Iterative Design For Performant Code

January 2023

@ciura_victor

Victor Ciura
Senior SW Engineer
Visual C++

Iterative Design For Performant Code

“Software is getting slower more rapidly than hardware becomes faster.”

We often hear that our programs "should be efficient" and we should "squeeze every bit of performance" out of them. But what does that really mean? What's the difference between efficiency and performance? When does this really matter? Are these two goals ever in conflict with each other?

Let's spend some time deeply thinking about data structures and memory access patterns. Just a bit of theory and a whole lot of code analysis (C++). We'll get to build a simple autocomplete suggestion engine in 2 fundamentally different ways and figure out together the pros & cons of each.

Spoiler:
No AI/ML or any other fancy stuff: just "Algorithms + Data Structures = Programs".

Q & A

Ask questions as we go along...

STL Algorithms - Principles and Practice

"Prefer algorithm calls to hand-written loops"

Scott Meyers, "Effective STL"

Why prefer to use (STL) algorithms?

👉 Goal: No Raw Loops {}

Sean Parent - C++ Seasoning, 2013

Why prefer to use (STL) algorithms?

👉 Goal: No Raw Loops {}

Sean Parent - C++ Seasoning, 2013

Whenever you want to write a `for/while` loop:

```
for(int i = 0; i < v.size(); ++i) { … }
```

Why prefer to use (STL) algorithms?

👉 Goal: No Raw Loops {}

Sean Parent - C++ Seasoning, 2013

Whenever you want to write a `for/while` loop:

```
for(int i = 0; i < v.size(); ++i) { … }
```

Put the Mouse Down and

Step Away from the Keyboard !

Burk Hufnagel

Why prefer to use (STL) algorithms?

Correctness

Fewer opportunities to write bugs like:

- iterator invalidation
- copy/paste bugs
- iterator range bugs
- loop continuations or early loop breaks
- guaranteeing loop invariants
- issues with algorithm logic

Why prefer to use (STL) algorithms?

Code is a liability:

maintenance, people, knowledge, dependencies, sharing, etc.

More code => more bugs, more test units, more maintenance, more documentation

Why prefer to use (STL) algorithms?

Code Clarity

- Algorithm names say what they do
- Raw “for” loops don’t (without reading/understanding the whole body)
- We get to program at a higher level of abstraction by using well-known verbs (find, sort, remove, count, transform)
- A piece of code is read many more times than it’s modified
- Maintenance of a piece of code is greatly helped if all future programmers understand (with confidence) what that code does

Why prefer to use (STL) algorithms?

Simplicity

- Simpler code is more readable code
- Understandable and expressive
- Usually, shorter means simpler *(but not always)*
- Unsurprising code is more maintainable code
- Idioms are immediately recognized
- Code that moves complexity to abstractions (libraries) often has less bugs
- Compilers and libraries are often much better than you at optimizing
    - they're *guaranteed* to be better than someone who's not *measuring*

Why prefer to use (STL) algorithms?

What's the difference?

Performance / Efficiency

- Vendor implementations are highly **tuned** (most of the time)
- Avoid some unnecessary temporary copies (leverage **move** operations for objects)
- Function helpers and functors are **inlined** away (no abstraction penalty)
- Compiler optimizers can do a better job without worrying about pointer aliasing (auto-vectorization, auto-parallelization, loop unrolling, dependency checking, etc.)

The difference between Efficiency and Performance

Efficiency	Performance
the amount of work you need to do	how fast you can do that work
governed by your algorithm	governed by your data structures

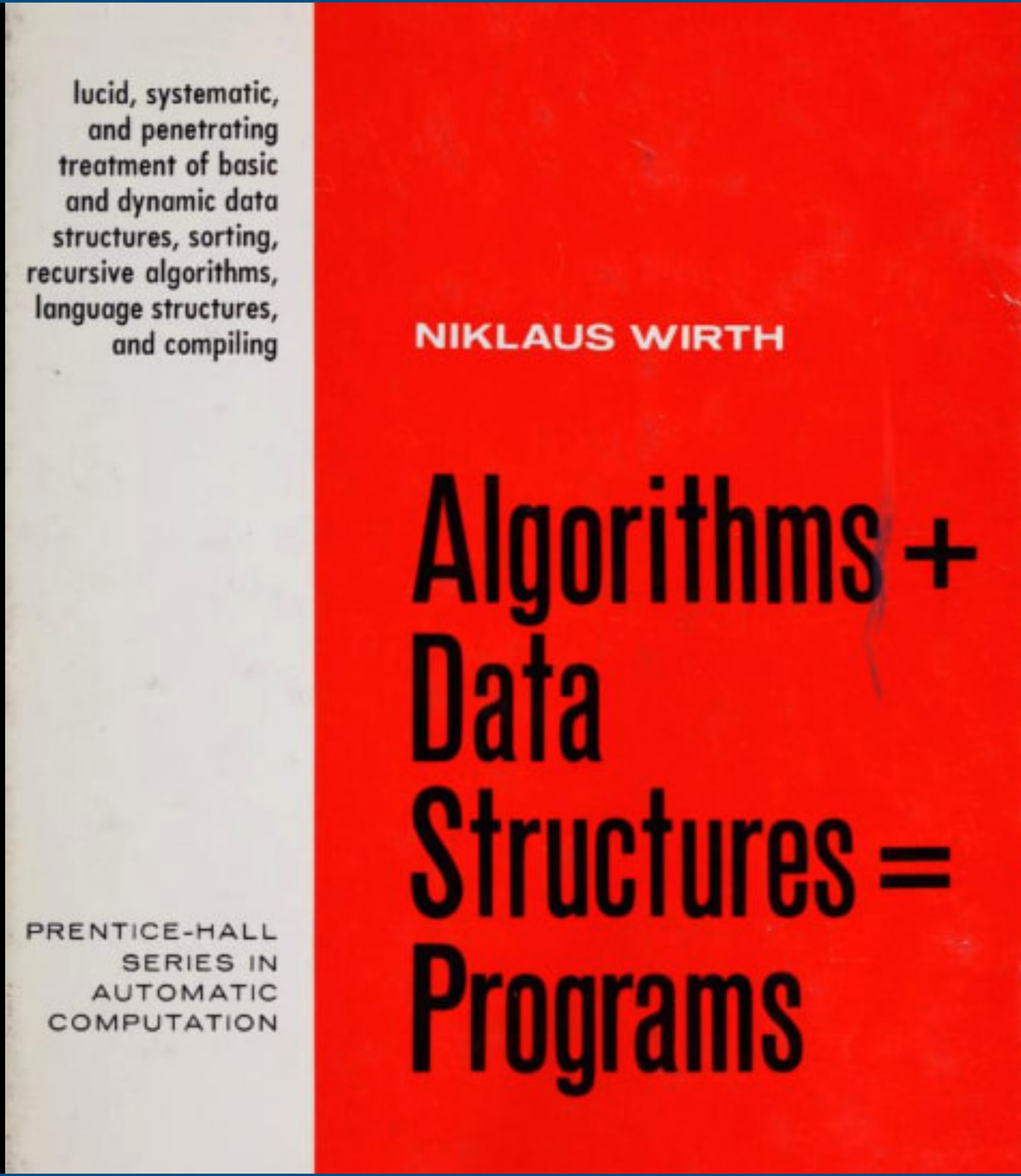
ℹ️ Efficiency and performance are not necessarily dependent on one another.

The difference between Efficiency and Performance

Why do we care ?

