with external memory.

When designing algorithms in the external memory model, we assume that M and B are both known to the algorithm designer. This is a bit unfair (these are not easy to find out for any particular computer and frequently change when upgrading machines) but will make our life easier. There is some research into *cache-oblivious* algorithms and data structures, which are designed to work well in the external memory model *without* knowing M and B , but this is beyond the scope of cs240.

We also assume that $B \ll M \ll n$, i.e., B is significantly smaller (usually by some order of magnitudes) than M , and M in turn is significantly smaller than n . However, B and M are big enough that we do not consider them to be constants; we will state our number of block-transfers in terms of all three parameters.

This external memory model gives us a whole new way to consider run-time. Suddenly we *only* count block transfers and do not care about anything done in internal memory. As such, one should re-visit *all* data structures and algorithms in this model. We know that an algorithm with run-time $\Theta(f(n))$ also uses $O(f(n))$ block-transfers, because at worst every primitive operation can trigger one block-transfer. But we would like to do better, and ideally use only $\Theta(n/B)$ block-transfers (the minimum required if we want to look at all input data).

11.1.2 Stream-based algorithms

Recall that we introduced streams in Section 10.0.1 and wrote many of the algorithms in Chapter 10 using only streams for input and output. This was done because streams adapt well to the external memory model, and so these algorithms immediately also work well in the external memory model.

Recall that in an input-stream we can only access the item at the top, and in an output-stream we can only append an item at the end (the ‘tail’). We assume that streams are stored in external memory such that consecutive items of the stream are in the same block (e.g. as array). We then need just two blocks in internal memory at any given time, one for the input and one for the output. Whenever the block of input has been used up, we load the next block of the input stream into internal memory. Whenever the block of output has been filled, we write it

into external memory and clear it in internal memory. Thus it takes $O(n/B)$ block transfers to get all input and $O(s/B)$ block transfers to write the output (where s is the output-size). See Figure 11.3.

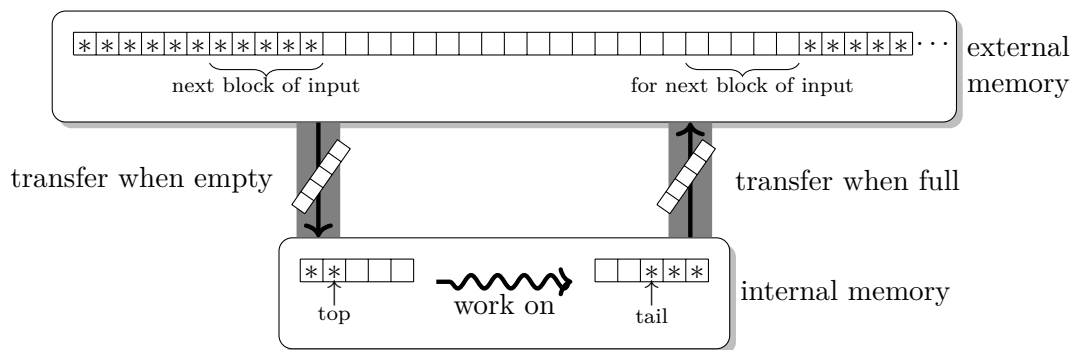


Figure 11.3: Streams naturally adapt to the external memory model.

With this, any method that uses streams for the input and output and that uses little auxiliary space will adapt automatically to external memory and use $\Theta(n/B)$ block transfers. Examples include Lempel-Ziv-Welch encoding and prefix-free-encoding if the encoding trie is mutually agreed upon, but *not* bzip2 (due to the Burrows-Wheeler-Transform). Huffman-encoding uses $\Theta(n/B)$ block transfers, but more than Lempel-Ziv-Welch, since it needs to go through the input-stream twice, once to compute frequencies and once to do the actual encoding.

Pattern matching with the Knuth-Morris-Pratt algorithm or the Boyer-Moore algorithm can easily be expressed using a stream for text T (we look at only one character of T at a time, and once we are done with it we never need to re-visit it). So this can be done with $O(n/B)$ block transfers in the external memory model, presuming the pattern is small enough to fit into internal memory. On the other hand, suffix trees and suffix arrays need to know the entire text for the construction, and so do not easily adapt to external memory.

11.2 External sorting (cs240e)

Now we study how to sort in the external memory model. An algorithm such as *heap-sort* needs to access places all over the input array, which would make it difficult to do this with few block transfers. But one algorithm adapts well to the external memory model: *merge-sort* from Chapter 1. Its subroutine *merge* can easily be rephrased using streams; see Algorithm 11.1. Thus *merge* uses $2n/B$ block transfers. Recall that *merge-sort* has a recursion depth of $\approx \log_2 n$. We therefore use $\approx 2(n/B) \log_2 n$ block transfers in total. This is achieved even without knowing B , as long as the streams handle block-transfers correctly.

So *merge-sort* seems a good choice for sorting with external memory. We can, however, improve it further with a small tweak to the *merge*-routine.

Algorithm 11.1: *merge*(S_1, S_2, S) // based on streams

Input : Input-streams S_1, S_2 contain items in sorted order. Output-stream S .

```

1 while  $S_1$  or  $S_2$  is not empty do
2   if  $S_1$  is empty then
3      $S.append(S_2.pop())$ 
4   else if  $S_2$  is empty then
5      $S.append(S_1.pop())$ 
6   else if  $S_1.top() < S_2.top()$  then
7      $S.append(S_1.pop())$ 
8   else
9      $S.append(S_2.pop())$ 

```

11.2.1 *d*-way merge

The *merge* routine could also be called a *2-way merge* because we merge two input-streams into one output-stream. To use even fewer block transfers when sorting with external memory, we now generalize this to a *d-way merge* where we merge d streams at once (for some number d of our choosing). To do so, we must quickly determine which of the d streams contains the currently smallest item. For two streams we did a direct comparison between the items at the top of a stream. When we have a larger number d of streams, doing a direct comparison would be both more cumbersome to code and too slow. Instead, we store the current tops of the streams in a *min-oriented* priority queue and repeatedly extract the minimum. To be able to then advance the correct stream, the items in the priority queue will also store information about the stream that they belong to. Algorithm 11.2 gives the code for *d*-way merging.

Algorithm 11.2: *d*-way merge(S_1, \dots, S_d, S)

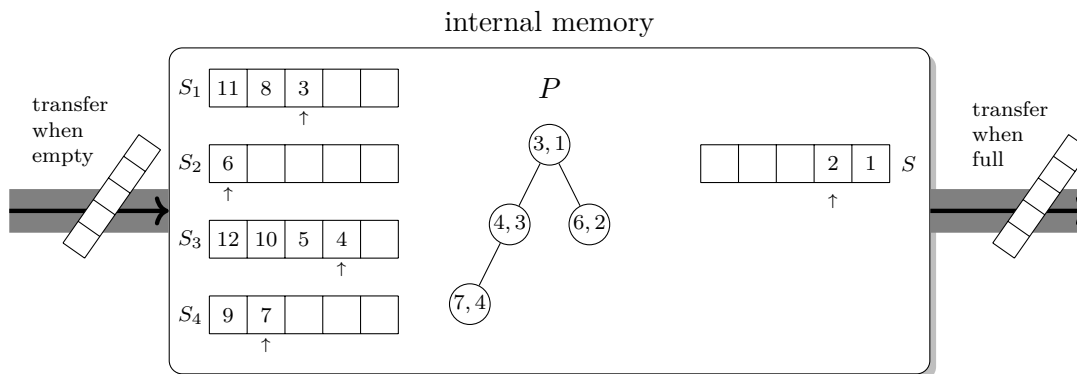
Input : Input-streams S_1, \dots, S_d that are in sorted order. Output-stream S .

```

1  $P \leftarrow$  empty priority queue that stores pairs and is min-oriented w.r.t. first aspect
2 for  $i \leftarrow 1$  to  $d$  do  $P.insert(\langle S_i.top(), i \rangle)$  // Populate priority queue
3 while  $P$  is not empty do // repeatedly extract minimum
4    $\langle x, i \rangle \leftarrow P.deleteMin()$ 
   // Item  $x$  is top-item of stream  $S_i$ 
5    $S.append(S_i.pop())$ 
6   if  $S_i$  is not empty then  $P.insert(\langle S_i.top(), i \rangle)$ 

```

The reader may wonder why we even bother with the priority queue P . Presumably P (which has size $\Theta(d)$) is small enough that it will fit into internal memory, so all operations on P are within internal memory and do not count. True, but *d*-way merge is a useful concept even

Figure 11.4: d -way merging with a min-oriented priority queue.

for algorithms not concerned with external memory. Also, using a priority queue takes no more space than an unsorted array (recall that heaps are stored in arrays), and there is only small overhead for inserting at the price of much faster search.

Let us analyze the run-time of d -way merge sort first in the RAM model (i.e., counting internal memory operations but not caring about block transfers). Priority queue P has size at most d at all times, so we need $\log d$ time to find the minimum. We extract the minimum (and replace it by another item) a total of n times, so the run-time for d -way merge in the RAM model is $\Theta(n \log d)$.

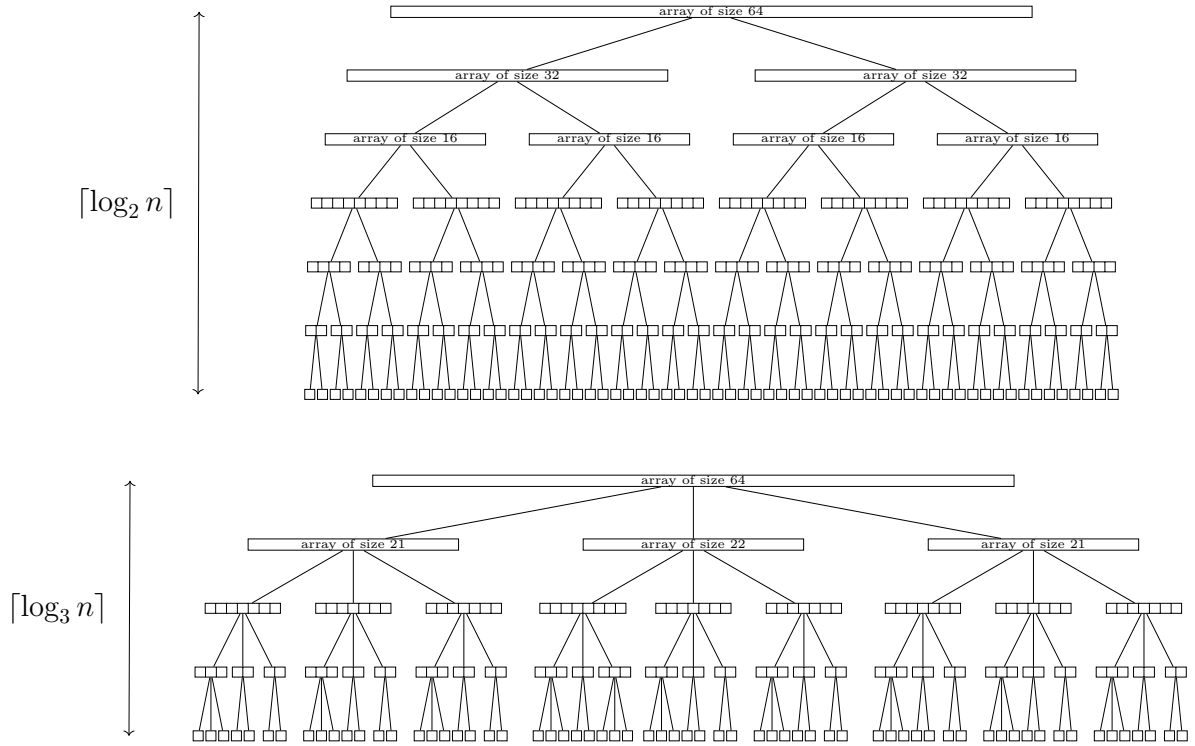
When counting the number of block transfers, we observe that again we are only using input-streams and output-streams. At any given time we therefore have $d+1$ blocks in internal memory (one from each input-stream and one for the output-stream). So assuming that these blocks and P fits into internal memory (which will depend on our choice of d), we use $\Theta(n/B)$ block transfers for d -way merge.

11.2.2 d -way merge-sort

Recall that *merge-sort* operates by splitting the input array in half, then splitting the sub-arrays in half, etc., until it has reached arrays of size 1. Then it goes back up and merges sub-arrays in the reverse order. See also Figure 11.5.

We can now naturally generalize this to d -way *merge-sort* where we split the input array into d (approximately equal-sized) parts, and recursively split until all parts have size 1. Then we go back up and use d -way *merge* to merge the parts together. See Figure 11.5, and notice that with this idea, the recursion-depth is $\lceil \log_d n \rceil$.

Each execution of d -way merge takes $O(N \log d)$ time, where N is the number of items in the sub-array that we consider. Looking at all the executions of d -way merge that happen within one level of the recursion, they all together hence take time $O(n \log d)$, since every input item is in exactly one of the sub-arrays. Assuming time units are such that on each level we spend at

Figure 11.5: The idea of merge-sort, and generalizing it to d -way merge-sort.

most $n \log d$ time units, the total run-time (in the RAM model) is

$$(\text{recursion depth}) \cdot (\text{work on each level}) \leq (\log_d n) \cdot (n \log d) = \frac{\log n}{\log d} \cdot n \log d = n \log n.$$

So in the RAM model, d -way merge-sort has run-time $\Theta(n \log n)$. This is the same asymptotic run-time as merge-sort, and in practice d -way merge-sort is likely slower because of the overhead for the priority queue.

However, d -way merge-sort is much better than merge-sort in the external memory model with an appropriate choice of d . Choose d to be maximal such that Figure 11.4 is accurate, i.e., the priority queue P of d -way merge and $d+1$ blocks all fit into internal memory. This holds for some $d \in \Theta(M/B)$. With this, we need no block transfers, except the ones used by the streams. Each execution of d -way merge loads the elements once from external memory and writes them back once. Thus on each level of the recursion we use at most $2n/B$ block transfers for all executions of d -way merge on this level together.

With this, the number of block transfers is asymptotically at most

$$(\text{recursion depth}) \cdot (\text{transfers on each level}) \leq (\log_d n) \cdot (2n/B)$$

Actually, we can make the bound a bit stronger. We illustrate this first on the example $d = 3$ in Figure 11.5. If (say) $M = 8$, then the arrays of size 7 and 8 on level 2 each fit into internal memory. In consequence, on the last three layers we do not need any block transfers at all; the entire sub-array can get sorted in internal memory. Generalizing to arbitrary d and M , observe similarly that the bottom-most $\log_d(M)$ layers do not need block transfers, because the sub-array then has size at most M and fits into internal memory. Thus

$$\#\{\text{layers that use block transfers}\} = \log_d(n) - \log_d(M) = \log_d(n/M) = \frac{\log(n/M)}{\log(d)}.$$

Recall that $d \in \Theta(M/B)$. For the following re-formulations, let us assume that $d = c \cdot \frac{M}{B}$ for some constant $c > 0$. Then the total number of block transfers used by d -way merge-sort is proportional to

$$\frac{\log(n/M)}{\log d} \cdot \frac{2n}{B} \leq \frac{\log(n/M)}{\log(cM/B)} \cdot \frac{2n}{B} = \frac{\log(n/M)}{\log(M/B) + \log c} \cdot \frac{2n}{B} \in \Theta\left(\frac{\log(n/M)}{\log(M/B)} \cdot \frac{n}{B}\right)$$

and we have hence shown:

Theorem 11.1. *We can sort n numbers in the external memory model using $\Theta\left(\frac{\log(n/M)}{\log(M/B)} \cdot \frac{n}{B}\right)$ block-transfers.*

One can show that in the external memory model, any comparison-based sorting algorithm requires $\Omega\left(\frac{\log(n/M)}{\log(M/B)} \cdot \frac{n}{B}\right)$ block transfers. The proof of this is beyond the scope of cs240. As such, d -way merge-sort is asymptotically optimal with respect to the number of block transfers.

Let us see on a sample set of sizes why d -way merge-sort is much better than using the standard sorting algorithms in the external memory model. Consider the following (reasonably realistic) example of $n = 2^{50}$, $M = 2^{30}$ and $B = 2^{15}$. With *heap-sort*, we would use roughly $n \log n$ comparisons. (For ease of computation, we will ignore here the constants hidden in the asymptotic notation.) Since *heap-sort* does comparisons all over the array, it is fair to assume that each such comparisons requires a block transfer. So we have perhaps $50 \cdot 2^{50}$ block transfers, a truly large number. With *merge-sort*, the numbers are slightly better, because *merge* requires only $2n/B = 2^{36}$ block transfers. We have $\log n$ rounds of merging, so this gives roughly $50 \cdot 2^{36}$ block transfers. But now consider d -way merge-sort. For ease of computation, we assume here that $d = M/B = 2^{15}$. Again we require $2n/B = 2^{36}$ block transfers for each round of d -way merge, but now we have only $\lceil \log(n/M)/\log(d) = \log(2^{20})/\log(2^{15}) \rceil = \lceil 20/15 \rceil = 2$ rounds! We are many orders of magnitude faster than heapsort in this example (and experiments have shown that this is true in general).

11.3 External dictionaries

We now turn to the question of how to implement ADT Dictionary in the external memory model. We can easily do this with $O(\log n)$ block transfers per operation by using any implementation that has $O(\log n)$ run-time (such as AVL-trees). But we would like to do better by

restricting which items of a binary search tree are stored in one block, and hence saving block transfers.

11.3.1 The overall idea

We first give an outline of the idea, illustrated in Figure 11.6. Assume we have a binary tree that is perfectly balanced. (For this outline, we will assume that n is divisible as needed, so all levels of the tree are full.) The idea is to divide the tree into disjoint subtrees such that each subtree fits into one block. See Figure 11.6.