Because: *"Software is getting slower more rapidly than hardware becomes faster."*

"A Plea for Lean Software" - *Niklaus Wirth*



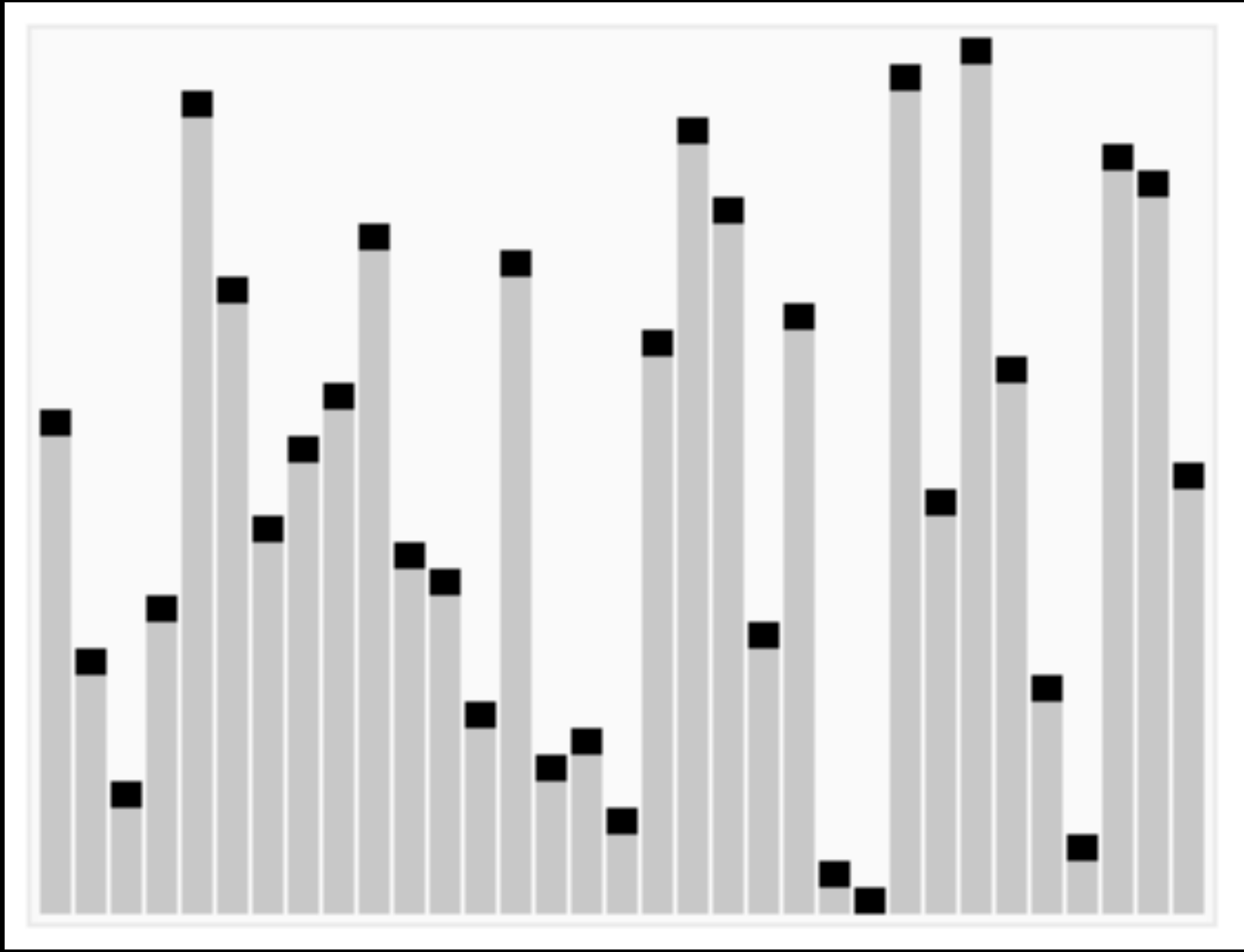
The Big-O

Algorithm	Data structure	Time complexity:Best	Time complexity:Average	Time complexity:Worst	Space complexity:Worst
Quick sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(n)$
Merge sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Heap sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Smooth sort	Array	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bogo sort	Array	$O(n)$	$O(n\, n!)$	$O(\infty)$	$O(1)$

wikipedia.org/wiki/Computational_complexity_theory

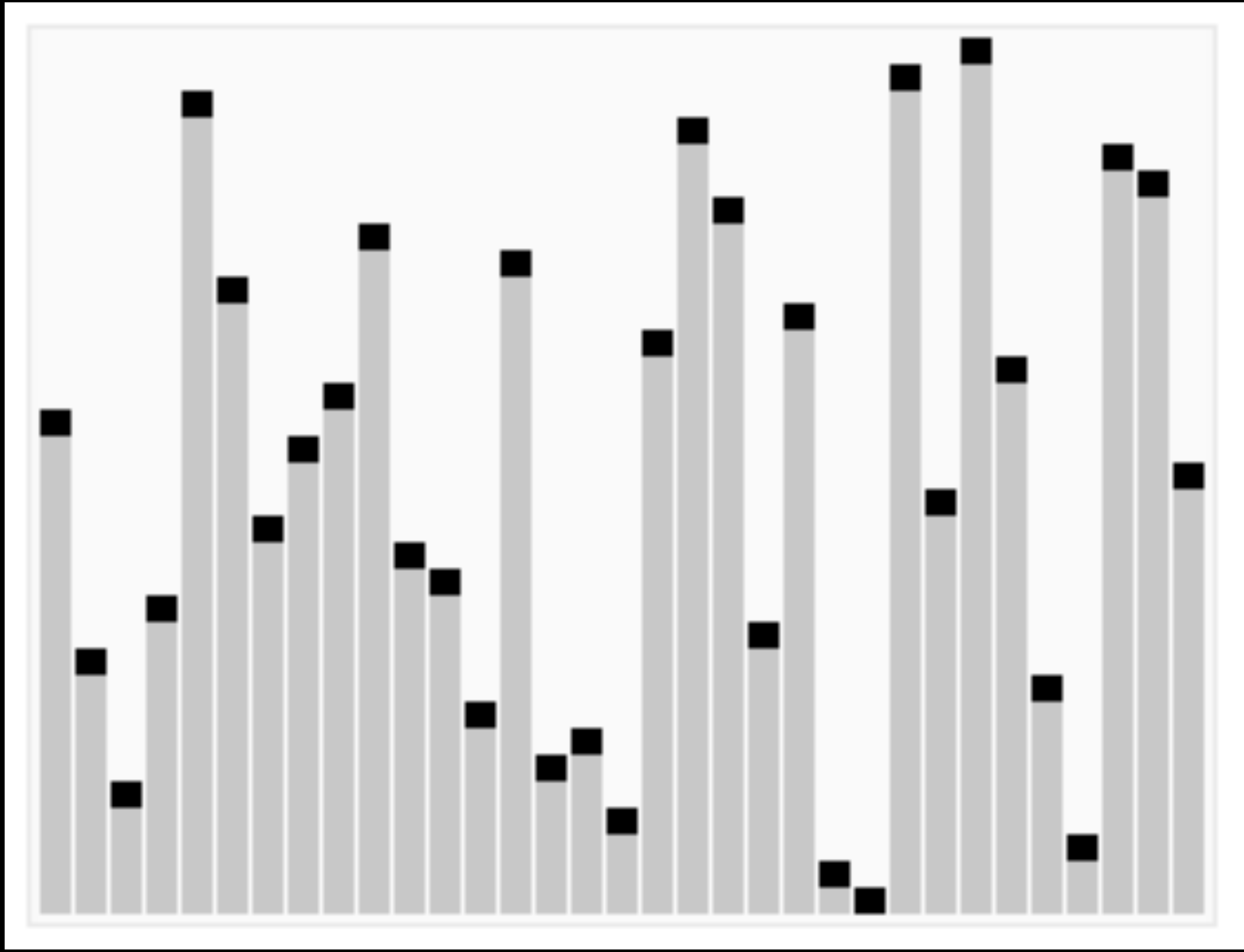
The Big-O

Recognize the algorithm?



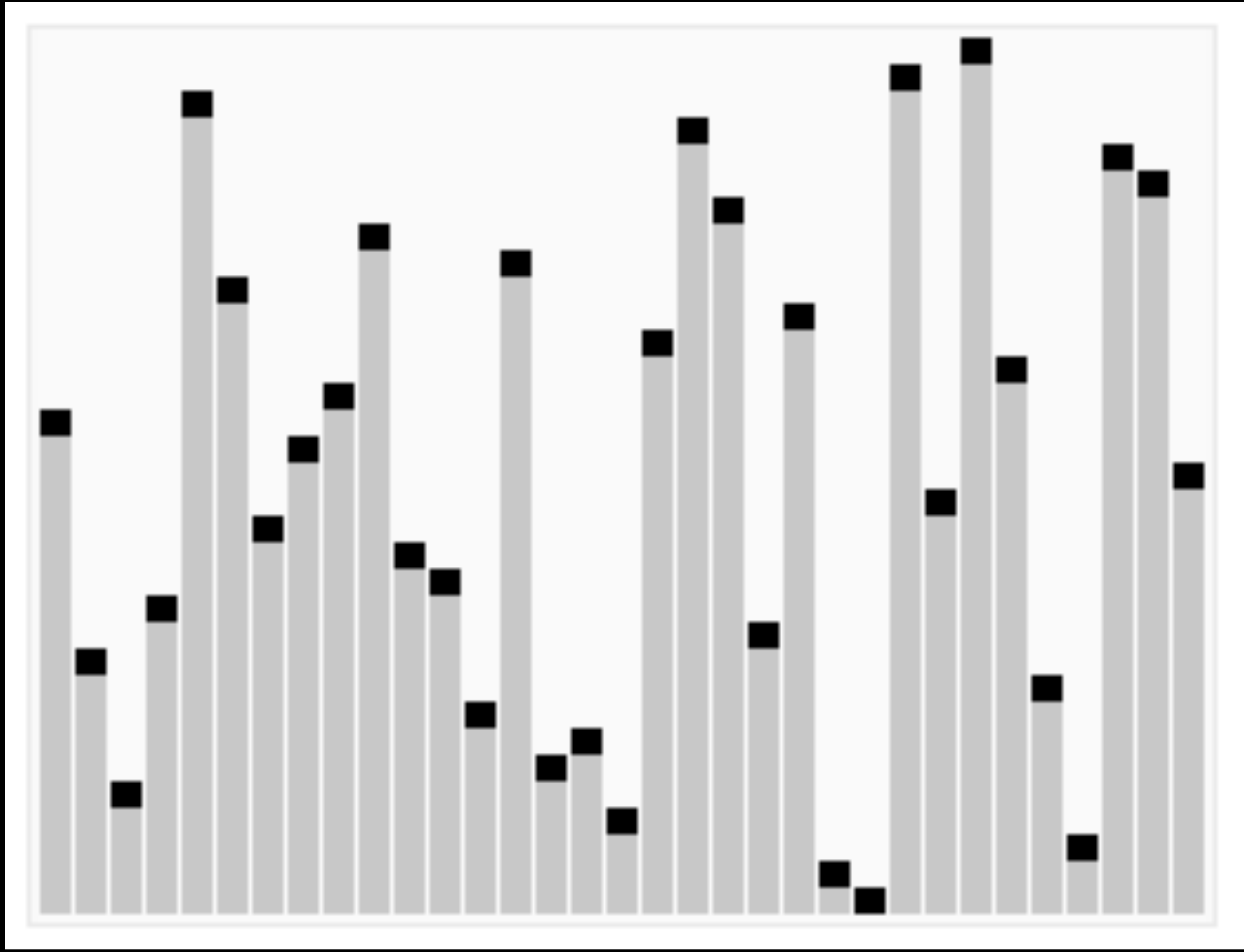
The Big-O

Recognize the algorithm?



The Big-O

Recognize the algorithm?



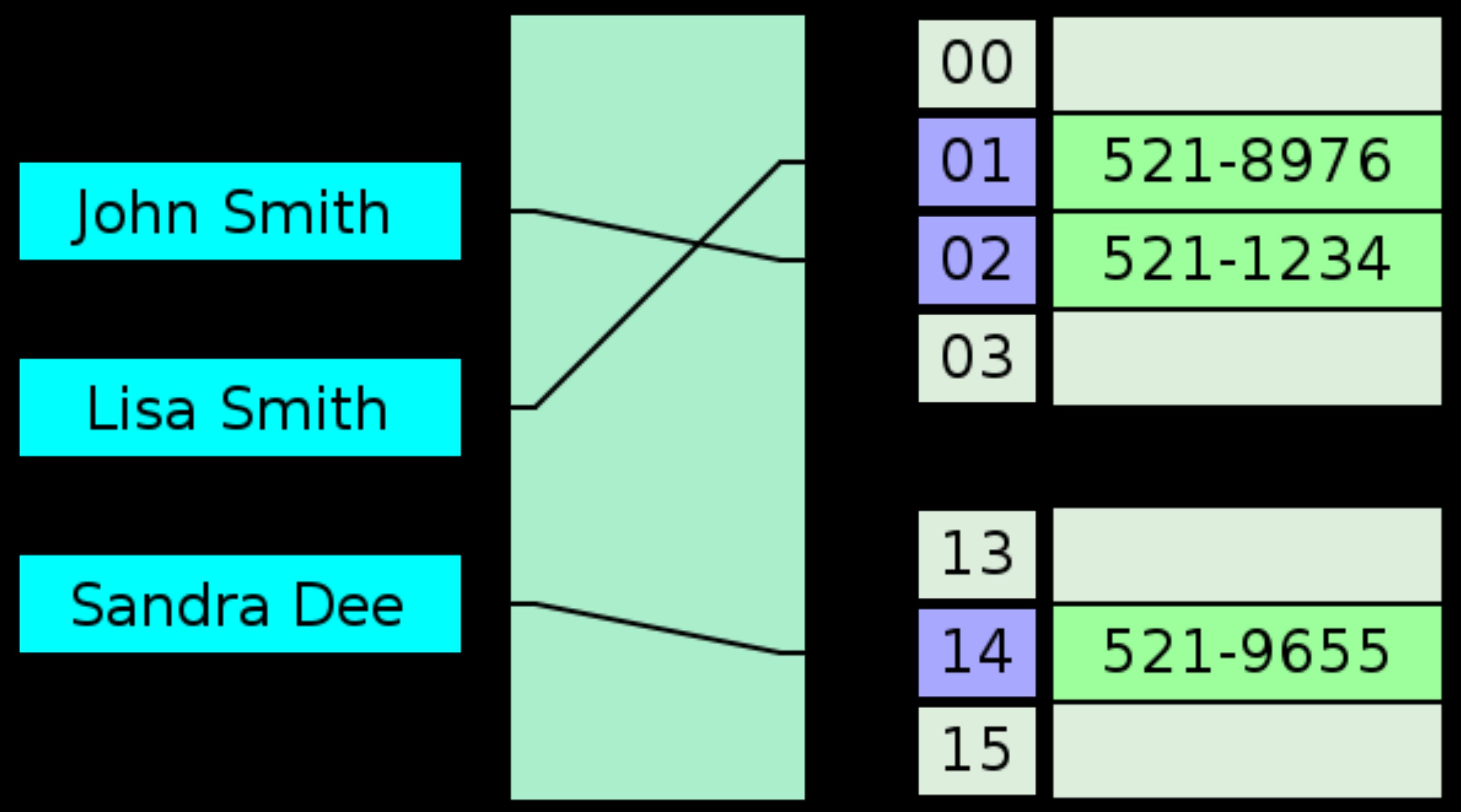
quicksort algorithm

has *average* case performance:

`O(n log n)`

What about Data Structures ? 📦

Data structures along with the operations they provide, also have complexity guarantees



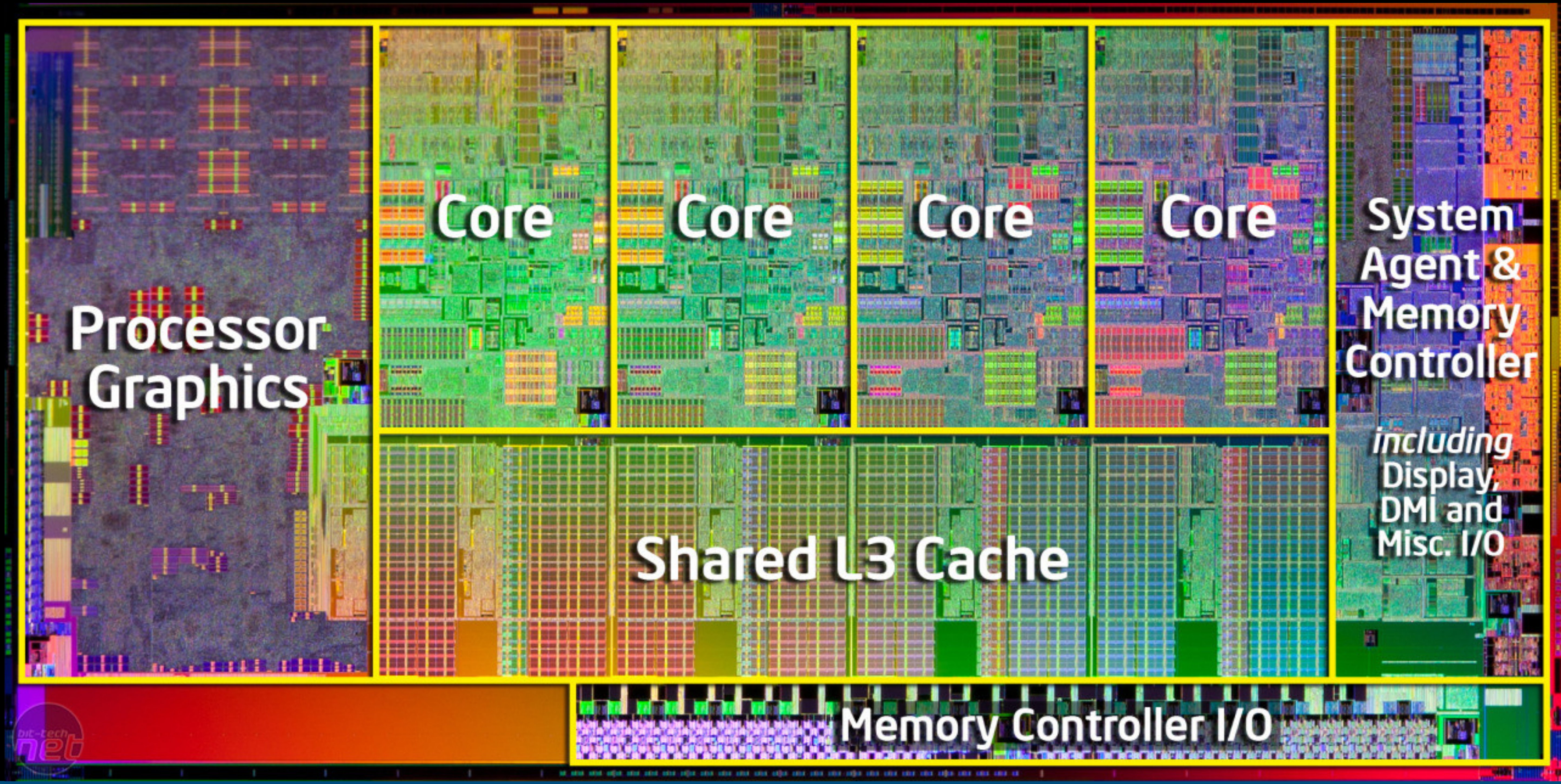
STL Containers Big-O cheat-sheet

	A	B	C	D	E	F	G	H	I
1	**C++ STL**	**insert @end**	**insert @pos**	**erase @end**	**erase @pos**	**find**	**sort**	**iterator**	**comment**
2	**vector**	O(1)	O(dist(pos,end))	O(1)	O(dist(pos,end))	O(n)	O(n*log(n))	RandomAccess	array
3	**dequeue**	@begin/@end O(1)	O(dist(pos,begin/end))	@begin/@end O(1)	O(dist(pos,begin/end))	O(n)	O(n*log(n))	RandomAccess	
4	**list**	O(1)	O(1)	O(1)	@pos O(1); @key O(n)	O(n)	O(n*log(n))	Bidirectional	doubly linked
5	**stack**	O(1) push()	-	O(1) pop()	-	O(n)	-	same as container	adaptor<dequeue, list, vector>
6	**queue**	O(1) push()	-	O(1) pop() @begin	-	O(n)	-	same as container	adaptor<dequeue, list>
7	**set/map**	-	O(log(n))	-	@pos O(1); @key O(log(n)+count(key))	O(log(n))	sorted	Bidirectional	red-black tree (balanced BST)
8	**unordered_set/ unordered_map**	-	avg O(1); worst O(n)	-	@pos avg O(1) worst O(n); @key O(count(key))	avg O(1); worst O(n)	-	Forward	hash_set/hash_map
9	**priority_queue**	push() O(log(n))	-	pop() O(log(n))	-	top() O(1)	-	RandomAccess	adaptor<vector, dequeue> => constant time extraction of the largest (default) element, at the expense of logarithmic insertion
10	**make_heap(range)**	push_heap() O(2*log(n))	-	pop_heap() O(2*log(n))	-	max is first	O(n*log(n))	RandomAccess	constructs a max heap in the range

What about Performance ? 🚀

How fast can the CPU execute each step from the algorithms.

This is mostly determined by the native (CPU) data types used and *your choice* of data structures.



Optimization

Strategy

- Identification: ***profile*** the application and identify the worst performing parts
- Comprehension: understand what the code is trying to achieve and why it is slow
- Iteration: change the code based on step 2 and then re-profile; repeat until fast enough

Don't trust your instinct !
Always benchmark the code changes.