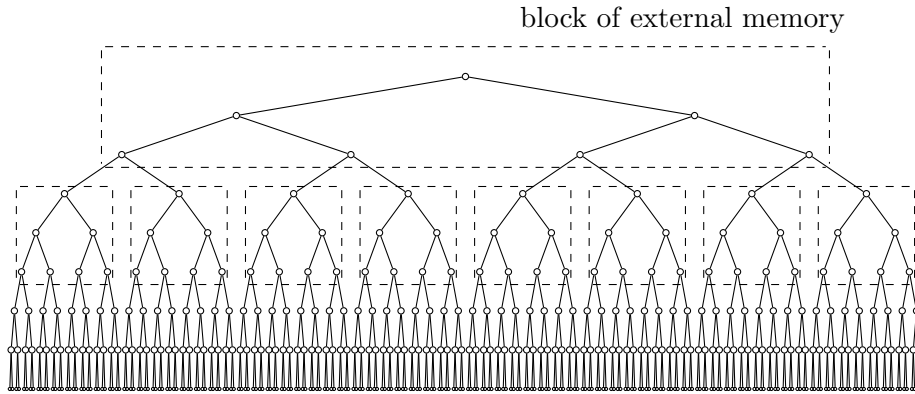


Figure 11.6: The idea that leads to a B -tree.

Let us do the math to see just how many block transfers this would save. Let b be the maximal power of 2 such that $b - 1$ nodes can be stored in one block, (Put differently, a complete binary tree of height $\log b - 1$ fits into one block of size B .) In the example $b = 8$. We typically have $b < B$ since nodes need more than one computer word of storage space, but $b \in \Theta(B)$.

Now split the tree into blocks that each holds a subtree of height $\log b - 1$. In consequence, during a search from the root to some leaf, we need to load only one block for every $(\log b)^{\text{th}}$ of the $\log n$ levels. Assuming $b = c \cdot B$ (for some constant c), the number of block transfers becomes

$$\frac{\log n}{\log b} = \frac{\log n}{\log(c \cdot B)} = \frac{\log n}{\log B + \log c} \in \Theta\left(\frac{\log n}{\log B}\right) = \Theta(\log_B n).$$

One can argue that $\Omega(\log_B n)$ block transfers are *required* for searching in the external memory model (details are omitted), so this is asymptotically optimal.

To see the difference that the switch from $\log n$ to $\log_B n$ makes, consider again the example of $n = 2^{50}$ and $B = 2^{15}$. (The size of M is irrelevant here.) Let us make the reasonable assumption that four computer-words suffice to store a node, so $b = \frac{1}{4}B = 2^{13}$. Then a search that loads a block per level would take $\log n = 50$ block-transfers, while a search that only loads a block every $(\log b)^{\text{th}}$ level would only need $\lceil \log n / \log b \rceil = \lceil 50/13 \rceil = 4$ block-transfers. This is a *huge* speedup.

Unfortunately, maintaining such a partition-structure while inserting and deleting into the tree is difficult. If we used (say) an AVL-tree, then its structure undergoes major changes during rotations, and rearranging the tree while keeping the above layout on blocks seems impossible. Therefore we will design an entirely new realization of ADT Dictionary from scratch. (It turns out that this would also have given a fairly competitive realization of ADT Dictionary in the RAM model.)

The limits of amortization: (cs240e) We have argued above that the rotations make AVL-trees unsuitable for our overall idea. But what about scapegoat trees? They do not use rotations, so the structure could be mostly maintained. And the occasional complete rebuild might be slow, but not affect the average over all operations, right?

Unfortunately, while the average over all operations is indeed small, in this setting where n is huge bounding the average is no longer sufficient. We would occasionally have to rebuild the *entire* tree. This means loading *all* n key-value pairs, which means $\Omega(n/B)$ block transfers for sure, and possibly more depending on how much information we need to move between different blocks while rebuilding. So one operation could suddenly take minutes where most of the time it took seconds. This is likely not acceptable—we really want an implementation where the *worst-case* number of block transfers is $O(\log_B n)$, every time.

For the same reason, other tricks that used amortization, such as lazy deletion, also become unacceptably slow when dealing with the truly huge input-sizes that occur with external memory.

11.3.2 2-4-trees

We now explain how to get from the general idea of the previous section to the specific data structure defined below. Look at the tree in Figure 11.6 again, and now view the blocks (dashed squares) as nodes. These are *multi-way* nodes, i.e., each node stores $b - 1$ key-value pairs and has b sub-trees (where b as before is such that $b - 1$ nodes can fit into a block). Moreover, there is a strict relationship in the order of keys between those that are at the sub-trees and those that are stored in the nodes. The idea for our data structure is to use such multi-way nodes, but to be a bit more flexible in the number of stored key-value pairs so that we can insert and delete while doing only few rearrangements with respect to which key-value pairs are stored in which block.

Recall that in a multi-way tree, the *arity* or *order* is the maximum permitted number of subtrees at a node. In this subsection, we explain our data structure only for the special case of arity $b = 4$. Such a small value of b makes little difference for the number of block transfers (the improvement is at best a factor of 2), but gives examples that are easier to draw. The resulting realization of ADT Dictionary is called a *2-4-tree* and illustrated in Figure 11.7.

Definition 11.1. A 2-4-tree is a multi-way tree with the following structural property:

1. Every node is one of the following three types:
 - A 1-node: It stores one key-value pair and has two subtrees.

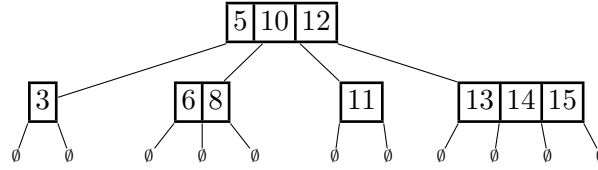


Figure 11.7: An example of a 2-4-tree.

- A 2-node: It stores two key-value pairs and has three subtrees.
 - A 3-node: It stores three key-value pairs and has four subtrees.
2. All empty subtrees are on the same level.
- It also has the following order-property: At a d -node, the keys and subtrees can be ordered as $\langle T_0, k_1, T_1, \dots, k_d, T_d \rangle$ such that all keys in T_i fall into the range between k_i and k_{i+1} (assuming $k_0 := -\infty$ and $k_{d+1} := \infty$).

Note in particular that a d -node has *exactly* $d+1$ subtrees, and all our operations need to maintain this property by adding empty subtrees (but only in the last level!). Figure 11.8 illustrates the order-property at a 3-node, the other types of nodes are similar.

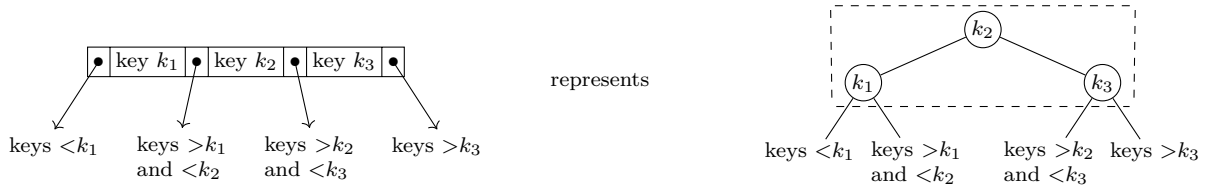


Figure 11.8: The order-property at a 3-node.

The restriction that all empty subtrees are on the same level is *crucial* for the height-bound that we develop below. Note that we could not have imposed such a requirement in a binary search tree, because this would have uniquely fixed the structure and hence restricted the number of keys to be of the form $n = 2^h - 1$ for some integer h . But in a multi-way tree where nodes can store more than one key at a node, we have flexibility in the structure even when all empty subtrees are on one layer, and hence (as we will see below) have a 2-4-tree for any value of n . Now we prove a bound on the height.

Lemma 11.1. Any 2-4-tree T with n keys has height at most $\log_2(\frac{n+1}{2})$.

Proof. Let h be the height, so layers $0, \dots, h$ of T contain nodes, not empty subtrees. By induction one easily shows that layer i for $i \leq h$ contains at least 2^i nodes. This clearly holds on layer 0, and if it holds for layer $i-1$, then each of the 2^{i-1} nodes in layer $i-1$ has at least two subtrees. None of the subtrees in layer i are empty by $i \leq h$ and the structural condition, so there are at least 2^i nodes in layer i . Therefore in total there are at least $1 + 2 + \dots + 2^h$ nodes, and each of them stores at least one key. Therefore $n \geq 1 + 2 + \dots + 2^h = 2^{h+1} - 1$, and reformulating gives $h \leq \log(n+1) - 1 = \log_2(\frac{n+1}{2})$. \square

Search: Let us study how to perform the usual ADT Dictionary operations, i.e., *search*, *insert* and *delete*. Operation *search* is essentially unchanged: Either the key is stored in the current node, or the order-property tells us in which subtree it would have to be, and we follow those links until we either find the key or hit an empty subtree. See Algorithm 11.3.

Algorithm 11.3: $24Tree::search(k, v \leftarrow root, p \leftarrow NIL)$

Input : Key k to search, subtree v where we search, parent p of v

```

1 if  $v$  represents empty subtree then return “not found, would be in  $p$ ”
2 Let  $\langle T_0, k_1, \dots, k_d, T_d \rangle$  be the key-subtree list at  $v$ 
3 if  $k < k_1$  then  $24Tree::search(k, T_0, v)$ 
4 else
5    $i \leftarrow$  maximal index such that  $k_i \leq k$ 
6   if  $k_i = k$  then return  $k_i$  and its associated value
7   else  $24Tree::search(k, T_i, v)$ 

```

Note that the time spent at each node is significantly more in $24Tree::search$ than it was in a binary search tree. In particular, in a binary search tree we do only one comparison at each node, whereas here we may do multiple comparisons to find the maximal index i with $k_i \leq k$. For 2-4-trees, the order is at most 4, hence there are at most 3 keys and we can find i with at most 2 comparisons, so this is still constant time. But if (as we will do soon) the order of the tree is bigger, then this overhead at each node will be non-constant.

Insert: Now we turn to insertion. This is quite different from what we did for binary search trees, because nodes can now store more than one key. The first step is, as usual, to search for the key k that we want to insert. Observe that when $24Tree::search$ fails to find key k , then it returns the leaf where k should have been. If we are lucky then this leaf stores currently only one or two keys. If so, then we can simply add k (and an empty subtree) to the leaf and we are done. See Figure 11.9.

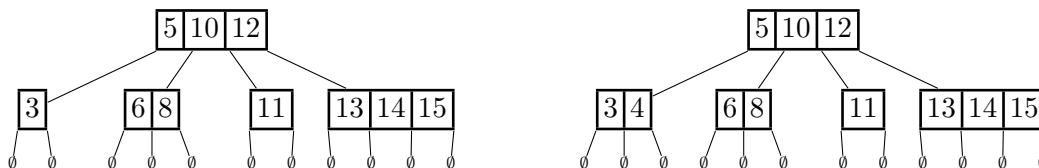


Figure 11.9: Insertion into a 2-4 tree, easy case. We inserted key 4.

However, if we are unlucky, then the leaf v where k should be already has the maximum possible number of keys. We call this situation an *overflow* because adding k to the node would cause it to violate the structural property because it is too full. To deal with overflow, we do a local rearrangement at node v called a *node split*. Add k and an empty subtree to node v , so

7458 that it now has 4 keys and 5 subtrees. Determine the third smallest of those keys and move it
 7459 to the parent p of v . (We could equally have taken the second smallest of those keys, but will
 7460 stick with the third smallest to keep the code simple.) This splits node v into two nodes v' and
 7461 v'' . See Figure 11.10. One verifies that that the number of subtrees exactly works out: after
 7462 suitable rearrangements each of the nodes p, v', v'' has exactly one more subtree than it stores
 7463 keys, and the order property is respected.

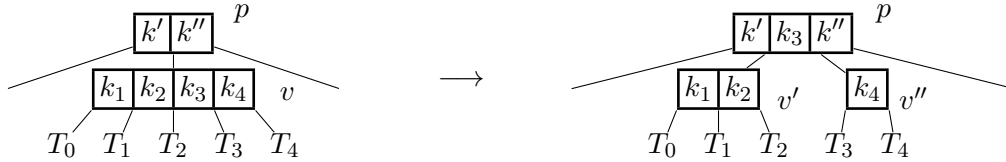


Figure 11.10: A node-split in a 2-4-tree.

7464 This operation has added a new key and subtree to parent p . If p had only one or two keys
 7465 before, then this is fine, but otherwise parent p in turn now has overflow. If so, then we repeat
 7466 at the parent, and so on. If we repeat all the way to the root, and the root has overflow, then
 7467 the root in turn will be split into two nodes, and a new node that becomes the new parent will
 7468 be created. In this manner, a 2-4-tree grows ‘upward from the root’, rather than downward from
 7469 the leaves as it would have happened in binary search trees. See Algorithm 11.4 for the code
 7470 and Figure 11.11 for an example.

Algorithm 11.4: *24Tree::insert(k)*

```

1  $v \leftarrow 24Tree::search(k)$                                 // leaf where  $k$  should be
2 Add  $k$  and an empty subtree at the appropriate place in the key-subtree-list of  $v$ 
3 while  $v$  has 4 keys do                                     // overflow leads to node split
4   Let  $\langle T_0, k_1, \dots, k_4, T_4 \rangle$  be key-subtree list at  $v$ 
5   if  $v$  has no parent then                                  // tree grows upwards
6     create a parent of  $v$  without keys
7    $p \leftarrow$  parent of  $v$ 
8    $v' \leftarrow$  new node with keys  $k_1, k_2$  and subtrees  $T_0, T_1, T_2$ 
9    $v'' \leftarrow$  new node with key  $k_4$  and subtrees  $T_3, T_4$ 
10  Replace  $\langle v \rangle$  by  $\langle v', k_3, v'' \rangle$  in key-subtree-list of  $p$ 
11   $v \leftarrow p$ 

```

7471 **Delete:** Now we turn to deletion. The first part of this is quite similar to binary search trees.
 7472 We want the structural change to occur at a node v that has an empty subtree. Thus, first find
 7473 the node that contains the key k that we wish to delete, and if it does not have empty subtrees,
 7474 then exchange k with its successor k' and delete k' instead. So for the following description

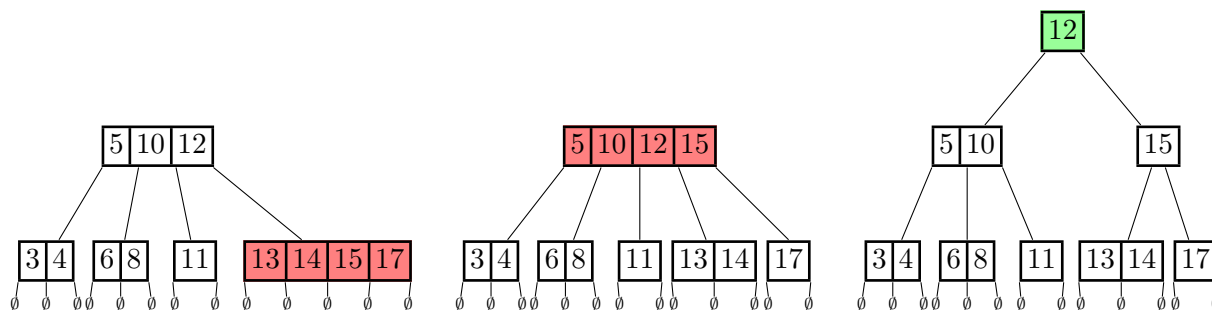


Figure 11.11: Insertion in a 2-4-tree may lead to repeated overflow; here we inserted key 17.

7475 we will assume that the node v where the structural change occurs has an empty subtree. For
 7476 a 2-4-tree, we know that nodes with empty subtrees are all on the bottommost level, so the
 7477 structural change happens at a node v that is a leaf. We delete one key and one empty subtree
 7478 from leaf v . See Figure 11.12.

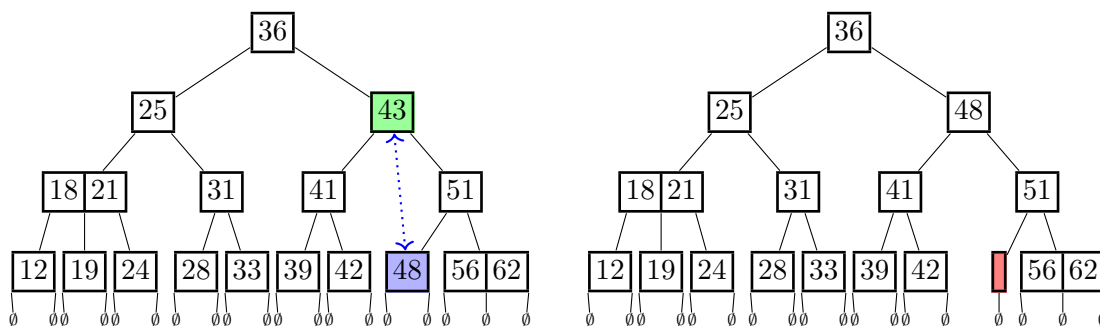


Figure 11.12: Deleting key 43 leads to underflow.

7479 Clearly the order property hold, but now we may have to restore the structural property.
 7480 Recall that nodes must have between 1 and 3 keys, so if node v had only one key previously, then
 7481 it now has no key left and only one (empty) subtree and so violates the structural property. We
 7482 call this *underflow* because node v has now fewer keys than is permitted in a 2-4-tree. We must
 7483 somehow restore the balance by transferring a key from elsewhere into v while maintaining the
 7484 order-property. This part is quite different from binary search trees, and has three sub-cases:

7485 **Case 1.** v has a sibling u that is immediately to the right of v (in the key-subtree-list of the
 7486 parent p of v). Furthermore, u stores two or more keys. (This case applies in Figure 11.12.)
 7487 In this case, we can transfer one key from u to parent p , transfer one key from parent
 7488 p to v and re-arrange the subtrees such that the order-property continues to hold. See
 7489 Figure 11.13 for an abstract depiction of the operation, and Figure 11.14 for the result of
 7490 applying it to the tree from Figure 11.12.