Optimization

Very often, code becomes a bottleneck for one of four reasons:

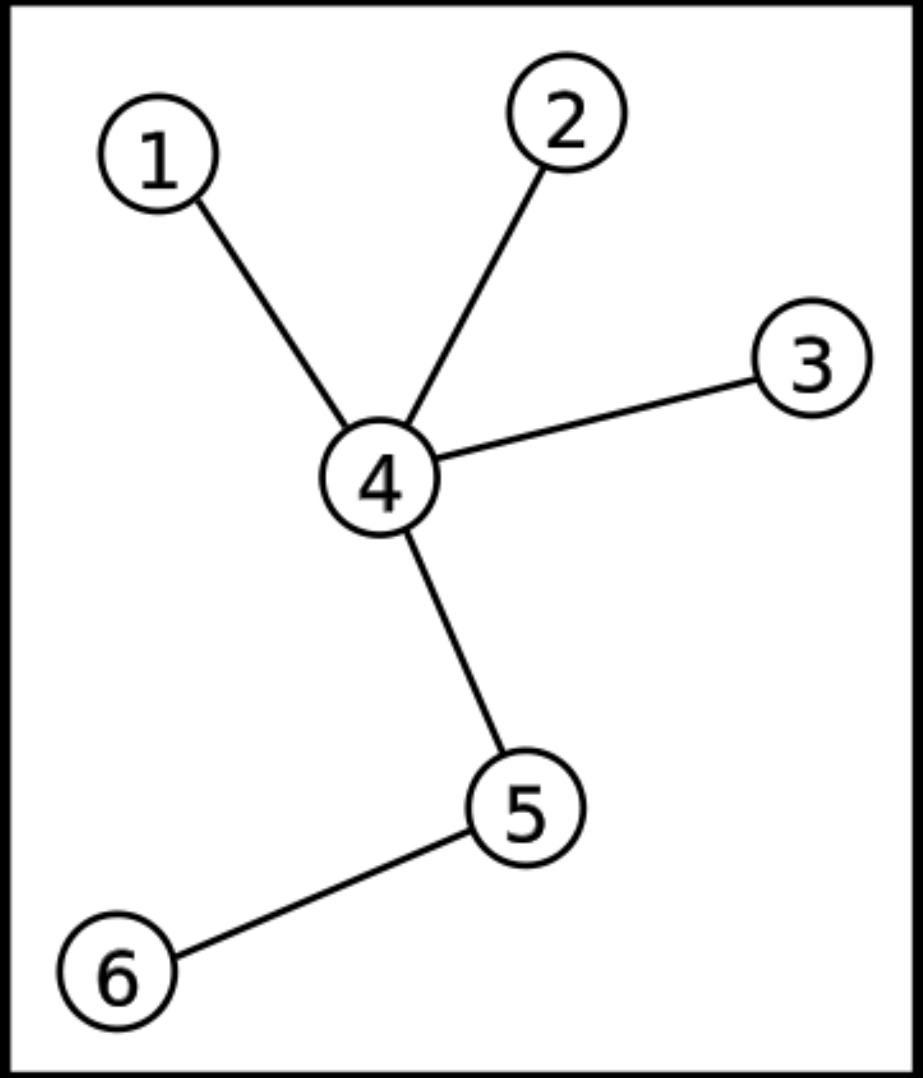
- It's being called too often
- It's a bad choice of algorithm: O(n^2) vs O(n), for example
- It's doing unnecessary work or it is doing necessary work too frequently
- The data is bad: either too much data or the layout and access patterns are bad

Focus

Today, let's focus on data structures

Because this is part of a course on graph algorithms,

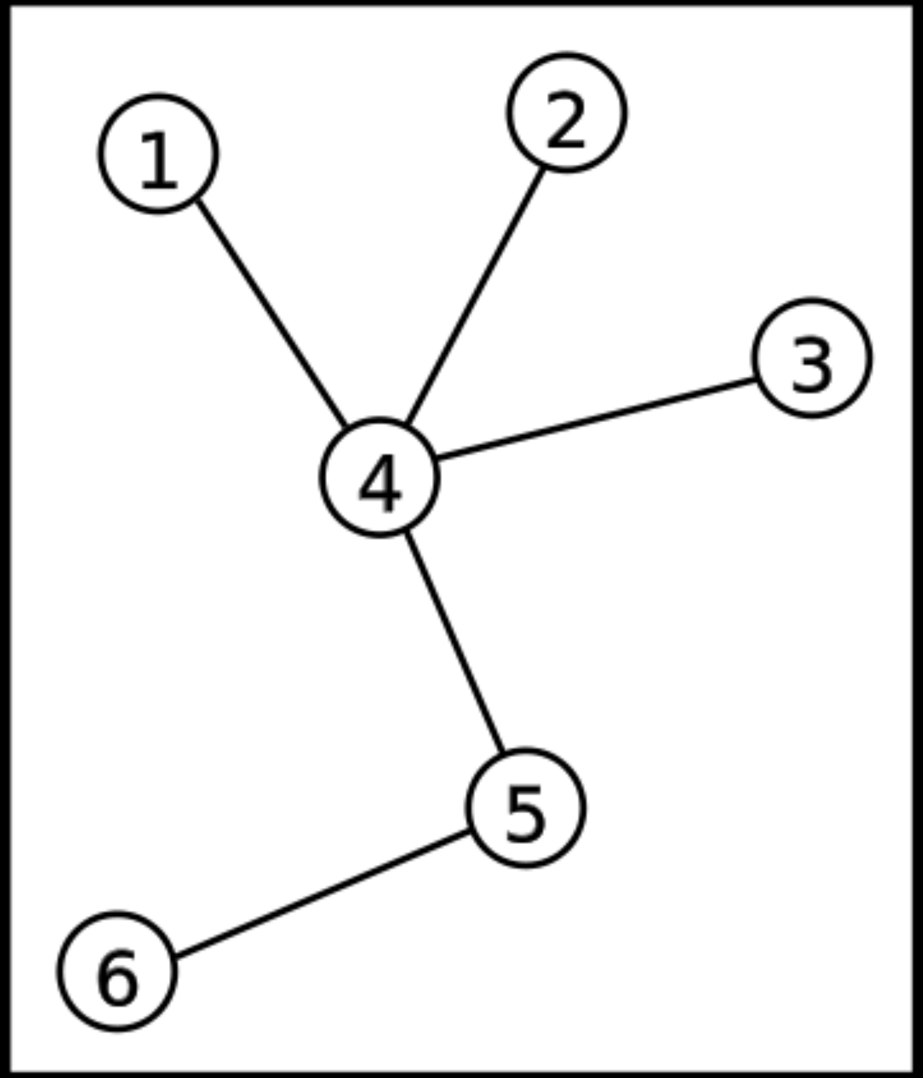
let's focus specifically on *node-based* data structures: graphs & trees.



In graph theory,

tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph.

forest is an undirected graph in which any two vertices are connected by at most one path, or equivalently an acyclic undirected graph, or equivalently a disjoint union of trees.



Tree data structures

Abstract data type that simulates a hierarchical tree structure,

with a root value and subtrees of children with a parent node,

represented as a set of linked nodes.

Trees

You probably already know a lot about trees, of different types,
each with individual specific properties and use cases in computer science.