7491 We call this operation a *transfer* or also a *rotation*. (It strongly resembles a single-left

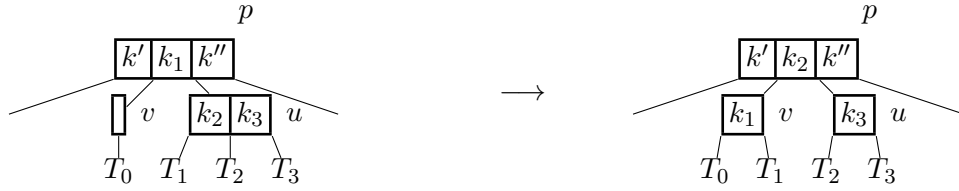


Figure 11.13: A rotation to resolve underflow.

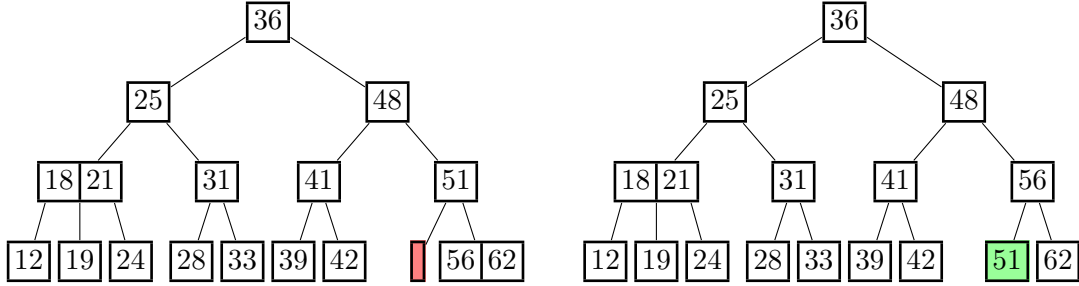


Figure 11.14: Deletion in a 2-4-tree with underflow solved by rotation. We omit the empty subtrees.

rotation: The key from u moves up while the key from p moves down and left.) Notice that after such a rotation, the parent-node p has the same number of keys as before, and both v and u have at least one key. As such, the rotation restores all required properties of a 2-4-tree and we are done with repairing the underflow.

Case 2. v has a sibling u that is immediately to the left of v (in the key-subtree-list of the parent p of v). Furthermore, u stores two or more keys.

This is handled symmetrically by moving one key from u to p , one key from p to v , and re-arranging the subtrees. Again this repairs the underflow.

Case 3. Neither of the above two cases holds. Unfortunately we then cannot repair the underflow locally. Instead, we combine v with one of its immediate siblings. We know that v has an immediate sibling u since the parent has at least two subtrees by the structural properties. We also know that u has only one key, otherwise an earlier case would apply. So the combined node v' would have one key and hence be small enough.

However, v has one subtree and u has two subtrees, so v' would have one key and three subtrees, which is not allowed. On the other hand, the parent p of v loses a subtree when combining v and u . To satisfy the structural property, we therefore add one key from the parent p to the combined node v' . See Figure 11.15. Effectively this is the reverse operation of a node split; we call this a *node merge*.

A node merge removes one key from the parent p , so might in turn create underflow in parent p . So we may need to recurse in the parent. In the extreme case this may repeat all the way until we have underflow at the root; if so then we simply delete the root and

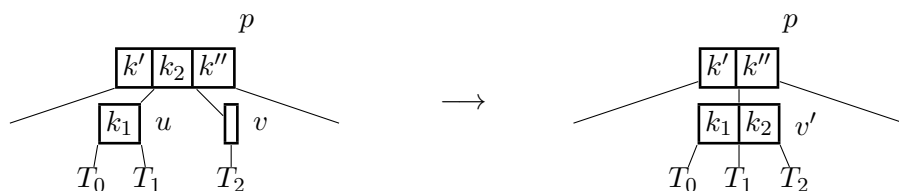


Figure 11.15: A node merge to attempt to resolve underflow.

the height of the 2-4-tree decreases. Figure 11.16 gives an example.

All three cases are combined Algorithm 11.5, which gives the pseudo-code for deletion in a 2-4-tree.

Algorithm 11.5: *24Tree::delete(k)*

```

1  $v \leftarrow 24Tree::search(k)$                                 // node containing  $k$ 
2 if  $v$  is not a leaf then
3    $v \leftarrow$  leaf that contains successor  $k'$  of  $k$ , and exchange  $k$  and  $k'$ 
4 delete  $k$  and one empty subtree in  $v$ 
5 while  $v$  has 0 keys do                                     // underflow
6   if  $v$  is the root then delete  $v$ , make its child the new root and break
7   if  $v$  has an immediate sibling  $u$  with 2 or more keys then
8     rotate a key and subtree from  $u$  to  $v$  as in Figure 11.13 and break
9   else
10    merge  $v$  with an immediate sibling as in Figure 11.15
11     $v \leftarrow$  parent of  $v$  // and repeat the while-loop

```

Summarizing 2-4-trees, they are a realization of ADT Dictionary where all operations have run-time $O(\log n)$ in the RAM model. This is competitive with AVL-trees as far as worst-case run-times are concerned. Experiments have shown that in fact they are faster than AVL-trees in practice, especially if one re-interprets them as *binary* search trees as discussed in the next section.

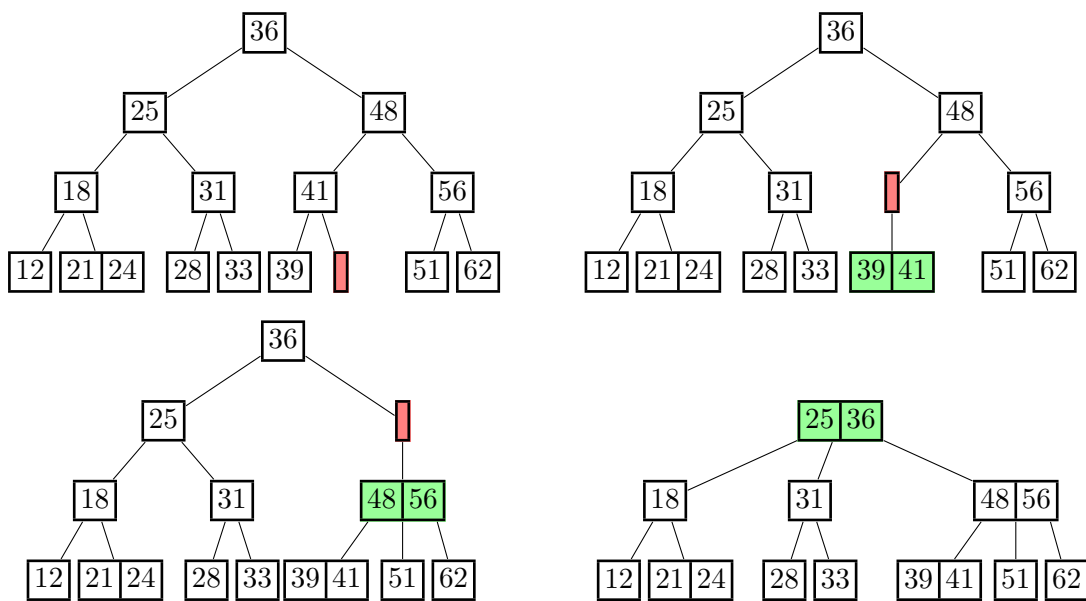


Figure 11.16: Underflow resolved by (repeated) merging, leading to a decrease in height.

11.3.3 Red-black trees (cs240e)

In 2-4-trees, every node uses constant space, but in practice there is quite a bit of space-waste. We must keep enough space in each node so that it can hold three keys and four subtrees, because during later insertions the node may gain more keys (and we probably would not want to copy the entire node over to a bigger node to do this). In order to make 2-4-trees fast in practice, the better idea is to convert the 2-4-tree into a regular binary search tree. This binary search tree comes with a special color-coding: every node is either red or black. (Note that we need only one bit to store this information.) We call this type of binary search tree a *red-black-tree*.

Recall that a *d*-node in a 2-4-tree is a node that stores *d* key-value pairs (and $d \in \{1, 2, 3\}$). We convert a 2-4-tree into a red-black tree as follows:

- A *d*-node *v* becomes a black node with *d*−1 red children.
- The black node receives the upper median of the keys at *v*. In consequence, the black node and its red children form a binary search tree of height at most 1.
- The *d*+1 subtrees of *v* are distributed among the subtrees of the black and red nodes in the only way feasible to maintain a binary search tree.

See Figure 11.17 for an example.

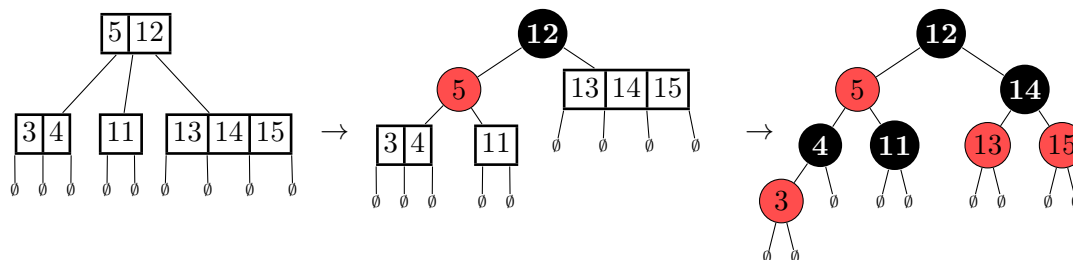


Figure 11.17: Converting a 2-4-tree into a red-black tree.

Recall that a 2-4-tree has all empty subtrees on the same level. Directly from this, and from the construction, one verifies the following:

Observation 11.1. *A red-black tree T has the following properties:*

- *The root is black.*
- *Any red node has a black parent.*
- *Any empty subtree has the same black-depth, i.e., the same number of black nodes on the path from the empty subtree to the root of T .*

It turns out that this *exactly* characterizes the binary search trees that could be obtained from a 2-4-tree. To see this, observe that if we have a binary search tree T with a red/black node-coloring that satisfies the properties of Observation 11.1, then we can convert it into a 2-4-tree T' as follows (see also Figure 11.18):

- Every black node v of T becomes a node v' of T' .

- If c_1, \dots, c_d are the children of v in T (for $d \in \{0, 1, 2\}$), then node v' also includes the keys of all children of v that are red.
- The subtrees of v' are the subtrees of v where there is no red child, as well as the subtrees of the red children among c_1, \dots, c_d .

We leave it as an exercise to verify that this indeed gives a 2-4-tree, and only note that the condition on the black-depth of T is crucial to show that all empty subtrees in T' are on the same level.

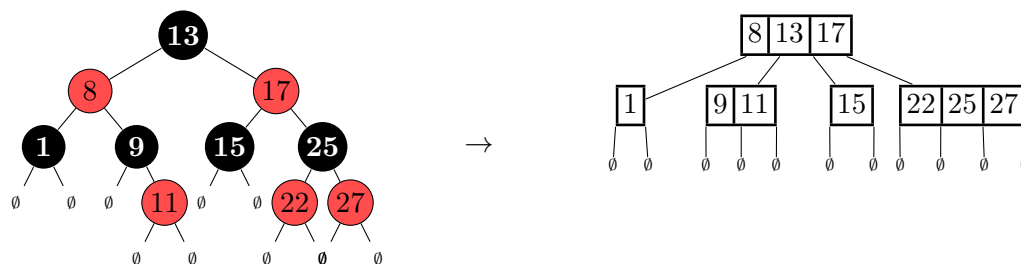


Figure 11.18: Converting a red-black tree to a 2-4-tree.

Because the class of 2-4-tree equals the class of binary search trees that satisfies Observation 11.1, one might as well have used this observation as the *definition* of a red-black tree (and this is how it is commonly done in the literature). This is yet another class of balanced binary search trees. In particular, the height is logarithmic because 2-4-trees have height at most $\log_2(\frac{n+1}{2})$ by Lemma 11.1, and the height of red-black-trees is approximately twice that.

Search in a red-black tree is straightforward as in all binary search trees. Insertion and deletion is obviously feasible in $O(\log n)$ time, because it was feasible in $O(\log n)$ time in the corresponding 2-4-tree, and the changes can be ‘translated’ to the red-black tree. One can in fact describe the algorithm for insertion and deletion directly for red-black-trees (via suitable rotations involving siblings and cousins); see Algorithm 11.6 for *insert* (the one for *delete* is left as an exercise).²

Summarizing, red-black trees are another class of balanced binary search trees that have worst-case time $O(\log n)$ in the RAM model for all operations. Contrasting them to AVL-trees, they tend to have bigger height, but use fewer rotations during insertion and deletion. So they are a bit slower for *search*, but faster for updates. They also use less space, since they only need one extra bit per node to store the color.


11.3.4 *a-b-trees*

The 2-4-trees that we saw do not improve the number of block transfers substantially since their height could still be $\log_2 n$ (this happens if all nodes are 1-nodes). To reduce the number of

²Since the insertion-routine for 2-4-trees is much easier to memorize, for assignments and exams it may be easier to convert relevant parts of the red-black tree to a 2-4-tree, do the insertion there, and then convert back.

Algorithm 11.6: *RBTree::insert(k)*

```

1  $z \leftarrow \text{BST}::\text{insert}(k)$ 
2  $z.\text{color} \leftarrow \text{red}$ 
3 while  $z$  has a parent  $p$  and  $p.\text{color} = \text{red}$  do                                // inv:  $z$  is red
    // Extract all nodes that would be in the same 2-4-tree node  $v$ 
4      $g \leftarrow p.\text{parent}$                                                     //  $g$  exists and must be black
5      $c \leftarrow$  child of  $g$  that is not  $p$ 
6     if  $c = \text{NIL}$  or  $c.\text{color} = \text{black}$  then
        // Node  $v$  had only  $g, p, z$ . Re-arrange them via rotations to be 
7          $z \leftarrow \text{restructure}(z, p, g)$ 
8          $z.\text{color} \leftarrow \text{black}; z.\text{left}.\text{color} \leftarrow \text{red}; z.\text{right}.\text{color} \leftarrow \text{red}$ 
9         break
10    else
        // Node  $v$  has  $g, p, c, z$ . Overflow  $\Rightarrow$  node-split.
11         $p.\text{color} \leftarrow \text{black}; c.\text{color} \leftarrow \text{black}; g.\text{color} \leftarrow \text{red}$ 
12         $z \leftarrow g$ 
13 if  $z.\text{color} = \text{red}$  then  $z.\text{color} \leftarrow \text{black}$     // root is red  $\Rightarrow$  increase black-depth

```

7575 block transfers, we need to store more keys at each node. The resulting realization of ADT
 7576 Dictionary is called an *a-b-tree*. Figure 11.19 shows a 3-6-tree.

7577 **Definition 11.2.** Let $a \geq 2$ and b be integers with $a \lceil b/2 \rceil = \lfloor (b+1)/2 \rfloor$. An *a-b-tree* is a
 7578 multi-way tree that satisfies the following structural properties:

- 7579 • The multi-way tree has order b , i.e., any node has at most b (possibly empty) subtrees.
- 7580 • Any node that has d subtrees stores exactly $d - 1$ key-value pairs.
- 7581 • Any node that is not the root has at least a subtrees (hence stores at least $a - 1$ keys).
- 7582 • The root has at least 2 subtrees, hence stores at least one key.
- 7583 • All empty subtrees are at the same level.

7584 It also satisfies the same order-property as a 2-4-tree: At any node the keys and subtrees can be
 7585 ordered as $\langle T_0, k_1, T_1, \dots, k_d, T_d \rangle$ such that all keys in T_i fall into the range between k_i and k_{i+1}
 7586 (assuming $k_0 := -\infty$ and $k_{d+1} := \infty$).

7587 Put differently, an *a-b-tree* is very much like a 2-4-tree, except that the range of the number
 7588 of subtrees has changed from “between 2 and 4” to “between a and b ”. There is, however, one
 7589 exception at the root, which is allowed to have fewer than a subtrees. (We will see below why
 7590 this is necessary.)

7591 **Height:** Recall that we already showed that a 2-4-tree has height $\log_2(\frac{n+1}{2})$. In a similar fashion
 7592 we now analyze the height of an *a-b-tree*. The rough idea is that on every level we have a times

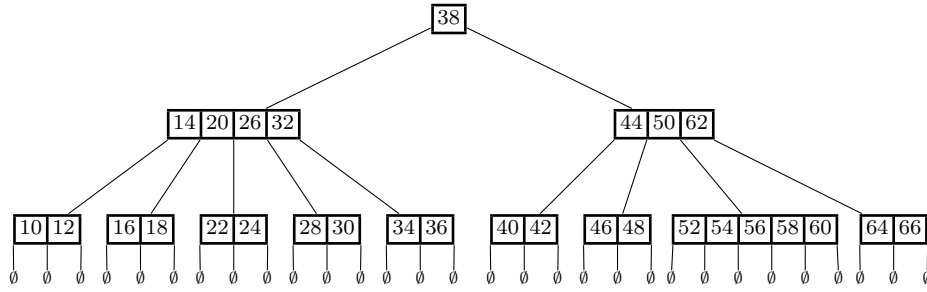


Figure 11.19: A 3-6-tree.

as many nodes as on the level above, because every node is required to have a subtrees, and those subtrees are not allowed to be empty except on the bottommost level. Therefore we have roughly a^i nodes on level i , and since each node has keys we should have $h \approx \log_a n$. This rough idea is a bit too simple, because we have the exception-rule at the root. On the other hand, each node actually stores at least $a - 1 \geq 1$ keys, which works in our favor. Note that the following bound exactly matches the bound from Lemma 11.1.

Lemma 11.2. *Any a - b -tree has height at most $\log_a \left(\frac{n+1}{2} \right) \in O(\log_a n)$.*

We will give two proofs of this, using slightly different approaches.

Proof. We first determine some lower bounds on the number of keys and subtree-links that must exist on each level, see also Table 11.20:

- On level 0, we have only one node (the root). This stores at least one key-value-pair and has at least two links to subtrees. In total we hence have on level 0 at least one key-value pair and at least two links to subtrees.
- On level 1, we have one node for every link that came from the previous level. Thus we have at least two nodes. Each of those stores at least $(a - 1)$ key-value pairs and has at least a links to subtrees. In total we hence have on level 1 at least $2(a - 1)$ key-value pairs and at least $2a$ links to subtrees.
- On level 2, we have one node for every link that came from the previous level. Thus we have at least $2a$ nodes. Each of those stores at least $(a - 1)$ key-value pairs and has at least a links to subtrees. In total we hence have on level 2 at least $2a(a - 1)$ key-value pairs and at least $2a^2$ links to subtrees.
- Iterating, on level i , we have one node for every link that came from the previous level. Thus we have at least $2a^{i-1}$ nodes. Each of those stores at least $(a - 1)$ key-value pairs and has at least a links to subtrees. In total we hence have on level i at least $2a^{i-1}(a - 1)$ key-value pairs and $2a^i$ links to subtrees.

Putting it all together, on levels $0, \dots, h$ (where h is the height of the a - b -tree) the total

Level	Nodes	key-value pairs	links
0	1	1	2
1	2	$2(a-1)$	$2a$
2	$2a$	$2a(a-1)$	$2a^2$
3	$2a^2$	$2a^2(a-1)$	$2a^3$
\vdots	\vdots	\vdots	\vdots
h	$2a^{h-1}$	$2a^{h-1}(a-1)$	$2a^h$

Table 11.20: Lower bounds on the number of key-value pairs on each level.

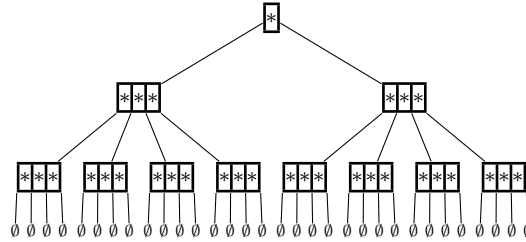


Figure 11.21: The 4-8-tree of height 2 with the smallest possible number of keys.

number of key-value pairs is

$$n \geq 1 + \sum_{i=1}^h 2a^{i-1}(a-1) = 1 + 2(a-1) \sum_{i=0}^{h-1} a^i = 1 + 2(a-1) \frac{a^h - 1}{a - 1} = 1 + 2(a^h - 1) = 2a^h - 1$$

7618 which shows that $h \leq \log_a(\frac{n+1}{2}) \in O(\log_a n)$ as desired. \square

7619 Now we give a second proof that uses a detour that could be interesting in its own right.

7620 *Proof.* We claim first that level i contains at least $2a^{i-1}$ roots of subtrees for $i \geq 1$. This is
 7621 easily shown by induction—on level 1, we have at least 2 subtrees, and with each later level
 7622 there must be a subtrees for every node on the previous level. Therefore on level $h+1$ (the level
 7623 with the empty subtrees) we have $2a^h$ subtrees.

Now we claim that any a - b -tree with m empty subtrees contains exactly $m-1$ key-value pairs. This is again easily shown by induction. If the root contains k key-value pairs, then it has $k+1$ subtrees T_0, \dots, T_k . Let n_i and m_i be the number of key-value pairs and empty subtrees in T_i ; by induction $n_i = m_i - 1$. Every empty subtree of T belongs to exactly one of T_0, \dots, T_k , so $m = \sum_{i=0}^k m_i$. Adding in the k key-value pairs at the root, hence

$$n = k + \sum_{i=0}^k n_i = k + \sum_{i=0}^k (m_i - 1) = k - (k+1) + \sum_{i=0}^k m_i = m - 1.$$

Combining the two results, any a - b -tree has at least $2a^h$ empty subtrees, so at least $2a^h - 1$ key-value pairs, so $n \geq 2a^h - 1$ which gives the result. \square

Operations: The operations in an a - b -tree are almost verbatim the ones of the 2-4-tree, with the following changes:

- Algorithm *search* is exactly the same, except more care must now be taken in how we find the appropriate subtree for recursion. We may have up to $b - 1$ keys stored that we must search in. Since we have the keys in sorted order, we assume that they are stored in a way that enables us to search within one node in $O(\log b)$ time. (For example we could store the keys inside one node as an AVL-tree.)
- For *insert*, having *overflow* means having one more key and subtree than is allowed. For an a - b -tree it thus means having b keys and $b+1$ subtrees. If overflow happens, then as before we resolve it by node-splitting. This creates two nodes with $\lceil (b+1)/2 \rceil$ and $\lfloor (b+1)/2 \rfloor$ subtrees, respectively. Recall that we restricted $a \leq \lfloor (b+1)/2 \rfloor$, therefore the newly created nodes have sufficiently many subtrees for an a - b -tree. See Figure 11.22 for an example.

Also, if overflow happens at the root, then we split the root and create a new root with one propagated key. Notice that this new root has one key and two children. This is the reason for the exemption that permits the root to have fewer subtrees than other nodes.

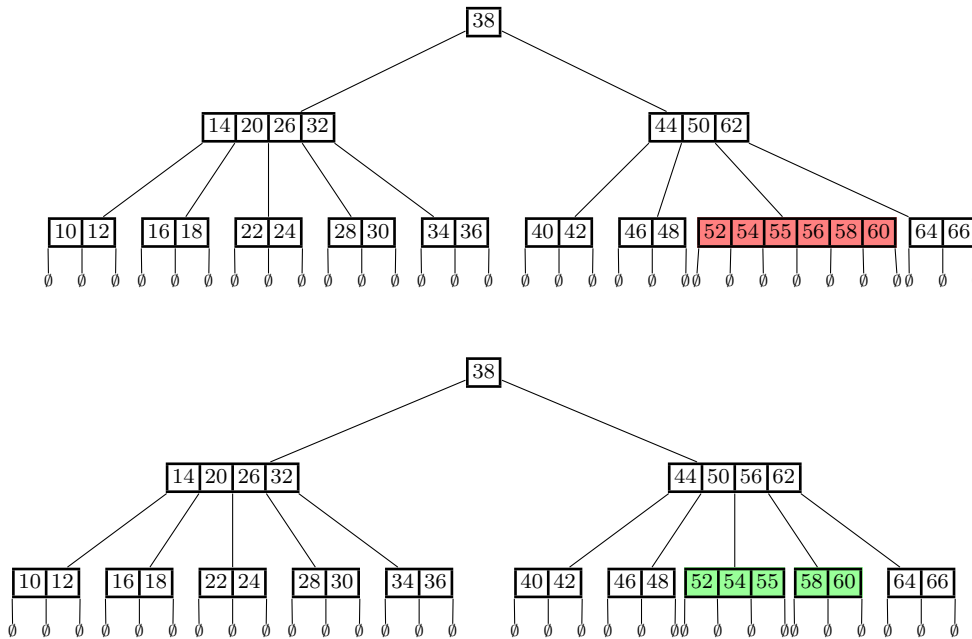


Figure 11.22: Inserting key 60 into the 3-6-tree leads to overflow, which is resolved by node-splitting.

- For *delete*, *underflow* means having one fewer key and subtree than is allowed. Thus in an a - b -tree, underflow at the root means having no key and one subtree. Such underflow is resolved as before by removing the root and making its unique child the new root.
- For a non-root v , underflow means that the node has $a - 2$ keys and $a - 1$ subtrees. We attempt to resolve it by transferring a key from an immediate sibling. This is possible whenever one immediate sibling has more than a subtrees and resolves the underflow.
- If all immediate siblings have exactly a subtrees, then we use node-merging, i.e., merge v with an immediate sibling u . We know that v has $a - 1$ subtrees and u has a subtrees, so the combined node v' has $2a - 1$ subtrees. Since $a \leq \lceil b/2 \rceil$ we have $b \geq 2a - 1$ so v' ends with sufficiently few subtrees. Exactly as for 2-4-trees we need to include one key-value pair from the parent in v' to satisfy the structural properties, and hence need to recurse at the parent.

For all three operations, $O(\text{height})$ nodes are being handled, and handling one node takes $O(\log_b n)$ time (presuming keys are stored appropriately). Since the height is in $O(\log_a n)$, the run-time is hence proportional to

$$\log b \cdot \log_a(n) = \log n \cdot \frac{\log b}{\log a}.$$

To minimize this run-time, we should maximize a for a given b . But we are also requiring that $a \leq \lceil b/2 \rceil$, so we should use $a = \lceil b/2 \rceil$. With this, $\log a \geq \log b - 1$ and so the run-time for all operations is $O(\log n)$.

11.3.5 B -trees

In the discussion of a - b -trees of the previous section, we analyzed the run-time in the RAM model. Here they give another method of guaranteeing worst-case run-time $O(\log n)$ for all operations, but are no better than other balanced binary search trees (and in fact, may be worse depending on how exactly we store keys and links in each node).

The real reason why we studied a - b -trees is the external memory model: they can be used for a realization of ADT Dictionary that uses very few block transfers. This is called a B -tree (where the ‘ B ’ reminds of the block size). Figure 11.23 shows an example where the order (the maximum permitted number of subtrees at a node) is 4.

Definition 11.3. A B -tree is an a - b -tree with the following restrictions:

- Every node of the B -tree fits into one block of memory.
- The order b of the B -tree is the maximal number that satisfies the above constraint.
- We set $a = \lceil b/2 \rceil$.

B -trees are specifically designed to work with external memory, and therefore we show in Figure 11.24 a close-up that considers internal vs. external memory. The order of the B -tree is exactly chosen such that one node fits into one block. In the example, we assume $B = 17$,

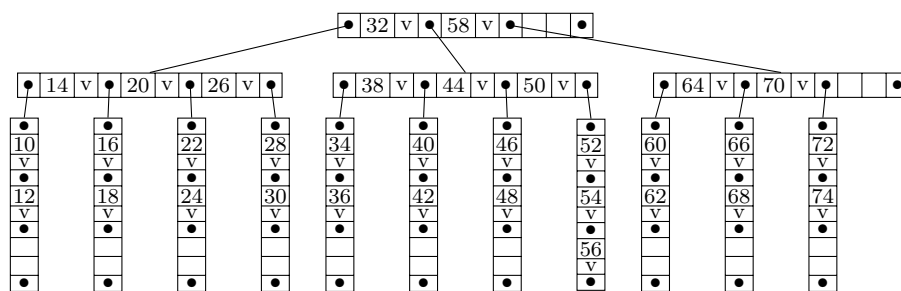


Figure 11.23: A B -tree of order 4. ‘v’ represents the value associated with the key next to it.

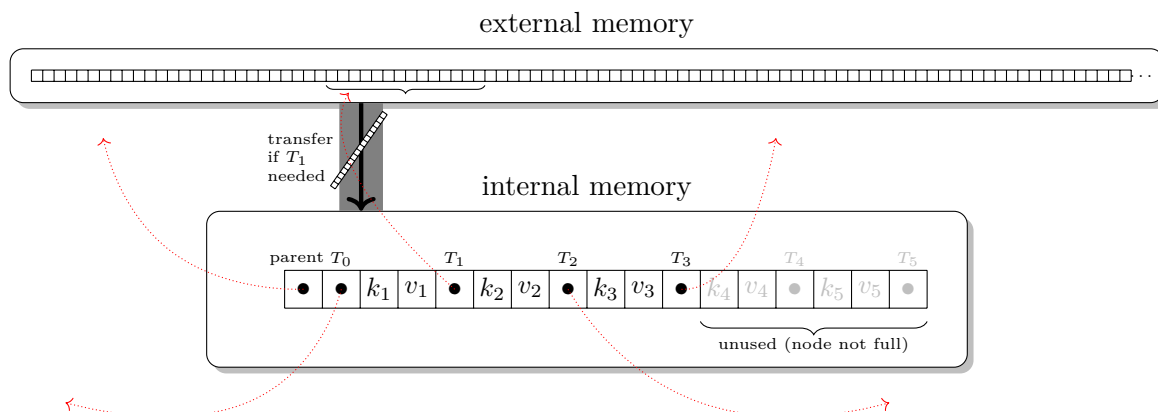


Figure 11.24: A node of a B -tree in the external memory model. If $B = 17$ then the B -tree can have order 6.

i.e., we can fit 17 computer-words into one block and can hence use order 6: Each node has a parent-reference, 6 subtrees, 5 keys and 5 values to store.

Let us see how B -trees perform in the external memory model. Since operations in internal memory are not counted, and since we chose the order such that an entire node fits into one block, we only need to worry how often we have to load a node during an operation. Each operation only affects the nodes on the search-path (and perhaps one sibling of each since we may merge and/or split nodes). Therefore the number of block transfers is proportional to the height of the B -tree.

Lemma 11.3. *The height of a B -tree is in $O(\log_B(n))$, assuming $B \geq 36$.*

Proof. We know that the height is in $O(\log_a(n))$, so we mostly need to bound a suitably. We chose the order b such that $b \in \Theta(B)$; specifically $b \approx B/3$. We chose $a \approx b/2$, therefore $a \approx B/6$. To be able to do arithmetic, let us assume that $a \geq cB$ for some constant $c \approx 1/6$. Since $B \geq 36$,

therefore $c \geq 1/\sqrt{B}$. With this, the height of the B -tree is proportional to

$$\log_a n \leq \log_{cB} n = \frac{\log n}{\log(cB)} \leq \frac{\log n}{\log \sqrt{B}} = \frac{\log n}{\frac{1}{2} \log B} = 2 \frac{\log n}{\log B} = 2 \log_B(n).$$

7682

□

7683 Summarizing B -trees, they support all operations of ADT Dictionary using $O(\log_B n)$ block
 7684 transfers. This asymptotically matches the lower bound of $\Omega(\log_B n)$ for comparison-based
 7685 search in the external memory models. B -trees are *extremely* important when it comes to
 7686 implementing data bases that are too large to fit into internal memory, and you may well
 7687 encounter them again in cs448 (Database System Implementations).

7688 11.3.6 B -tree variations (cs240e)

7689 Even though B -trees have the asymptotically optimal number of block transfers, in practice
 7690 they can still be improved. We sketch here a few ideas.

7691 **Pre-emptive splitting and merging:** In the B -trees that we described, we potentially tra-
 7692 verse the tree twice during insertion and deletion. Namely, we first go down from the root to
 7693 find the leaf ℓ where we insert/delete the key k . Then we go back up from that leaf ℓ and deal
 7694 with overflow/underflow, which potentially needs to go back up all the way to the root. Thus
 7695 the number of block transfers could be as much as twice the height of the B -tree.

7696 We can avoid going both down and up (and also save the parent-reference) with a strategy
 7697 called *pre-emptive splitting and merging*. While we search for a key (during insert/delete), we
 7698 *always* split a node if its number of keys is at the upper and lower bound.

7699 We illustrate this for insertion (of key 49) on the in the B -tree of Figure 11.23. See Fig-
 7700 ure 11.25. We search at the root and go from there to the middle child v . Node v has already
 7701 three keys, the maximum allowed. Since we are inserting in this subtree, it is possible that there
 7702 will be overflow, forcing node v to be split. Because we do not want to have to do a second pass
 7703 to do this later, we split node v already now.

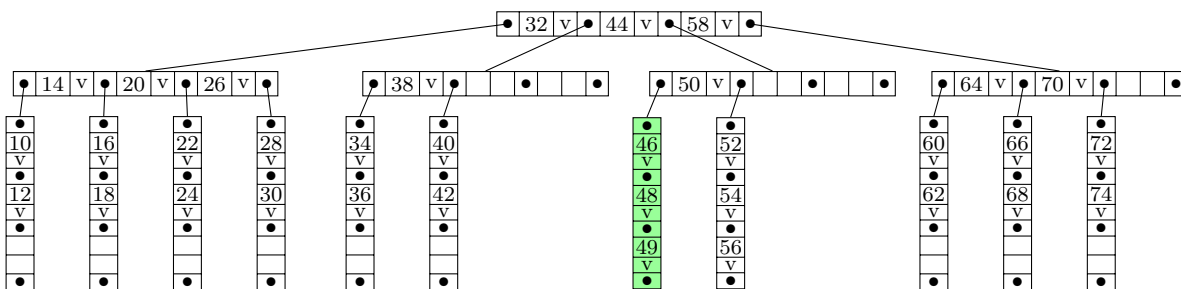


Figure 11.25: The B -tree gets changed with pre-emptive splitting when inserting 49.

Notice that in this example, we would not have needed to split this node for inserting key 49, because the leaf where 49 will get added has room to add it without overflowing. But we cannot know this yet, and hence split v regardless. This will not lead to overflow at the parent of v , because we previously searched at the parent and ensured that it has at least one entry free (else we would have split it as well).

Also note that for pre-emptive splitting to give a valid a - b -tree, we now must allow $a = \lfloor b/2 \rfloor$, because we split the b subtrees at node v between the two new nodes, one of which will have only $\lfloor b/2 \rfloor$ subtrees. For even b , this makes no difference.

B^+ -trees: We have throughout this course not thought much about the values that are associated with the keys, since we reserve $\Theta(1)$ space for the key anyway, and so the value is just overhead that disappears in the asymptotic notation. However, for B -trees it makes a difference whether we store the values with the keys—the values take up space on one block, which affects the choice of order b , hence the height and hence the number of block transfers. We therefore propose here a variant of B -trees that stores values elsewhere, therefore can fit more keys into one block and performs better.

We first need a small detour. Some of the data structures that we have seen earlier were trees where internal nodes only store information to guide the search, while all actual entries are stored at the leaves. Examples of this include tries and kd-trees. In general, any tree-structure that is defined by order relative to the root can be converted to such a *decision-tree variant*; see Figure 11.26 for a binary search tree and a decision-tree based variant. (This is exactly the same as the 1-dimensional kd-trees that we saw with priority search trees.)