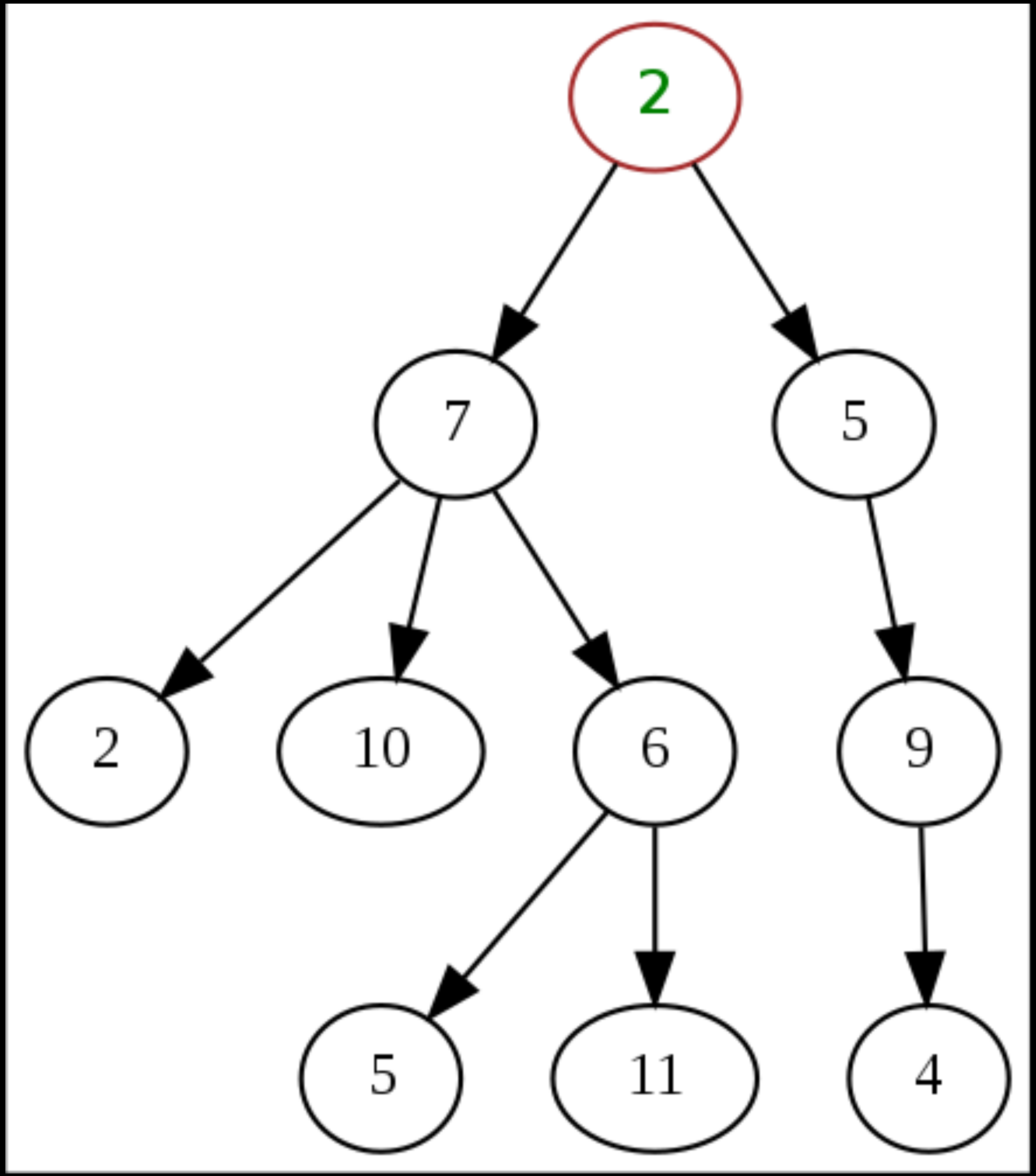
Trees

You probably already know a lot about trees, of different types,
each with individual specific properties and use cases in computer science.

But, I bet you don't know they look like this:

Trees

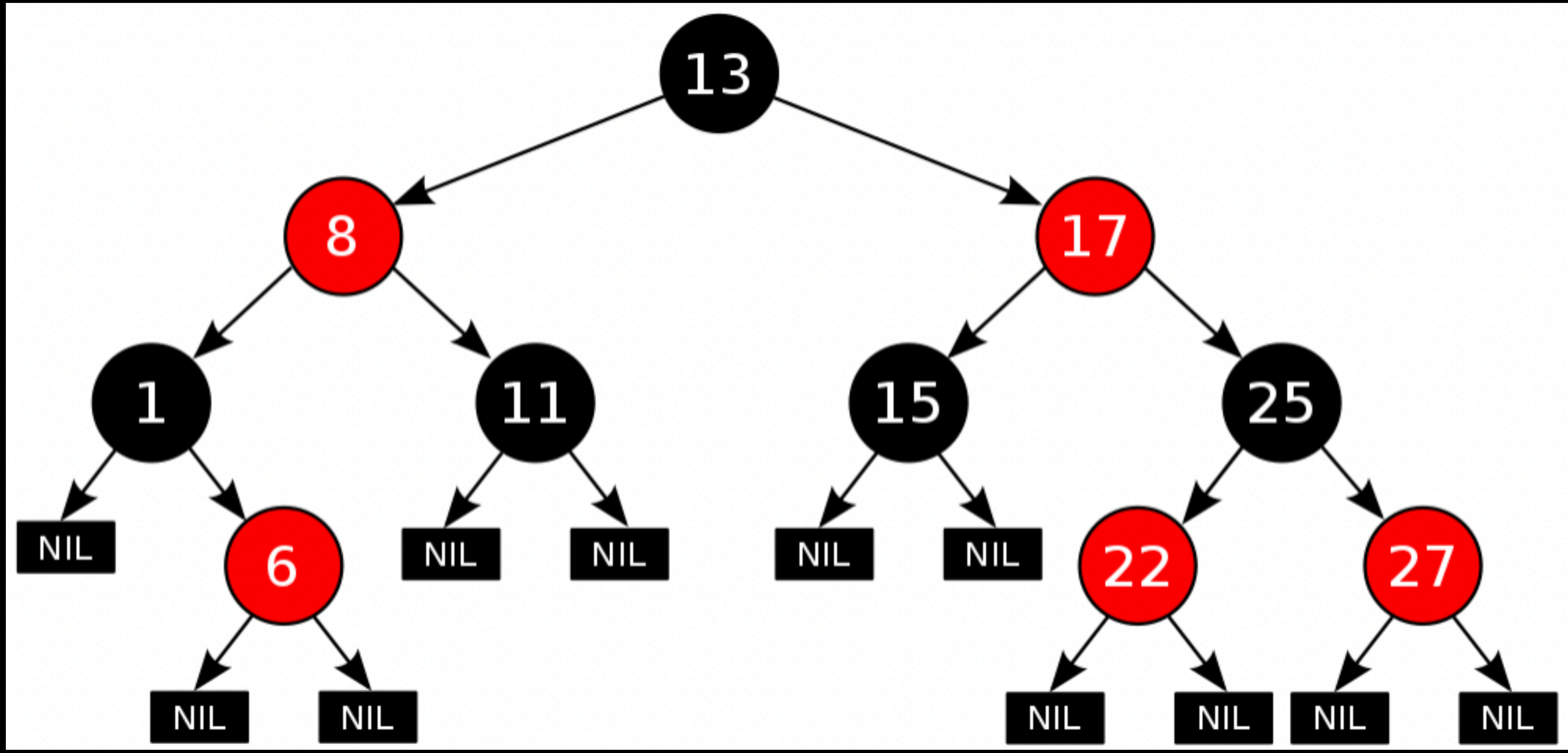
Trees



Red–black tree

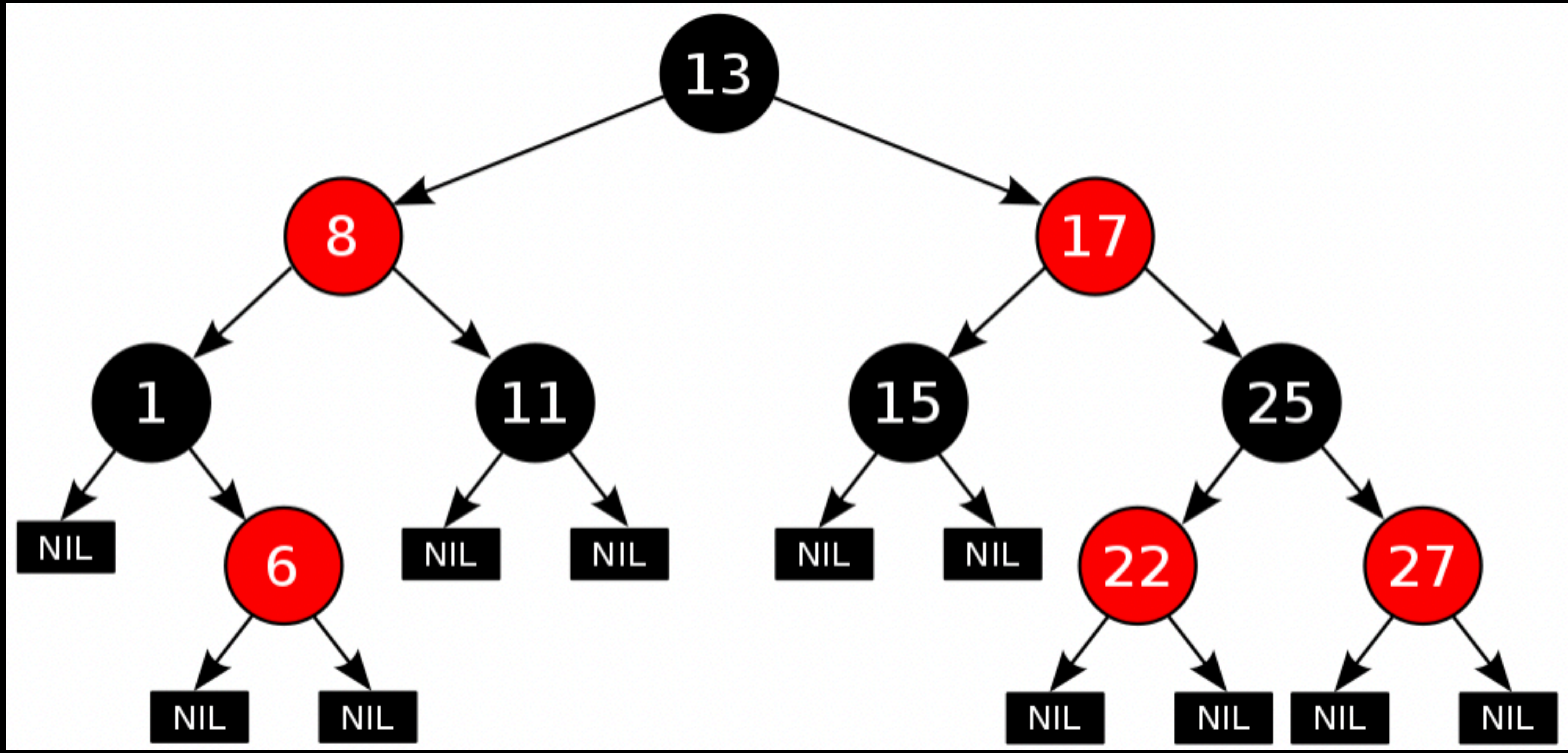
Self-balancing binary search tree

Each node stores an extra bit representing color (red/black), used to ensure that the tree remains *approximately* balanced during insertions and deletions.



Red-black tree

	Average/Worst
space	O(n)
lookup (search)	O(log n)
insert	O(log n)
delete	O(log n)

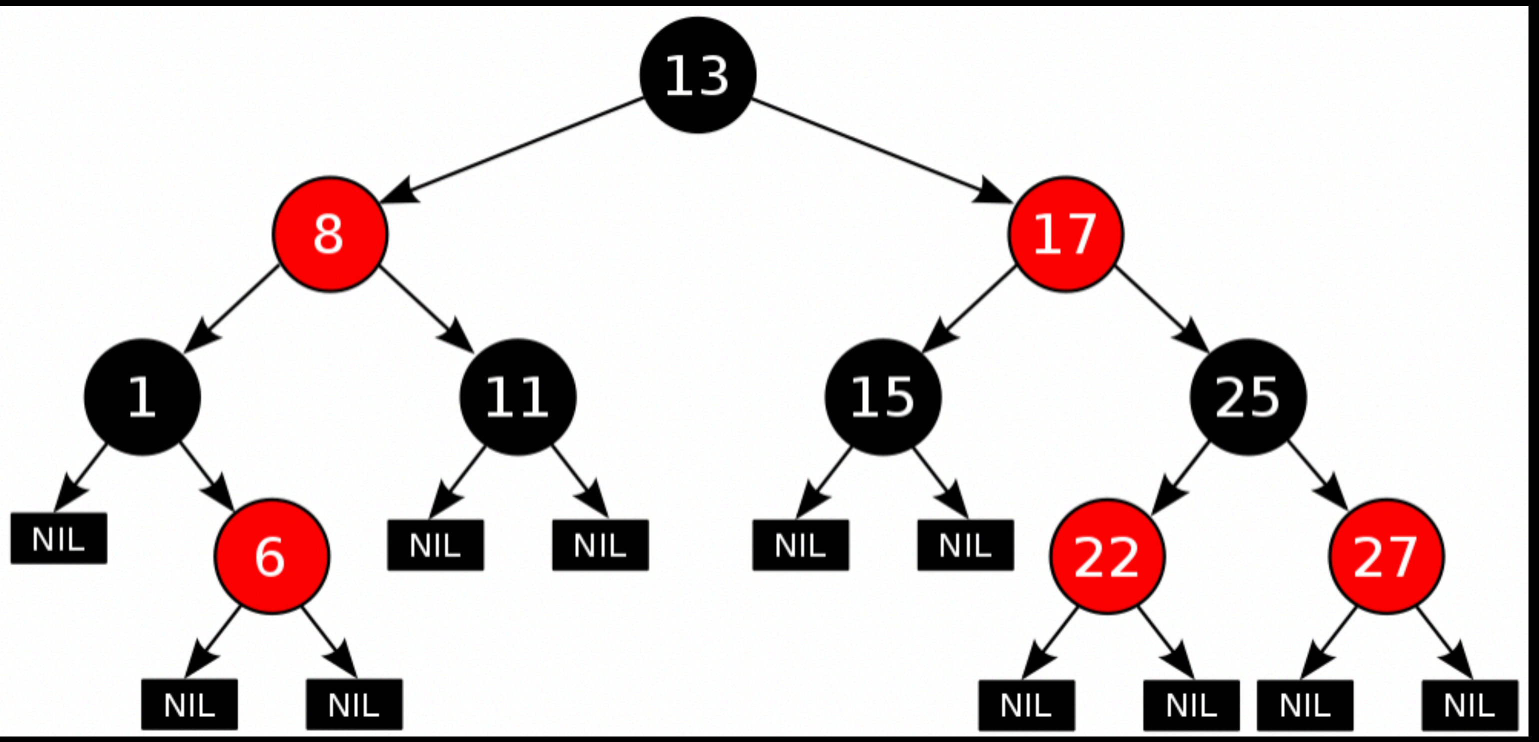


As opposed to other BSTs, the re-balancing is not perfect, but guarantees searching in O(log n) time

Red-black tree

Why am I narrowing to this special kind of binary search tree?