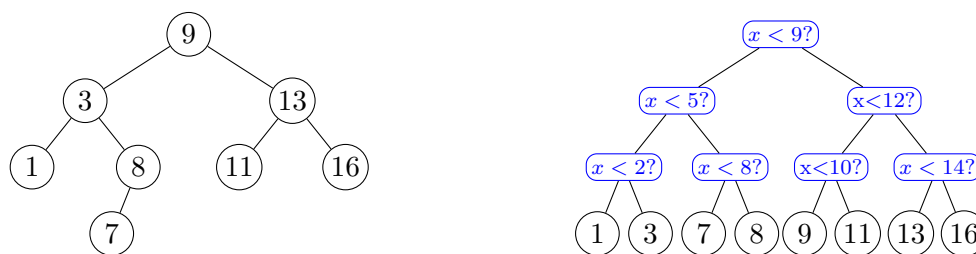
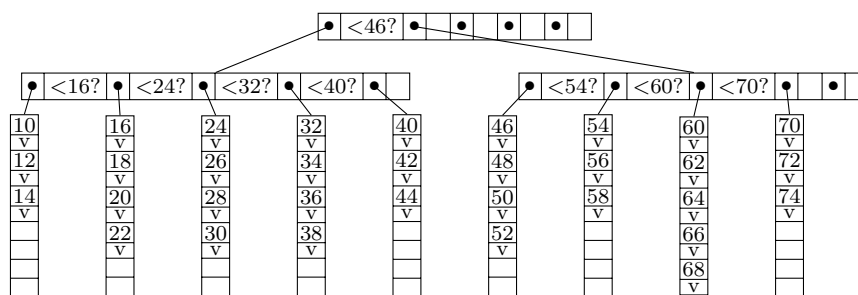


Figure 11.26: A binary search tree, and a decision-tree variant with the same keys.

In general, using a decision-tree based variant may seem like a waste of space (it typically uses $\approx 2n$ nodes for n items). But as we will see, they have advantages in the external memory model.

We now study the decision-tree variant of a B -tree, so internal nodes store *only* keys to guide the search, and the actual key-value pairs are at the leaves. Since leaves are then a different type of node anyway, we can also improve them and *not* store links to subtrees (all those subtrees are empty anyway, and will never be replaced by non-empty subtree since an a - b -tree grows from the root upward). See Figure 11.27 for an example.

Figure 11.27: A B^+ -tree.

To see why using B^+ -trees is advantageous, let us study an example. Assume (as in Figure 11.23 and 11.27) $B = 10$, so a block has space for 10 entries that can be references, keys, or values. How many keys can we with height 2? For the B -tree, each node can store 3 key-value pairs, hence have 4 subtrees. (We will generously not even consider that it might also need to store a parent-reference.) So with three layers the maximum number of nodes is $1 + 4 + 16 = 21$, and since each node stores up to three keys, we can store up to 63 key-value pairs. In contrast, in a B^+ -tree each internal node can have up to five sub-trees. So with three layers the maximum number of leaves is 25. Also, each of these leaves can store up to 5 key-value pairs since it does not store sub-tree references. So with the same height we can store up to 125 key-value pairs, or almost twice as much as with the B -tree. In other words, the height of a B^+ -tree is typically noticeably smaller than the height of a B -tree, resulting in faster searches.

LSM-trees: B -trees (and similarly B^+ -trees) achieve the optimum number of block-transfers for searches, and generally perform well when searching. But they are not ideal for updates, since quite a few block-transfers may be used. Therefore extensions have been designed that handle insertions better.

Observe that B -trees never need more than $O(1)$ blocks in internal memory. But typically $M \gg B$, so we have space in internal memory to store more information. This has been exploited in so-called *log-structured merge trees*.³ The idea is to keep a dictionary C_0 in internal memory, as well as a B^+ -tree C_1 stored in external memory. Dictionary C_0 keeps a log of all insertions or requests to change a value. Whenever we want to perform a search, we first search within C_0 (which should be fast, since it is in internal memory) and only if we are unsuccessful do we start a search in C_1 . This needs no more page loads as if we had stored everything in the B^+ -tree.

The main difference happens during *insert* (or when requesting to change a value, which is handled similarly). Here the insertion happens *only* in C_0 if there is space left in internal memory. Only if the internal memory is full do we do page loads to write relevant information to C_1 . When transferring an item from C_0 to C_1 , we do need to do some block-transfers and possibly do node-splits, but during the same time we can transfer *all* items from C_0 that fit into the loaded blocks. While the worst-case bound is no better, in practice we will typically transfer quite a few items and hence the average number of block transfers per item is noticeably smaller.

We only mention briefly here without details that the idea can be developed even further. In particular we could move parts of C_1 into (exponentially bigger) dictionaries C_2, C_3, \dots , all in external memory, and move items from C_i into C_{i+1} only if C_i reaches some defined size-limit. There are also different methods of how to do the update, i.e., what all to move from C_i to C_{i+1} , and how to make the lookups within the dictionaries more efficient (e.g. use hashing).

11.4 Extendible hashing

One of the most popular methods of realizing ADT Dictionary is hashing, due to its simplicity and since it can use an array without any overhead for references. We have seen that hashing operations have $O(1)$ expected run-time as long as we keep the load-factor small enough and the hash-function is well-chosen (preferably randomly from a universal hash function). In consequence any hashing operation uses an expected number of $O(1)$ block transfers, since at the worst each operation executes one block transfer.

As such, on first sight it looks as if hashing should perform well in the external memory model. However, there is a major problem. In order to have $O(1)$ expected run-time, we need to keep the load factor small, which means that we need to re-hash whenever the load factor gets too big. Recall that re-hashing means moving *all* items in the dictionary to a new location. This takes $\Theta(n/B)$ block-transfers, which means that the occasional insert-operation takes nearly forever. This is unacceptable. We are therefore now developing a data structure

³The ‘log’ here refers to ‘keeping a log of the updates’, and *not* to ‘logarithm’.

- To *insert* a key-value pair with key k , we search for k , find that it should be at leaf ℓ , and load the appropriate block P . If block P has space, then we simply add the key-value pair to it.
- In the (rare) situation that the block does not have space, we need to split P into two blocks P_0 and P_1 . To do so, let d be the depth of leaf ℓ (we have hence used the leading d bits k to get to leaf ℓ). Expand the trie by one more level, and split the keys in P by the next bit (i.e., the $(d+1)^{st}$ bit) to get P_0 and P_1 .
- In the (extremely rare) situation that *all* keys of P have the same $(d+1)^{st}$ bit, we need to repeat the splitting, possibly multiple times, until the keys have been split into multiple block. Then we add k . See Algorithm 11.7 for the code and Figure 11.29 for an example.

Algorithm 11.7: *trieOfBlocks::insert(k, v)*

Input : k is a bit-string (possibly from a hash-value of the original key)

```

1  $\ell \leftarrow \text{trie}::\text{searchUntilLeaf}(D, k)$            // use search, but modify to stop at leaf
2  $d \leftarrow$  depth of  $\ell$  in  $D$ 
3 load block  $P$  that  $\ell$  refers to
4 while  $P$  has no room for additional items do           // expand trie
5   Split  $P$  into two blocks  $P_0$  and  $P_1$  by bits at index  $d$ 
6   Create two children  $\ell_0$  and  $\ell_1$  of  $\ell$ , linked to  $P_0$  and  $P_1$ 
7    $\ell \leftarrow \ell_{k[d]}$ ,  $P \leftarrow P_{k[d]}$ ,  $d \leftarrow d+1$ 
8 add  $(k, v)$  to  $P$ 

```

So in most situations we need only two block transfers (to load P and to write it back when we have added k). If we need to split P once, then we need three block transfers (we write two blocks back). If we need to split P multiple times (say $x > 1$ times) then we need $x + 2$ block transfers. But how likely is this? We need to split twice only if the b keys in P all have the same $(d+1)^{st}$ bit. Assuming that all bit-strings are equally likely to be keys, the probability of this is $(\frac{1}{2})^b$, a *very* small number. (Recall that $b \in \Theta(B)$.) The probability that we need to do it $x \geq 2$ times is $(\frac{1}{2})^{b(x-1)}$, an *even* smaller number. So the expected number of block transfers is very close to 2.

- To *delete* a key k , we search for k , find it in leaf ℓ , load the block at ℓ , delete k , and write the block back to external memory. This takes two block transfers.

We could now spend some time on “cleaning up” (i.e., re-combine blocks if they are small enough to fit into one block), but since external memory is cheap while cleaning up would require some block transfers and hence be slow, we usually do not do that and instead live with the slight waste of space.

A trie of blocks wastes some space, since many blocks have extra space available, and during *insert* we may even be creating blocks that have no keys in them at all (though this should be very rare). Since external memory is cheap, wasting space is a relatively minor concern. It has also been shown in experiments that on bit-strings that are generated uniformly at random, not

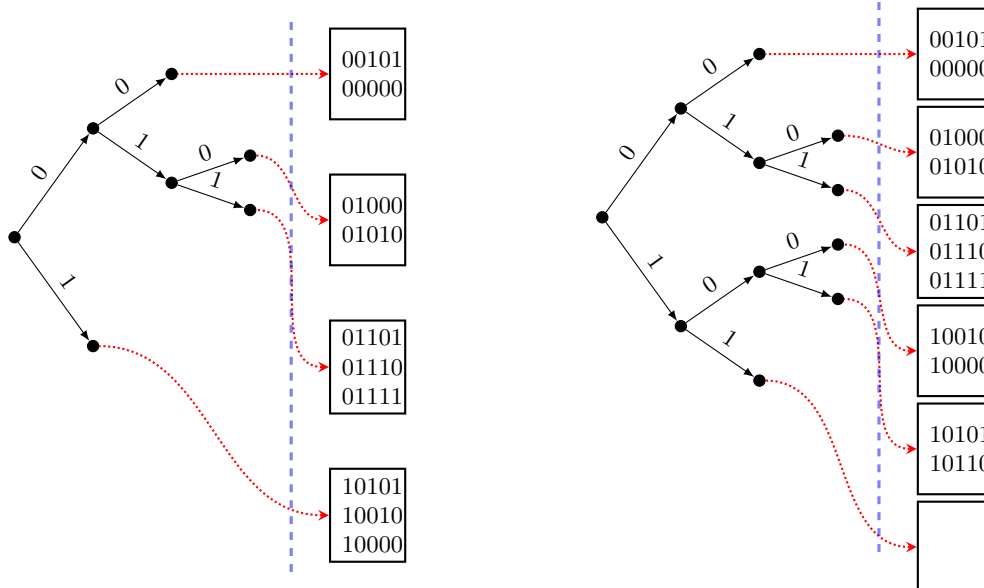


Figure 11.29: Inserting 10110 into a trie of blocks.

much space is wasted and the blocks are about 69% full on average.

Summarizing, for most operations we use only one or two block transfers, and even when we need more (during *insert*) it should be a small constant number of block transfers. We *never* need to load all items of the dictionary into internal memory (in contrast to re-hashing).

11.4.2 A few discussion items

We have ignored a few issues. First, we have assumed that the block-dictionary D fits into internal memory. This is a fairly reasonable assumption. If the bit-strings are randomly distributed, then D should be quite balanced, and so should have roughly as many leaves as it has internal nodes. So if D has size $\Theta(M)$ (where M is the internal memory size) then it should have $\Theta(M)$ leaves and hence store $\Theta(M \cdot B)$ many key-value pairs. Put differently, for $n \in \Theta(M \cdot B)$ (a large number) we would expect D to fit into internal memory. But if it does not fit into internal memory, then we can apply similar techniques as we used for B -trees. Put differently, split D into sub-tries of maximal height that fit into one block, and load the parts of D that are needed during a search. One can argue (details are left as an exercise) that the expected number of block transfers for randomly generated bit-strings is then $O(\log_B(\frac{n}{B}))$.

We have also assumed during *insert* that after sufficiently many splits the keys in block P are no longer all in one block. This holds if all keys are different, but recall that our motivation was hashing (and what we call “keys” are really hash-values of keys). During hashing, we have collisions, and so some of our keys may be identical. It could in particular happen that there

are $b+1$ identical keys (recall that a block can hold b key-value pairs). In this case, *insert* would fail.

In hashing in the RAM-model, a b -fold collision of hash-values would tell us that the hash function is bad (or the load-factor too big) and so we should re-hash. We do not want to re-hash in the external memory model, but we do have the option to change the hash-function so that not all these keys fall into one block. Specifically, assume that we previously had hash-function $h(\cdot)$, and there were $b+1$ keys that all hashed to the same value. Define a second hash-function $h'(\cdot)$, and use as new bit-string for a key k the bit-string $h(k)++h'(k)$ (i.e., concatenate). This has the same leading bits as the old bit-string, so any key k belongs to the same leaf of the trie of blocks as previously. But since we are free to choose the hash-function h' , and we can choose it to act very different from h , this *hash-function expansion* should differentiate the keys that previously all had the same hash-value, and so insertion should succeed.

11.4.3 Avoiding the trie

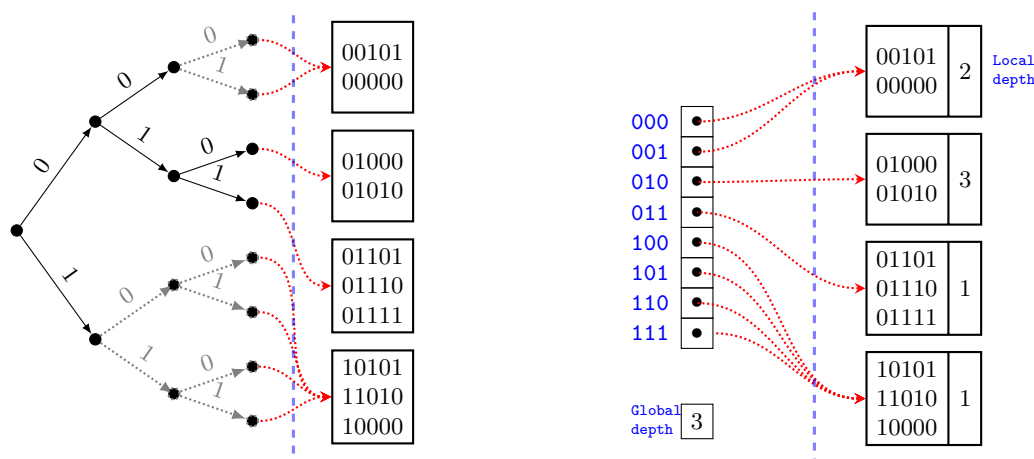
We explained extendible hashing with the help of the trie of blocks because this makes understanding the operations easy. However, with a different approach we can save space for the block-dictionary D by using an array, rather than a trie with links. (Recall that we want block-dictionary D to fit into internal memory, so saving space here is worth it.)

Two notations will be helpful. Given a trie of blocks D , let d_D (the *global depth*) be the height of the trie. Also, for any leaf ℓ of the trie, let the *local depth* d_ℓ be the depth of this leaf; clearly $d_\ell \leq d_D$. We use the term *local depth* also for the block that is referred to by leaf ℓ , and we assume that every block stores its local depth.

Now we replace the trie by an array A of length 2^{d_D} . This means that the indices of A (in $\{0, \dots, 2^{d_D} - 1\}$) correspond to bit-strings of length d_D , and we will show them as bit-strings. Array $A[w]$ (for some bit-string w) points to the same block as the leaf ℓ in the trie of blocks that we would have reached when parsing w . A different way to say the same thing is to think of the trie as expanded until all leaves have depth d , and then letting A refer to the same blocks as the leaves. See Figure 11.30.

Since blocks store the local depth, it is quite easy to create the pseudo-code for the various operations, without needing to build a trie, see Algorithm 11.8 for *insert*. The only major change occurs if we want to split a block that has the maximum possible depth (i.e., $d_\ell = d_D$). This corresponds to a split in the trie that increases the height of the trie, hence an increase in the global depth d_D . To reflect this accurately, we need to double the size of array A , i.e., create a new array A' of size 2^{d+1} and copy references from A to A' . See Figure 11.31 for an example. This may feel much like re-hashing, but there are two crucial differences. First, A and A' (usually) fit into internal memory, so copying them is fast. Second, every entry of A' is a link that was previously a link in A . So we do *not* need to load any block to create A' , and *insert* in extendible hashing uses exactly as many block-transfers as it used for the trie with blocks.

Summarizing, if we replace the trie of blocks D by an array A that only stores the references to the blocks, then the space used for the block-dictionary in internal memory is smaller (hence

Figure 11.30: Converting a trie of blocks to an array of size 2^{dD} .

7885 a larger block-dictionary will fit before we have to resort to B -tree like structures), without
 7886 sacrificing performance with respect to the number of block transfers.

7887 11.5 Take-home messages

- 7888 • If a very large amount of memory is used, then the assumption that all memory-accesses
 7889 take the same time is no longer realistic and a different memory model should be used.
- 7890 • Algorithms that use streams, rather than explicit arrays or lists, tend to perform well even
 7891 when using external memory and even without knowing the block size.
- 7892 • There are realizations of ADT Dictionary that are tailored to the external memory model
 7893 and that perform better (with regards to the number of block transfers) than the realiza-
 7894 tions we saw earlier.

7895 11.6 Historical remarks

7896 Dealing with external memory was a problem from the earliest days of computers. Initially (in
 7897 the 1950s) the devices to provide extra (but slow-to-access) memory were tape drives, but over
 7898 the years they were replaced by disks, and then flash drives and cloud storage. The external
 7899 memory model was first formulated by Aggarwal and Vitter [AV88]; they also provide the
 7900 lower bounds on sorting and show how to match them by modifying merge-sort. Numerous
 7901 improvements (including how to create better initial runs, and how to adapt radix-sort) are
 7902 discussed extensively in Knuth's book [Knu98, Section 5.4]. We have studied here only how
 7903 to adapt *merge-sort* to external memory; one can also adapt *quick-sort* (see [Ver88] and the
 7904 references therein).

Algorithm 11.8: *extendibleHashing::insert*(k, v)

Input : k is a bit-string (possibly from a hash-value of the original key)

```

1  $d_D \leftarrow$  global depth
2  $w \leftarrow k[0..d_D-1]$ , interpreted as an integer in  $\{0, \dots, 2^{d_D} - 1\}$ 
3 load block  $P$  that  $A[w]$  refers to
4 while  $P$  has no room for additional items do                                // need new block
5      $d_\ell \leftarrow$  local depth of  $P$ 
6     if ( $d_\ell$  equals  $d_D$ ) then                                                // expand block-dictionary
7          $d_D++$ ,  $A' \leftarrow$  new array of size  $2^{d_D}$ 
8         for ( $i = 0; i < 2^{d_D}; i++$ ) do  $A'[i] \leftarrow A[\lfloor i/2 \rfloor]$            // copy over
9          $A \leftarrow A'$ 
10    Split  $P$  into two blocks  $P_0$  and  $P_1$  by bits at index  $d_\ell$ 
11    Set local depth of  $P_0$  and  $P_1$  to be  $d_\ell + 1$ 
12     $w_d \leftarrow k[0..d_\ell-1]$                                                 // update links to  $P_0$  and  $P_1$ 
13    for  $i = 0, 1$  do for all extensions of  $w_d++i$  into a length- $d_D$ -bit-string  $x$  do
14         $A[x] = P_i$ 
15     $P \leftarrow P_{k[d]}$ 
15 insert ( $k, v$ ) into  $P$ 

```

The idea that led towards B -trees (break a bigger binary search tree into subtrees that fit into a block) was considered in the 1960s [Lan63], but the updates in the proposed structures were significantly more complicated and needed many block transfers. B -trees were introduced by Bayer and McCreight in 1972 [BM72]. They quickly became *the* standard method of storing large files; a 1979 survey by Comer [Com79] lists numerous early development and improvements, including B^+ -trees. LSM-trees were introduced in 1996 by O’Neil, Cheng, Gawlick and O’Neil [OCGO96], see a 2020 survey by Luo and Carey [LC20] for further developments.

Extendible hashing appears to have been developed multiple times around the same time, by Fagin, Nievergelt, Pippenger and Strong [FNPS79] in 1979, but also by Litwin in 1980 [Lit80] with improvements by Larsen [Lar80]. All these sources operate directly using the hash-function and using an array for the global dictionary; the explanation via a block of tries should be mostly seen as a didactic tool.

The topic of external memory is vast, and we have only scratched its surface. In particular, in our model the external memory was purely for storage, and all computation is done in internal memory. While this was a reasonable model a few decades ago, these days external memory typically means cloud storage, and typically the computers in the cloud can not only store, but also manipulate the data. This raises a whole lot of other issues, in particular relating to concurrency (what if different clients access the same data around the same time?). Since we are in the age of big data, and are not only generating vast amounts of data but the rate at which we



are generating appears to be increasing, dealing with questions how to split data across systems and coordinate among them is very important and an active research area.

are generating appears to be increasing, dealing with questions how to split data across systems
and coordinate among them is very important and an active research area.

In conclusion

As the course-title suggests, this course was about how to store data and how to manage data that has been stored. Specific topics that we have seen are:

- How to re-organize data. We were interested in how to *sort* data, and studied numerous algorithms here. In particular, there are multiple algorithms that achieve $O(n \log n)$ worst-case run-time, such as *merge-sort* and *heap-sort*. In practice *quick-sort* is fastest, even though its worst-case run-time bound is worse. For special keys we can achieve better run-times using *count-sort* and *radix-sort*.

We also studied some problems where only partial sorting was needed, such as *priority queues* (which enable us to find the maximum quickly) and the *selection* problem.

- How to store structured data (key-value-pairs). We gave some general-purpose data structures here, such as balanced binary search trees and skip lists. But we also considered special keys (words, integers, points) and special situations (biased search-requests, range-queries) and designed data structures for these.
- How to manipulate unstructured data (text). We particular focused on how to search in text and how to compress real-life text to make it shorter.
- Along the way, we encountered numerous techniques and insights that apply to many problems:
 - We studied how to prove *lower bounds* for a problem, and in particular studied decision trees.
 - We saw the idea of *randomization* so that rather than relying on the input to be good, we rely on our luck to be good.
 - We encountered *pre-processing*, which is the idea that doing some initial work pays off for getting faster queries later.

You will in the next few terms encounter numerous courses that build on top of this. Of particular note is the required course cs341 (Algorithms), which continues the general theme of ‘given a problem, what is the best algorithm to solve it?’, but focuses on graph algorithms (and general algorithm-design techniques) and does much more elaborate lower bounds.

There are also a number of optional courses that build on top of cs240 (and cs341), and that you should consider taking if you enjoyed cs240. The following is a short list:

- cs466, Algorithm Design & Analysis: This course continues the themes of cs341, and

in particular expands on randomized algorithms and on amortized analysis. You may also encounter online algorithms (which we saw briefly with the MTF heuristic) and fixed-parameter tractable algorithm (yet another way of measuring complexity of an algorithm).

- cs348/448, Databases: Entries in a data base are in essence d -dimensional points (for typically rather large d). The range-queries that we saw are one kind of queries that data bases must support. Also, data bases tend to be huge, and dealing with external memory becomes important, so B -trees (and their improvements) are studied in more detail.
- cs482, Biological Sequence Analysis: This course deals with computational problems surrounding DNA sequencing, gene finding and related topics. In particular, the analysis of DNA strings is important, and the course usually covers more on pattern matching and suffix trees and arrays.
- co487, Applied Cryptography: Our discussion of text compression was only concerned with making the text shorter. In this course offered by the Combinatorics&Optimization department, the focus shifts on making the coded text secure.
- cs454, Distributed Systems: This expands greatly on the idea of external memory management, and how to coordinate work if not only the data storage but also the data manipulation is broken across multiple locations.

Let us conclude these course notes with some messages that are not specific to any particular kind of data or problem, but that should apply to nearly any problem solving task in your future.

- Most problems have many possible solutions. Don't implement the first one you can think of. Can you be faster and/or more space-efficient?
- Always analyze possible solution for a problem on paper *first*. Often you can eliminate some obviously bad solutions. You should save the time needed to do implementation and experimentation for those cases where theoretical analysis is not helpful to distinguish two solutions.
- There isn't always one best solution for a problem. The answer to "which algorithm is best?" is almost always "it depends"—it depends on the type of input and the trade-off between run-time and space.

Overall, when you entered cs240, you were (hopefully) a good programmer, but the tools and techniques that you have seen in this course enable you to be not just a programmer, but also to be a good code designer and a good manager of programmers. We hope that you will find it useful.

Appendix A

Probability theory

These appendices contain some of the material that should be familiar from other math-courses. For this reason there is only a quick list of terms and results that are needed in cs240; for motivations, examples and proofs see the appropriate textbooks from other courses.

A *random trial* is any procedure that can be repeated and that has a well-defined set (finite or infinite) of possible outcomes. Usually there is more than one possible outcome. In this course we only consider random trials where the outcome is a number, i.e., in \mathbb{R} . A *probability distribution* is a mathematical function that gives the probabilities of outcomes in a random trial. It is called *discrete* if there is a finite or countably infinite set of outcomes and *continuous* otherwise.

A *random variable* (typically denoted X) is a function that depends on the outcome of one or more random trials. The *expected value* of a random variable X , denoted $E[X]$, is the average over all its values, weighted by the probability that they happen. For a discrete random variable, therefore $E[X] = \sum_{\text{outcomes } x} x \cdot P(X = x)$. For a continuous random variable (and under some assumptions on its distribution), we have $E[X] = \int_{\mathbb{R}} x f(x) dx$, where $f(x)$ (the *density function*) is such that $P(x_1 \leq X \leq x_2) = \int_{x_1}^{x_2} f(x) dx$. The expected value is often denoted by μ or μ_X .

The *variance* $V[X]$ of a random variable X is $E[(X - \mu_X)^2]$, and usually denoted σ^2 or σ_X^2 . Table A.1 lists some well-known probability distributions that we might need in cs240, together with their expected values and variances.

A.1 Some rules

Numerous rules are known concerning the expected value and the variance; the following lists the most important ones:

- $E[aX] = aE[X]$ for any random variable X and any constant a .
- $E[X + Y] = E[X] + E[Y]$ for any two random variables X, Y .
- $V[X + a] = V[X]$ for any random variable X and any constant a .

Name	Parameters	Probabilities	Expected value	Variance
Uniform (discrete)	integer $n \geq 1$	$P(X=i) = \frac{1}{n}$ for $i = 1, \dots, n$	$\frac{n+1}{2}$	$\frac{n^2-1}{12}$
Uniform (continuous)	numbers $a < b$	density $f(x) = \frac{1}{b-a}$ for $a \leq x \leq b$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
Bernoulli	$0 < p < 1$	$P(X=0) = 1-p$ $P(X=1) = p$	p	$p(1-p)$
Binomial	$0 < p < 1$, integer $n \geq 1$	$P(X=i) = \binom{n}{i} p^i (1-p)^{n-i}$ for $i = 0, \dots, n$	np	$np(1-p)$
Geometric	$0 < p < 1$	$P(X=i) = (1-p)^{i-1} p$ for $i = 1, 2, 3, \dots$	$\frac{1}{p}$	$\frac{1-p}{p^2}$

Table A.1: Some probability distributions.

- $V[X+Y] = V[X] + V[Y]$ for two random variables X, Y that are *independent*, i.e., $P(X = x \text{ and } Y = y) = P(X = x) \cdot P(Y = y)$ for all x, y .
- $V[aX] = a^2 V[X]$ for any random variable X and any constant a .

To compute the expected value, the following reformulation is often easier to analyze.

Lemma A.1. *Let X be a random variable with outcomes in $\{0, 1, 2, \dots\}$ and finite expected value. Then $E[X] = \sum_{i \geq 1} P(X \geq i)$.*

Proof. For $i \geq 0$ we have $P(X = i) = P(X \geq i) - P(X \geq i+1)$. We can therefore write the expected value as a telescoping sum

$$\begin{aligned}
 E[X] &= \sum_{i \geq 0} i P(X = i) = \sum_{i \geq 0} i (P(X \geq i) - P(X \geq i+1)) \\
 &= \sum_{i \geq 0} \underbrace{i P(X \geq i)}_{=0 \text{ for } i=0} - \sum_{i \geq 0} i P(X \geq i+1) \\
 &= \sum_{i \geq 1} i P(X \geq i) - \sum_{i \geq 1} (i-1) P(X \geq i) = \sum_{i \geq 1} P(X \geq i)
 \end{aligned}$$

□

The following bounds on probabilities are also occasionally useful.

Lemma A.2 (Markov's inequality). *For any non-negative random variable X with finite expected value μ and any $a > 0$, we have*

$$P(X \geq a) \leq \frac{\mu}{a}.$$

Proof. We only give the proof for discrete variables. Since X is non-negative, we have

$$\mu = E[X] = \sum_x xP(X=x) = \underbrace{\sum_{0 \leq x < a} xP(X=x)}_{\geq 0} + \sum_{x \geq a} xP(X=x) \geq \sum_{x \geq a} aP(X=x) = aP(X \geq a).$$

8026

□

Lemma A.3 (Chebyshev's inequality). *Let X be a random variable with finite expected value μ and finite variance σ^2 . Then for any $t > 0$ we have*

$$P(|X - \mu| \geq t) \leq \frac{\sigma^2}{t^2} \quad \text{or equivalently} \quad P(|X - \mu| \geq t\sigma) \leq \frac{1}{t^2}.$$

Proof. Let Y be the random variable $(X - \mu)^2$, which is non-negative. Note that $E[Y] = E[(X - \mu)^2] = V[X] = \sigma^2$. By Markov's inequality,

$$P(|X - \mu| \geq t) = P((X - \mu)^2 \geq t^2) = P(Y \geq t^2) \leq \frac{E[Y]}{t^2} = \frac{\sigma^2}{t^2}.$$

8027

□

Lemma A.4 (Jensen's inequality). *If X is a random variable and f a concave function, then $E[f(X)] \leq f(E[X])$.*

Proof. To keep things simpler (and sufficient for where we need this in the course notes) we make two assumptions. First, X has a finite set \mathcal{I} of outcomes, hence $\mathcal{I}_f := \{f(i) : i \in \mathcal{I}\}$ is the set of all possible outcomes of $f(X)$. Second, f is injective.

We know that $E[X] = \sum_{i \in \mathcal{I}} i \cdot P(X=i)$. Recall that concavity means that $f(\lambda_1 x_1 + \cdots + \lambda_k x_k) \geq \lambda_1 f(x_1) + \cdots + \lambda_k f(x_k)$ for any $k \geq 1$, any x_1, \dots, x_k and any $\lambda_1, \dots, \lambda_k$ that sum to 1. Applying this by using $\{x_1, \dots, x_k\} = \mathcal{I}$ and $\sum_{i \in \mathcal{I}} P(X=i) = 1$ we get

$$f(E[X]) = f\left(\sum_{i \in \mathcal{I}} \underbrace{i}_{x_j} \cdot \underbrace{P(X=i)}_{\lambda_j}\right) \geq \sum_{i \in \mathcal{I}} P(X=i) \underbrace{f(i)}_{=: \ell} = \sum_{\ell \in \mathcal{I}_f} P(\underbrace{f(X)=\ell}_{\Leftrightarrow X=i \text{ since } f \text{ injective}}) \cdot \ell = E[f(X)].$$

8033

□

Appendix B

Modular arithmetic

Let $p \geq 2$ be an integer (frequently p will be a prime number). We call p the *modulus*. Two integers a and b are *congruent modulo p* if p is a divisor of their difference. Equivalently, $|a - b|$ is a multiple of p or $a = b + kp$ for some integer k (where k may be negative). The usual mathematical notation for ‘ a and b are congruent modulo p ’ is ‘ $a \equiv b \pmod{p}$ ’.

Nearly all arithmetic relationships carry over when ‘ $=$ ’ is replaced with ‘ $\equiv \pmod{p}$ ’. The following may get used without proof and hold for any $p \geq 2$ and any integers $a, b, c, a_1, b_1, a_2, b_2$:

- $a \equiv a \pmod{p}$
- $a \equiv b \pmod{p}$ implies $b \equiv a \pmod{p}$
- If $a \equiv b \pmod{p}$ and $b \equiv c \pmod{p}$ then $a \equiv c \pmod{p}$
- If $a \equiv b \pmod{p}$ then $a + k \equiv b + k \pmod{p}$ for any integer k .
- If $a \equiv b \pmod{p}$ then $ka \equiv kb \pmod{p}$ for any integer k .
- If $a_1 \equiv b_1 \pmod{p}$ and $a_2 \equiv b_2 \pmod{p}$ then $a_1 + a_2 \equiv b_1 + b_2 \pmod{p}$.
- If $a_1 \equiv b_1 \pmod{p}$ and $a_2 \equiv b_2 \pmod{p}$ then $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{p}$.

If p is a prime number, then the following also hold:

- For any integer $a \neq 0$, there exists $a' \in \{1, \dots, p-1\}$ such that $a \cdot a' \equiv 1 \pmod{p}$. This integer is usually denoted by a^{-1} , but should not be confused with the fraction $\frac{1}{a}$.¹
- If $a \equiv b \pmod{p}$ then $a^{-1} \equiv b^{-1} \pmod{p}$.
- If $ax \equiv b \pmod{p}$ for some integer x , then $x \equiv a^{-1}b \pmod{p}$.

B.1 Modular congruence vs. modulo operator

Given any integer a , there is a unique number b in $\{0, \dots, p-1\}$ such that $a \equiv b \pmod{p}$. For non-negative a , this number b is the same as the remainder when dividing a by p . The *modulo*

¹For those with background in group theory: For p a prime, the set $\{0, \dots, p\}$ forms a field with respect to operators $+$ and \cdot . This field is denoted \mathbb{Z}_p .

8058 *operation* is the operation that maps an integer a to this unique b . It sometimes is denoted by
 8059 $b \leftarrow a \bmod p$.

Note that this looks extremely similar to $a \equiv b \pmod{p}$, and in equations where both the modular congruence and the modulo operator are used, this can lead to confusion. Where helpful, we will therefore write ' $a \equiv_p b$ ' as shortcut for ' $a \equiv b \pmod{p}$ ', and ' $a \% p$ ' as an alternative for ' $a \bmod p$ '. We have

$$a \equiv_p b \text{ if and only if } (a - b) \% p = 0.$$

8060 Therefore the rules for modular congruence listed above can be transferred into rules for the
 8061 modulo operator. We list a few examples here:

8062 **Observation B.1.** *For any two integers a, b , we have $a \% p + b \% p \equiv_p (a + b) \% p$.*

8063 *Proof.* Write $a = k_a p + r_a$ and $b = k_b p + r_b$, where k_a, k_b, r_a, r_b are integers and $r_a = a \% p$ and
 8064 $r_b = b \% p$. We have $(k_a + k_b)p \equiv_p 0$, and so $r_a + r_b \equiv_p r_a + r_b + k_a p + k_b p = a + b \equiv_p (a + b) \% p$. \square

8065 **Observation B.2.** *For any two integers a, b , we have $(a \cdot b) \% p \equiv_p a \% p \cdot b \% p$.*

Proof. Write a and b as above. Then

$$a \cdot b = (k_a p + r_a) \cdot (k_b p + r_b) = (k_a k_b p + r_a k_b + k_a r_b) p + r_a r_b \equiv_p r_a r_b = (a \% p)(b \% p)$$

8066 which implies the result. \square

Bibliography

- [AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004. URL: <http://www.jstor.org/stable/3597229>.
- [AS89] Cecilia R. Aragon and Raimund Seidel. Randomized search trees. In *Foundations of Computer Science (FOCS'89)*, pages 540–545. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63531.
- [AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, sep 1988. doi:10.1145/48529.48535.
- [AVL62] G.M. Adel'son-Vel'skii and E.M. Landis. An algorithm for organization of information (in Russian). *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.
- [Bac94] P.G.H. Bachmann. *Die analytische Zahlentheorie*. Teubner, 1894. Available as 2018 reprint from *LULU PR*, ISBN 978-1332635641.
- [BEHW15] Dominic Berz, Marco Engstler, Moritz Heindl, and Florian Waibel. Comparison of lossless data compression methods, 03 2015. Technical Reports in Computing Science No. CS-07-2015, University of Applied Sciences Kempten. URL: https://www.researchgate.net/publication/335572104_Comparison_of_lossless_data_compression_methods.
- [Bel58] D. A. Bell. The Principles of Sorting. *The Computer Journal*, 1(2):71–77, 01 1958. doi:10.1093/comjnl/1.2.71.
- [Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. doi:10.1145/361002.361007.
- [Ben79] Jon Louis Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8(5):244–251, 1979. doi:10.1016/0020-0190(79)90117-0.
- [Ben86] Jon Louis Bentley. *Programming pearls*. Addison-Wesley, 1986.

- 8093 [Ber95] Jean Berstel. Axel Thue's papers on repetitions in words: a translation.
8094 Technical Report 20, Laboratoire d'algèbre, de combinatoire et d'informatique
8095 mathématique (LACIM), Université de Québec à Montréal, 1995. Available at
8096 <https://lacim.uqam.ca/wp-content/uploads/Publications/20.pdf> (last ac-
8097 cessed Jan 20, 2022).
- 8098 [BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered
8099 indexes. *Acta Inf.*, 1(3):173–189, sep 1972. doi:10.1007/BF00288683.
- 8100 [BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Com-
8101 mun. ACM*, 20(10):762–772, oct 1977. doi:10.1145/359842.359859.
- 8102 [BM85] Jon Louis Bentley and Catherine C. McGeoch. Amortized analyses of self-
8103 organizing sequential search heuristics. *Commun. ACM*, 28(4):404–411, 1985.
8104 doi:10.1145/3341.3349.
- 8105 [Bro13] Gerth Stølting Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro
8106 López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data
8107 Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the
8108 Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*,
8109 pages 150–163. Springer, 2013. doi:10.1007/978-3-642-40273-9_11.
- 8110 [CLP11] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range
8111 searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld,
8112 editors, *Proceedings of the 27th ACM Symposium on Computational Geometry,
8113 Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.
8114 1998198.
- 8115 [Com79] Douglas Comer. Ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.
8116 doi:10.1145/356770.356776.
- 8117 [CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J.
8118 Comput. Syst. Sci.*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.
- 8119 [dBCvKO08] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars.
8120 *Computational geometry: algorithms and applications, 3rd Edition*. Springer,
8121 2008.
- 8122 [Dou59] A. S. Douglas. Techniques for the Recording of, and Reference to data in a
8123 Computer. *The Computer Journal*, 2(1):1–9, 01 1959. doi:10.1093/comjnl/2.
8124 1.1.
- 8125 [Dum56] A. I. Dumey. Indexing for rapid random-access memory. *Computers and Automa-
8126 tion*, 5(12):6–9, 1956.

- 8127 [Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE*
8128 *Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.
8129 1975.1055349.
- 8130 [Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th An-*
8131 *annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach,*
8132 *Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997.
8133 doi:10.1109/SFCS.1997.646102.
- 8134 [FB74] Raphael Finkel and Jon Bentley. Quad trees: A data structure for retrieval on
8135 composite keys. *Acta Inf.*, 4:1–9, 03 1974. doi:10.1007/BF00288933.
- 8136 [FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm
8137 for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*,
8138 3(3):209–226, 1977. doi:10.1145/355744.355745.
- 8139 [FK17] Johannes Fischer and Dominik Köppl. Practical evaluation of lempel-ziv-78 and
8140 lempel-ziv-welch tries. In Gabriele Fici, Marinella Sciortino, and Rossano Ven-
8141 turini, editors, *String Processing and Information Retrieval - 24th International*
8142 *Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*,
8143 volume 10508 of *Lecture Notes in Computer Science*, pages 191–207. Springer,
8144 2017. doi:10.1007/978-3-319-67428-5_16.
- 8145 [Flo64] R.W. Floyd. Algorithm 245: Treesort3. *Commun. ACM*, 7(12):701, 1964.
- 8146 [FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong.
8147 Extendible hashing - A fast access method for dynamic files. *ACM Trans. Database*
8148 *Syst.*, 4(3):315–344, 1979. doi:10.1145/320083.320092.
- 8149 [FO82] Philippe Flajolet and Andrew Odlyzko. The average height of binary trees and
8150 other simple trees. *Journal of Computer and System Sciences*, 25(2):171–213,
8151 1982. doi:https://doi.org/10.1016/0022-0000(82)90004-6.
- 8152 [Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, sep 1960. doi:
8153 10.1145/367390.367400.
- 8154 [GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for
8155 text: Pat trees and pat arrays. In William B. Frakes and Ricardo A. Baeza-
8156 Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82.
8157 Prentice-Hall, 1992.
- 8158 [GM98] Anna Gambin and Adam Malinowski. Randomized meldable priority queues.
8159 In Branislav Rován, editor, *SOFSEM '98: Theory and Practice of Informatics*,
8160 *25th Conference on Current Trends in Theory and Practice of Informatics, Jasná,*

- 8161 *Slovakia, November 21-27, 1998, Proceedings*, volume 1521 of *Lecture Notes in*
 8162 *Computer Science*, pages 344–349. Springer, 1998. doi:10.1007/3-540-49477-4\
 8163 _26.
- 8164 [GR93] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachan-
 8165 dran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium*
 8166 *on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 165–
 8167 174. ACM/SIAM, 1993. URL: [http://dl.acm.org/citation.cfm?id=313559.](http://dl.acm.org/citation.cfm?id=313559.313676)
 8168 313676.
- 8169 [Hoa61] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, jul 1961.
 8170 doi:10.1145/366622.366647.
- 8171 [Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16,
 8172 01 1962. arXiv:[https://academic.oup.com/comjnl/article-pdf/5/1/10/](https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf)
 8173 [1111445/050010.pdf](https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf), doi:10.1093/comjnl/5.1.10.
- 8174 [Huf52] David A. Huffman. A method for the construction of minimum-redundancy
 8175 codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi:10.1109/JRPROC.
 8176 1952.273898.
- 8177 [KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching
 8178 in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 8179 [Knu92] Donald E. Knuth. *Axioms and Hulls*, volume 606 of *Lecture Notes in Computer*
 8180 *Science*. Springer, 1992. doi:10.1007/3-540-55611-7.
- 8181 [Knu97] Donald Ervin Knuth. *The art of computer programming, Volume I: Fundamental*
 8182 *Algorithms, 3rd Edition*. Addison-Wesley, 1997.
- 8183 [Knu98] Donald Ervin Knuth. *The art of computer programming, Volume III: Sorting and*
 8184 *Searching, 2nd Edition*. Addison-Wesley, 1998.
- 8185 [KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching
 8186 algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.
- 8187 [KW66] Alan G. Konheim and Benjamin Weiss. An occupancy discipline and applications.
 8188 *SIAM Journal on Applied Mathematics*, 14(6):1266–1274, 1966. URL: <https://www.jstor.org/stable/2946240>.
 8189
- 8190 [Lan09] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. B.G.
 8191 Teubner, 1909.
- 8192 [Lan63] Walter I. Landauer. The balanced tree and its utilization in information retrieval.
 8193 *IEEE Trans. Electron. Comput.*, 12:863–871, 1963. doi:[https://doi.org/10.](https://doi.org/10.1109/PGEC.1963.263589)
 8194 [1109/PGEC.1963.263589](https://doi.org/10.1109/PGEC.1963.263589).

- 8195 [Lar80] Per-Åke Larson. Linear hashing with partial expansions. In *Sixth International*
8196 *Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec,*
8197 *Canada, Proceedings*, pages 224–232. IEEE Computer Society, 1980.
- 8198 [LC20] Chen Luo and Michael J. Carey. LSM-based storage techniques: a survey. *VLDB*
8199 *J.*, 29(1):393–418, 2020. doi:10.1007/s00778-019-00555-y.
- 8200 [Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Sixth*
8201 *International Conference on Very Large Data Bases, October 1-3, 1980, Montreal,*
8202 *Quebec, Canada, Proceedings*, pages 212–223. IEEE Computer Society, 1980.
- 8203 [LM93] G.S. Lueker and M. Molodowitch. More analysis of double hashing. *Combinator-*
8204 *ica*, 13:83–96, 1993. doi:https://doi.org/10.1007/BF01202791.
- 8205 [Lue78] George S. Lueker. A data structure for orthogonal range queries. In *19th Annual*
8206 *Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA,*
8207 *16-18 October 1978*, pages 28–34. IEEE Computer Society, 1978. doi:10.1109/
8208 *SFCS.1978.1*.
- 8209 [LW80] D. T. Lee and C. K. Wong. Quintary trees: A file structure for multidimensional
8210 database systems. *ACM Trans. Database Syst.*, 5(3):339–353, 1980. doi:10.1145/
8211 *320613.320618*.
- 8212 [Mac03] David Mackay. *Information Theory, Inference and Learning Algorithms*. Cam-
8213 bridge University Press, 2003.
- 8214 [McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J.*
8215 *ACM*, 23(2):262–272, apr 1976. doi:10.1145/321941.321946.
- 8216 [McC85] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276,
8217 1985. doi:10.1137/0214021.
- 8218 [Men43] L.F. Menabrea. *Sketch of the Analytical Engine invented by Charles Babbage ...*
8219 *with notes by the translator. Extracted from the 'Scientific Memoirs,'.* R. & J.
8220 E. Taylor, 1843. The translator’s notes signed: A.L.L. ie. Augusta Ada King,
8221 Countess Lovelace. URL: <https://books.google.ca/books?id=hPRmnQEACAAJ>.
- 8222 [MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string
8223 searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 8224 [Mor68] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded
8225 in alphanumeric. *J. ACM*, 15(4):514–534, oct 1968. doi:10.1145/321479.
8226 *321481*.

- 8227 [MPS92] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists.
8228 In Greg N. Frederickson, editor, *Proceedings of the Third Annual ACM/SIGACT-*
8229 *SIAM Symposium on Discrete Algorithms, 27-29 January 1992, Orlando, Florida,*
8230 *USA*, pages 367–375. ACM/SIAM, 1992. URL: [http://dl.acm.org/citation.](http://dl.acm.org/citation.cfm?id=139404.139478)
8231 [cfm?id=139404.139478](http://dl.acm.org/citation.cfm?id=139404.139478).
- 8232 [MR98] Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*,
8233 45(2):288–323, mar 1998. doi:10.1145/274787.274812.
- 8234 [NR72] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance.
8235 In Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg,
8236 editors, *Proceedings of the 4th Annual ACM Symposium on Theory of Computing,*
8237 *May 1-3, 1972, Denver, Colorado, USA*, pages 137–142. ACM, 1972. doi:10.
8238 1145/800152.804906.
- 8239 [OCGO96] Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O’Neil. The
8240 log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
8241 doi:10.1007/s002360050048.
- 8242 [Pet57] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*,
8243 1(2):130–146, apr 1957. doi:10.1147/rd.12.0130.
- 8244 [PIA78] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search - A log log N
8245 search. *Commun. ACM*, 21(7):550–553, 1978. doi:10.1145/359545.359557.
- 8246 [PR77] Yehoshua Perl and Edward M. Reingold. Understanding the complexity of
8247 interpolation search. *Inf. Process. Lett.*, 6(6):219–222, 1977. doi:10.1016/
8248 0020-0190(77)90072-2.
- 8249 [PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*,
8250 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.
- 8251 [PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix
8252 array construction algorithms. *ACM Comput. Surv.*, 39(2):4–es, jul 2007. doi:
8253 10.1145/1242471.1242472.
- 8254 [Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun.*
8255 *ACM*, 33(6):668–676, jun 1990. doi:10.1145/78973.78977.
- 8256 [Rab80] Michael O Rabin. Probabilistic algorithm for testing primality.
8257 *Journal of Number Theory*, 12(1):128–138, 1980. URL: [https:](https://www.sciencedirect.com/science/article/pii/0022314X80900840)
8258 [//www.sciencedirect.com/science/article/pii/0022314X80900840](https://www.sciencedirect.com/science/article/pii/0022314X80900840),
8259 doi:[https://doi.org/10.1016/0022-314X\(80\)90084-0](https://doi.org/10.1016/0022-314X(80)90084-0).

- 8260 [RC67] A.H. Robinson and C. Cherry. Results of a prototype television bandwidth com-
8261 pression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967. doi:10.1109/
8262 PROC.1967.5493.
- 8263 [RS62] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions
8264 of prime numbers. *Illinois Journal of Mathematics*, 6(1):64 – 94, 1962. doi:
8265 10.1215/ijm/1255631807.
- 8266 [Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Com-*
8267 *put. Surv.*, 16(2):187–260, 1984. doi:10.1145/356924.356930.
- 8268 [Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-
8269 Wesley, 1990.
- 8270 [Sed98] Robert Sedgewick. *Algorithms in C++ - Parts 1-4: Fundamentals, Data Struc-*
8271 *tures, Sorting, Searching*. Addison-Wesley-Longman, 1998.
- 8272 [Sha48] C.E. Shannon. A mathematical theory of communication. *Bell System Technical*
8273 *Journal*, 27:379–423, 623–656, 1948.
- 8274 [SS85] Jörg-Rüdiger Sack and Thomas Strothotte. An algorithm for merging heaps. *Acta*
8275 *Informatica*, 22(2):171–186, 1985. doi:10.1007/BF00264229.
- 8276 [ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search
8277 trees. *J. ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.
- 8278 [Tho15] Mikkel Thorup. High speed hashing for integers and strings. *CoRR*,
8279 abs/1504.06804, 2015. URL: <http://arxiv.org/abs/1504.06804>.
- 8280 [Thu12] A. Thue. *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen*. Skrifter
8281 udgivne af Videnskabselskabet i Christiania. 1,Math.Nat.wiss.Kl.1912,1. Jacob
8282 Dybwad, 1912.
- 8283 [Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260,
8284 1995. doi:10.1007/BF01206331.
- 8285 [Ver88] A.Inkeri Verkamo. External quicksort. *Performance Evaluation*, 8(4):271–288,
8286 1988. doi:[https://doi.org/10.1016/0166-5316\(88\)90029-6](https://doi.org/10.1016/0166-5316(88)90029-6).
- 8287 [Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun.*
8288 *ACM*, 21(4):309–315, 1978. doi:10.1145/359460.359478.
- 8289 [Vui80] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*,
8290 23(4):229–239, apr 1980. doi:10.1145/358841.358852.

- 8291 [Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium*
8292 *on Switching and Automata Theory (swat 1973)*, pages 1–11, 1973. doi:10.1109/
8293 SWAT.1973.13.
- 8294 [Wel84] Terry A. Welch. A technique for high-performance data compression. *Computer*,
8295 17(6):8–19, 1984. doi:10.1109/MC.1984.1659158.
- 8296 [Wil59] Francis A. Williams. Handling identifies as internal symbols in language proces-
8297 sors. *Commun. ACM*, 2(6):21–24, jun 1959. doi:10.1145/368336.368352.
- 8298 [Wil64] J.W.J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–348, 1964.
- 8299 [Wil79] D.E. Williard. The super-b-tree algorithm, 1979. Report TR-03-79, Aiken Com-
8300 put. Lab., Harvard Univ., Cambridge, MA.
- 8301 [Win60] P. F. Windley. Trees, Forests and Rearranging. *The Computer Journal*, 3(2):84–
8302 88, 01 1960. doi:10.1093/comjnl/3.2.84.
- 8303 [WNM16] Sebastian Wild, Markus E. Nebel, and Hosam M. Mahmoud. Analysis of quick-
8304 elect under Yaroslavskiy’s dual-pivoting algorithm. *Algorithmica*, 74(1):485–506,
8305 2016. doi:10.1007/s00453-014-9953-x.
- 8306 [YY76] Andrew Chi-Chih Yao and F. Frances Yao. The complexity of searching an ordered
8307 random table (extended abstract). In *17th Annual Symposium on Foundations of*
8308 *Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 173–177.
8309 IEEE Computer Society, 1976. doi:10.1109/SFCS.1976.32.
- 8310 [ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate
8311 coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi:
8312 10.1109/TIT.1978.1055934.

List of Algorithms

8313			
8314	1.1	<i>bubble-sort</i> ($A, n \leftarrow A.size$)	9
8315	1.1	<i>bubble-sort</i> ($A, n \leftarrow A.size$) // repeated	11
8316	1.2	<i>insertion-sort</i> ($A, n \leftarrow A.size$)	12
8317	1.2	<i>insertion-sort</i> ($A, n \leftarrow A.size$)	29
8318	1.1	<i>bubble-sort</i> ($A, n \leftarrow A.size$) // repeated	30
8319	1.3	<i>merge-sort</i> ($A, n \leftarrow A.size, \ell \leftarrow 0, r \leftarrow n - 1, S \leftarrow \text{NIL}$)	33
8320	1.4	<i>merge</i> (A, ℓ, m, r, S)	34
8321	1.5	<i>sortedness-tester</i> (A, n)	39
8322	1.6	<i>avgCaseDemo</i> (A, n)	42
8323	1.7	<i>expectedDemo</i> ($A, n \leftarrow A.size$)	47
8324	2.1	<i>priority-queue-sort</i> ($A, n \leftarrow A.size$)	58
8325	2.2	<i>selection-sort</i> ($A, n \leftarrow A.size$)	59
8326	2.3	<i>fix-up</i> (A, i)	65
8327	2.4	<i>binaryHeap::insert</i> (k, v)	65
8328	2.5	<i>fix-down</i> ($A, i, n \leftarrow A.size$)	66
8329	2.6	<i>binaryHeap::deleteMax</i> ()	67
8330	2.7	<i>heap-sort</i> ($A, n \leftarrow A.size$)	67
8331	2.8	<i>heapify</i> ($A, n \leftarrow A.size$)	68
8332	2.9	<i>heapify-recursive</i> ($A, i \leftarrow \text{root}(), n \leftarrow A.size$)	71
8333	2.10	<i>meldableHeap::merge</i> (r_1, r_2)	73
8334	2.11	<i>binomialHeap::makeProper</i> ()	79
8335	3.1	<i>partition</i> (A, p, n) // simple variant	85
8336	3.2	<i>partition</i> (A, p, n) // Hoare's more sophisticated approach	86
8337	3.3	<i>quick-select</i> ($A, k, n \leftarrow A.size$)	90
8338	3.4	<i>inside-out-shuffle</i> ($A, n \leftarrow A.size$)	93
8339	3.5	<i>randomized-quick-select</i> ($A, k, n \leftarrow A.size$)	93
8340	3.6	<i>quick-sort</i> ($A, n \leftarrow A.size$)	97
8341	3.6	<i>quick-sort</i> ($A, n \leftarrow A.size$) // simple version repeated	101
8342	3.7	<i>quick-sort</i> ($A, n \leftarrow A.size$) // improved version	102

8343	3.8	<i>bucket-sort</i> ($A, n \leftarrow A.size, \ell \leftarrow 0, r \leftarrow n-1, d$)	110
8344	3.9	<i>MSD-radix-sort</i> ($A, n \leftarrow A.size, \ell \leftarrow 0, r \leftarrow n-1, d \leftarrow 1$)	111
8345	3.10	<i>LSD-radix-sort</i> ($A, n \leftarrow A.size$)	112
8346	4.1	<i>AVL::insert</i> (k, v)	124
8347	4.2	<i>rotate-left</i> (z)	127
8348	4.3	<i>restructure</i> (x, y, z)	128
8349	4.4	<i>AVL::delete</i> (k)	129
8350	4.5	<i>scapegoatTree::insert</i> (k, v)	134
8351	5.1	<i>treap::fix-up-with-rotations</i> (z)	146
8352	5.2	<i>treap::insert</i> (k, v)	146
8353	5.3	<i>getPredecessors</i> (k)	148
8354	5.4	<i>skipList::search</i> (k)	149
8355	5.5	<i>skipList::insert</i> (k, v)	150
8356	5.6	<i>skipList::delete</i> (k)	151
8357	5.7	<i>MTF-BST::insert</i> (k, v)	162
8358	5.8	<i>splayTree::insert</i> (k, v)	164
8359	6.1	<i>binary-search</i> (A, n, k)	172
8360	6.2	<i>binary-search-optimized</i> (A, n, k)	174
8361	6.3	<i>interpolation-search</i> (A, n, k)	178
8362	6.4	<i>interpolation-search-improved</i> (A, n, k)	180
8363	6.5	<i>strcmp</i> ($w_1, w_2, i \leftarrow 0, i' \leftarrow w_1.length-1$)	187
8364	6.6	<i>trie::search</i> ($v \leftarrow root, d \leftarrow 0, x$)	189
8365	6.7	<i>trie::prefixSearch</i> ($v \leftarrow root, d \leftarrow 0, x$)	191
8366	6.8	<i>compressedTrie::search</i> ($v \leftarrow root, x$)	197
8367	7.1	<i>probeSequenceHash::insert</i> (k, v)	210
8368	7.2	<i>probeSequenceHash::search</i> (k)	210
8369	7.3	<i>probeSequenceHash::delete</i> (k)	211
8370	7.4	<i>cuckooHash::insert</i> (k, v)	216
8371	7.5	<i>word-hash-value</i> (w, R, M)	226
8372	8.1	<i>quadTree::rangeSearch</i> ($r \leftarrow root, A$)	239
8373	8.2	<i>kdTree::rangeSearch</i> ($r \leftarrow root, A$)	244
8374	8.3	<i>BST::rangeSearch-recursive</i> ($r \leftarrow root, x_1, x_2$)	249
8375	8.4	<i>BST::rangeSearch</i> (x_1, x_2)	251
8376	8.5	<i>rangeTree::rangeSearch</i> (x_1, x_2, y_1, y_2)	252
8377	9.1	<i>bruteForce::patternMatching</i> (T, P)	263
8378	9.2	<i>KarpRabin::patternMatching</i> (T, P) // naive version	266

8379	9.3	<i>KarpRabin::patternMatching</i> (T, P)	268
8380	9.4	<i>KMP::patternMatching</i> (T, P)	275
8381	9.5	<i>KMP::failureArray</i> (P)	279
8382	9.6	<i>BoyerMoore::patternMatching</i> (T, P)	283
8383	9.7	<i>BoyerMoore::lastOccurrenceArray</i> (P)	284
8384	9.8	<i>suffixArray::patternMatching</i> (T, P)	295
8385	10.1	<i>charByChar::encoding</i> (S, C)	304
8386	10.2	<i>prefixFree::decoding</i> (C, S)	305
8387	10.3	<i>prefixFree::encoding</i> (S, C)	306
8388	10.4	<i>Huffman::buildTrie</i> (Σ_S, f)	309
8389	10.5	<i>Huffman::encoding</i> (S, C)	310
8390	10.6	<i>RLE::encoding</i> (S, C)	316
8391	10.7	<i>RLE::decoding</i> (C, S)	317
8392	10.8	<i>LZW::encoding</i> (S, C)	320
8393	10.9	<i>LZW::decoding</i> (C, S)	322
8394	10.10	<i>LZW::dictionaryLookup</i> ($D, code$)	323
8395	10.11	<i>MTF::encoding</i> (S, C)	328
8396	10.12	<i>MTF::decoding</i> (C, S)	328
8397	10.13	<i>BWT::encoding</i> (S, C)	331
8398	10.14	<i>BWT::decoding</i> (C, S)	333
8399	11.1	<i>merge</i> (S_1, S_2, S) // based on streams	342
8400	11.2	<i>d-way merge</i> (S_1, \dots, S_d, S)	342
8401	11.3	<i>24Tree::search</i> ($k, v \leftarrow root, p \leftarrow NIL$)	349
8402	11.4	<i>24Tree::insert</i> (k)	350
8403	11.5	<i>24Tree::delete</i> (k)	353
8404	11.6	<i>RBTree::insert</i> (k)	357
8405	11.7	<i>trieOfBlocks::insert</i> (k, v)	368
8406	11.8	<i>extendibleHashing::insert</i> (k, v)	372

Index

- 8407 O , 13
- 8408 Λ , 263
- 8409 Ω , 16
- 8410 Θ , 17
- 8411 ω , 20
- 8412 o , 20
- 8413 2-4-trees, 347

- 8414 a-b-tree, 357
- 8415 abstract data type (ADT), 55
 - 8416 ADT Dictionary, 115
 - 8417 ADT Priority Queue, 57
 - 8418 ADT Queue, 56
 - 8419 ADT Stack, 56
- 8420 access-frequency, 156
- 8421 access-probability, 156
- 8422 adaptive encoding, 318
- 8423 algorithm, 7
 - 8424 comparison-based, 106
 - 8425 deterministic, 8
 - 8426 non-deterministic, 8
 - 8427 offline, 159
 - 8428 online, 159
 - 8429 randomized, 8, 45
- 8430 almost-sorted array, 102
- 8431 alphabet, 185
- 8432 amortized analysis, 60
- 8433 ancestor, 61
- 8434 arithmetic sequence, 5
- 8435 arity, 62
- 8436 array
 - 8437 capacity, 59
 - 8438 size, 59
- 8439 ASCII, 185
- 8440 associate structure, 247
- 8441 asymptotic
 - 8442 growth rates, 18
 - 8443 lower bound, 16
 - 8444 notation, 13
 - 8445 upper bound, 13
- 8446 asymptotically
 - 8447 strictly larger, 20
 - 8448 strictly smaller, 20
 - 8449 the same, 17
- 8450 attribute, 231
- 8451 auxiliary space, 10
- 8452 average-case run-time, 38
- 8453 AVL-tree, 120
 - 8454 height, 122
 - 8455 property, 120

- 8456 B^+ -tree, 364
- 8457 B-tree, 361
- 8458 balanced binary search tree, 120
- 8459 best-case run-time, 13
- 8460 biased search-request, 156
- 8461 big- Ω , 16
- 8462 big- O , 13
- 8463 binary bijective numeration, 326
- 8464 binary heap, 63
- 8465 binary search, 117, 172
- 8466 binary search tree, 117
 - 8467 AVL-tree, 120
 - 8468 balanced, 120
 - 8469 perfectly balanced, 134
 - 8470 rotation, 125

- 8471 scapegoat tree, 133
- 8472 splay tree, 163
- 8473 treap, 144
- 8474 binary tree, 61
 - 8475 complete, 62
 - 8476 height, 62
 - 8477 in-order, 62
 - 8478 interior node, 62
 - 8479 leaf, 62
 - 8480 level, 61
 - 8481 level-order, 62
 - 8482 post-order, 62
 - 8483 pre-order, 62
- 8484 binary trie, 187
- 8485 binomial heap, 76
- 8486 bit, 185
- 8487 bit-string, 185
- 8488 black-depth, 355
- 8489 block, 339
 - 8490 transfer, 339
- 8491 boundary node, 238
- 8492 bounding box, 233
- 8493 bucket, 204
- 8494 bucket-sort, 110
- 8495 cache-oblivious, 340
- 8496 Catalan number, 144
- 8497 character, 185
- 8498 character-by-character encoding, 302
- 8499 Chebyshev's inequality, 183, 379
- 8500 code-word, 302
- 8501 coded text, 299
- 8502 collision, 203
- 8503 comparison-based algorithm, 39, 106
- 8504 competitive analysis, 159
- 8505 complete binary tree, 62
- 8506 complexity, 10
- 8507 compressed trie, 195
- 8508 compression ratio, 300
- 8509 concatenation, 315
- 8510 concave function, 75
- 8511 coordinate, 231
- 8512 cyclic shift, 329
- 8513 decision tree, 106
- 8514 decoding, 300
- 8515 depth, 61
- 8516 deterministic algorithm, 8
- 8517 deterministic finite automaton (DFA), 272
- 8518 direct addressing, 201
- 8519 disk transfer, 339
- 8520 double rotation, 126
- 8521 dynamic array, 49, 59
- 8522 dynamic scenario, 157
- 8523 Elias-gamma-code, 314
- 8524 empty tree, 61
- 8525 empty word, 263
- 8526 encoding, 299
- 8527 end-of-word symbol, 186
- 8528 expected access-cost, 156
- 8529 expected running time, 46
- 8530 expected value, 6, 377
- 8531 exponential run-time, 19
- 8532 extendible hashing, 366
- 8533 external memory model, 338
- 8534 factorial, 5
- 8535 failure-arc, 273
- 8536 Fibonacci hashing, 219
- 8537 Fibonacci numbers, 123
- 8538 finite automaton
 - 8539 deterministic, 272
 - 8540 non-deterministic, 271
- 8541 fixed-length encoding, 303
- 8542 flagged tree, 76
- 8543 flattening a string, 225
- 8544 full level, 62
- 8545 geometric sequence, 5
- 8546 good-suffix heuristic, 282
- 8547 guess, 262
- 8548 harmonic sequence, 5

- 8549 hash table, 203
- 8550 hash-function, 202
 - 8551 Carter-Wegman, 220
 - 8552 for words, 226
 - 8553 modular, 212
 - 8554 multiplicative, 212
- 8555 hash-function expansion, 370
- 8556 hash-table, 203
- 8557 hashing, 202
 - 8558 cuckoo, 215
 - 8559 extendible, 366
 - 8560 probe sequence, 209
 - 8561 universal, 222
 - 8562 with chaining, 204
- 8563 haystack, 262
- 8564 heap, 63
- 8565 heap-sort, 67
- 8566 heapify, 68
- 8567 height of a binary tree, 62
- 8568 Hoare's partition, 85
- 8569 Horner's rule, 226
- 8570 Huffman encoding, 310
- 8571 Huffman trie, 308
- 8572 in-order in a binary tree, 62
- 8573 in-place algorithm, 67, 86
- 8574 independent random variables, 378
- 8575 insertion-sort, 60
- 8576 inside node, 238
- 8577 instance, 6
- 8578 interior node, 62
- 8579 internal memory, 339
- 8580 interpolation search, 178
- 8581 ISO-8859, 185
- 8582 Jensen's inequality, 379
- 8583 Karp-Rabin pattern matching, 268
- 8584 kd-tree, 241
- 8585 key-comparison, 68, 90
- 8586 key-value-pair (KVP), 7, 57
- 8587 KMP-automaton, 273
- 8588 Knuth-Morris-Pratt pattern matching, 275
- 8589 l'Hôpital's rule, 26
- 8590 Las Vegas algorithm, 45
- 8591 last-occurrence heuristic, 282
- 8592 lazy deletion, 119, 210
- 8593 leaf of a binary tree, 62
- 8594 leaf-reference, 190
- 8595 left rotation, 126
- 8596 left subtree, 61
- 8597 Lempel-Ziv-Welch encoding, 320
- 8598 level, 61
- 8599 level-order in a binary tree, 62
- 8600 limit-rule, 23
- 8601 linear run-time, 19
- 8602 linearithmic run-time, 19
- 8603 link, 61
- 8604 little- ω , 20
- 8605 little- o , 20
- 8606 logarithm, 5
 - 8607 rule for asymptotics, 26
- 8608 logarithmic run-time, 18
- 8609 loop invariant, 29
- 8610 lower bound
 - 8611 searching, 172
 - 8612 sorting, 106
- 8613 main memory, 339
- 8614 Markov's inequality, 378
- 8615 max-oriented priority queue, 57
- 8616 median-finding, 84
- 8617 merge, 33
 - 8618 d -way, 342
 - 8619 2-way, 33
- 8620 merge-sort, 33
 - 8621 d -way, 343
 - 8622 2-way, 33
- 8623 min-oriented priority queue, 57
- 8624 modular hash function, 203, 212
- 8625 modulo
 - 8626 arithmetic rules, 381

- 8627 congruent, 381
- 8628 operator, 382
- 8629 Monte Carlo algorithm, 45
- 8630 Morse code, 302
- 8631 move-to-front heuristic (MTF), 159
- 8632 move-to-front transform, 327
- 8633 multi-way
 - 8634 node, 62, 347
 - 8635 tree, 62
 - 8636 trie, 197
- 8637 multiplication hash function, 212
- 8638 needle, 262
- 8639 NFA, 271
- 8640 NIL, 10, 61
- 8641 node, 61
- 8642 non-deterministic algorithm, 8
- 8643 non-deterministic finite automaton, 271
- 8644 oct-tree, 239
- 8645 offline algorithm, 159
- 8646 online algorithm, 159
- 8647 outside node, 238
- 8648 overflow at *a-b*-tree node, 349
- 8649 page load, 339
- 8650 Patricia trie, 195
- 8651 pattern matching, 261
- 8652 perfectly balanced tree, 134
- 8653 poly-logarithmic run-time, 18
- 8654 polynomial, 26
- 8655 polynomial rule for asymptotics, 26
- 8656 polynomial run-time, 19
- 8657 post-order in a binary tree, 62
- 8658 pre-emptive splitting, 363
- 8659 pre-order in a binary tree, 62
- 8660 pre-processing, 264
- 8661 predecessor, 118
- 8662 prefix, 186, 263
- 8663 prefix-free code, 305
- 8664 prefixSearch, 190
- 8665 primary structure, 247
- 8666 primitive operation, 10
- 8667 priority queue, 57
- 8668 priority search tree, 144
- 8669 probability distribution, 377
 - 8670 Bernoulli, 377
 - 8671 binomial, 377
 - 8672 geometric, 377
 - 8673 uniform, 377
- 8674 probe, 209
 - 8675 sequence, 209
- 8676 problem, 6
- 8677 proof from first principle, 14
- 8678 pruned trie, 192
- 8679 pseudo-random number generator (PRNG), 45
- 8680 pseudocode, 8
- 8681 quad-tree, 233
- 8682 quadratic run-time, 19
- 8683 Queue, 56
- 8684 radix, 109
 - 8685 tree, 187
- 8686 Random Access Machine (RAM), 9
- 8687 random variable, 377
- 8688 randomized algorithm, 8, 45
- 8689 range search, 230
 - 8690 orthogonal, 231
- 8691 range tree, 247
- 8692 re-hashing, 203, 207
- 8693 recursion depth, 98
- 8694 recursion stack, 36
- 8695 recursion tree, 98
- 8696 recursive function, 35
- 8697 reverse-order searching, 282
- 8698 right rotation, 125
- 8699 right subtree, 61
- 8700 root, 61
- 8701 rotation, 125
 - 8702 *a-b*-tree, 351
 - 8703 double-left, 126
 - 8704 double-right, 126

- 8705 left, 126
- 8706 right, 125
- 8707 single, 126
- 8708 run-length encoding, 314
- 8709 run-time, 10
- 8710 scapegoat tree, 132, 133
- 8711 search
 - 8712 binary, 172
 - 8713 interpolation, 178
- 8714 searching
 - 8715 lower bound, 172
- 8716 selection problem, 83
- 8717 selection sort, 59
- 8718 sentinel, 147
- 8719 single rotation, 126
- 8720 single-character encoding, 302, 313
- 8721 size of an instance, 10
- 8722 skip list, 147
- 8723 slot, 203
- 8724 sorting
 - 8725 lower bound, 106
 - 8726 permutation, 7, 39, 330
- 8727 source text, 299
- 8728 space-time tradeoff, 32
- 8729 spatial locality, 339
- 8730 splay tree, 132, 163
- 8731 splitting line, 241
- 8732 spread-factor, 236
- 8733 stable sorting, 71, 110
- 8734 stack, 56
- 8735 static encoding, 318
- 8736 static scenario, 156
- 8737 strcmp, 186
- 8738 stream, 301
 - 8739 reset, 301
- 8740 string, 185
- 8741 substring, 187, 263
 - 8742 non-empty, 263
- 8743 successful search, 209
- 8744 successor, 118
- 8745 suffix, 263
- 8746 suffix array, 293
- 8747 suffix tree, 292
- 8748 super-linear run-time, 19
- 8749 tail-recursion, 103
- 8750 temporal locality, 157
- 8751 ternary tree, 187
- 8752 text compression, 299
- 8753 theta, 17
- 8754 tower, 148
- 8755 transfer
 - 8756 *a-b*-tree, 351
- 8757 transpose heuristic, 162
- 8758 treap, 144
- 8759 trie
 - 8760 compressed, 195
 - 8761 multi-way, 197
 - 8762 pruned, 192
- 8763 underflow, 351
- 8764 uniform hashing assumption, 206
- 8765 uniform probing assumption, 213
- 8766 universal hashing, 222
- 8767 unsuccessful search, 209
- 8768 variable-length encoding, 303
- 8769 variance, 377
- 8770 vector, 59
- 8771 word, 185
- 8772 worst-case run-time, 12