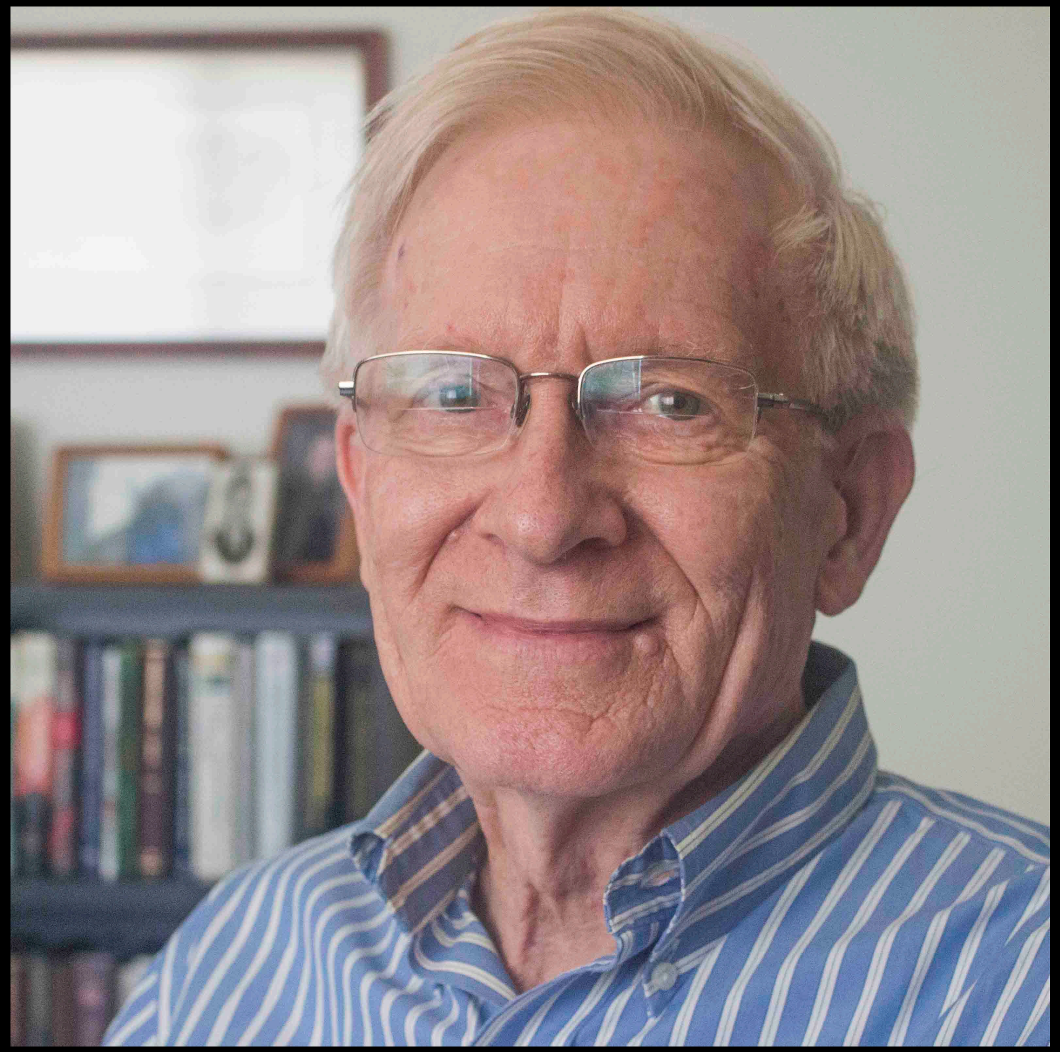
Because Alex Stepanov picked this kind of tree as the reference implementation for the API he designed for C++ STL associative data structures:
`std::map` & `std::set`

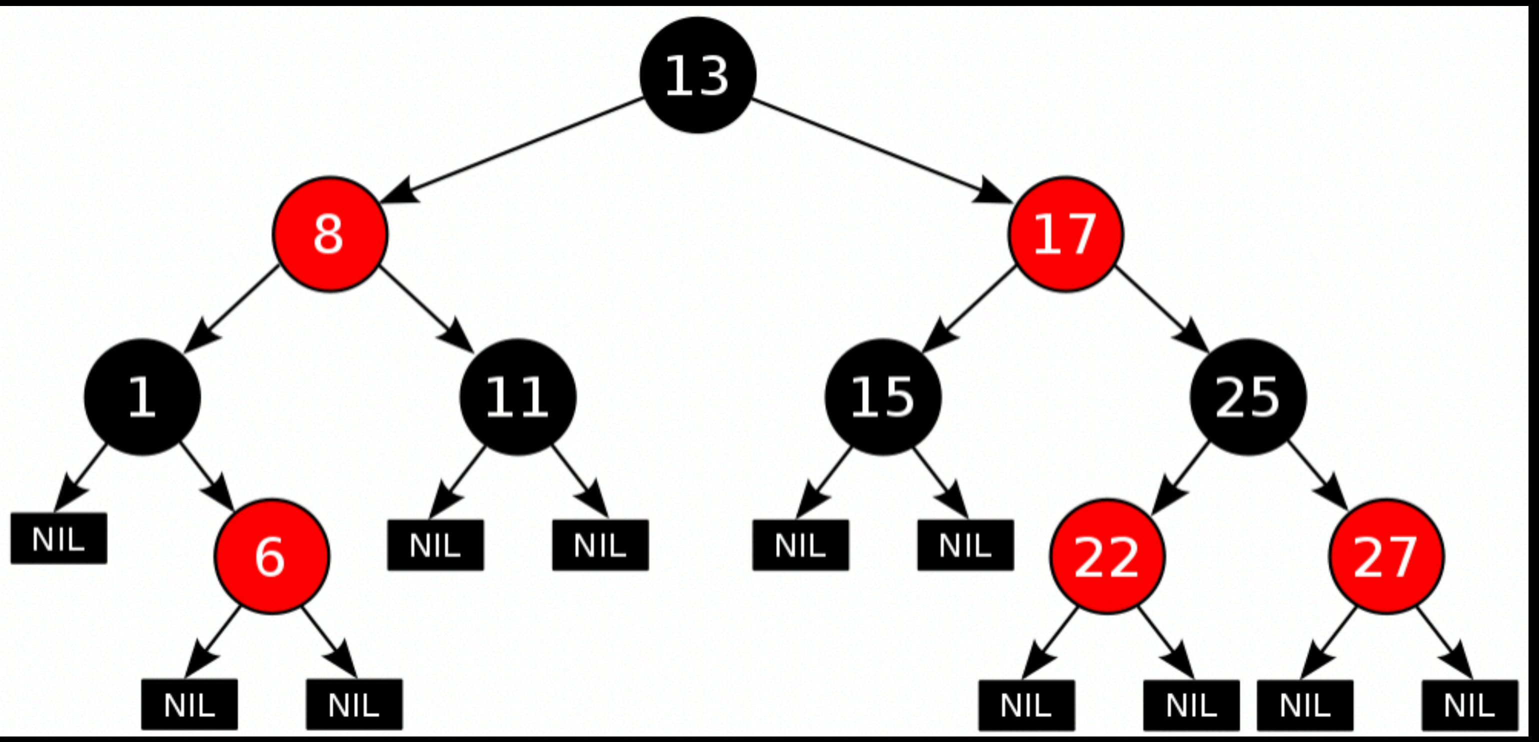


Red-black tree

Why am I narrowing to this special kind of binary search tree?

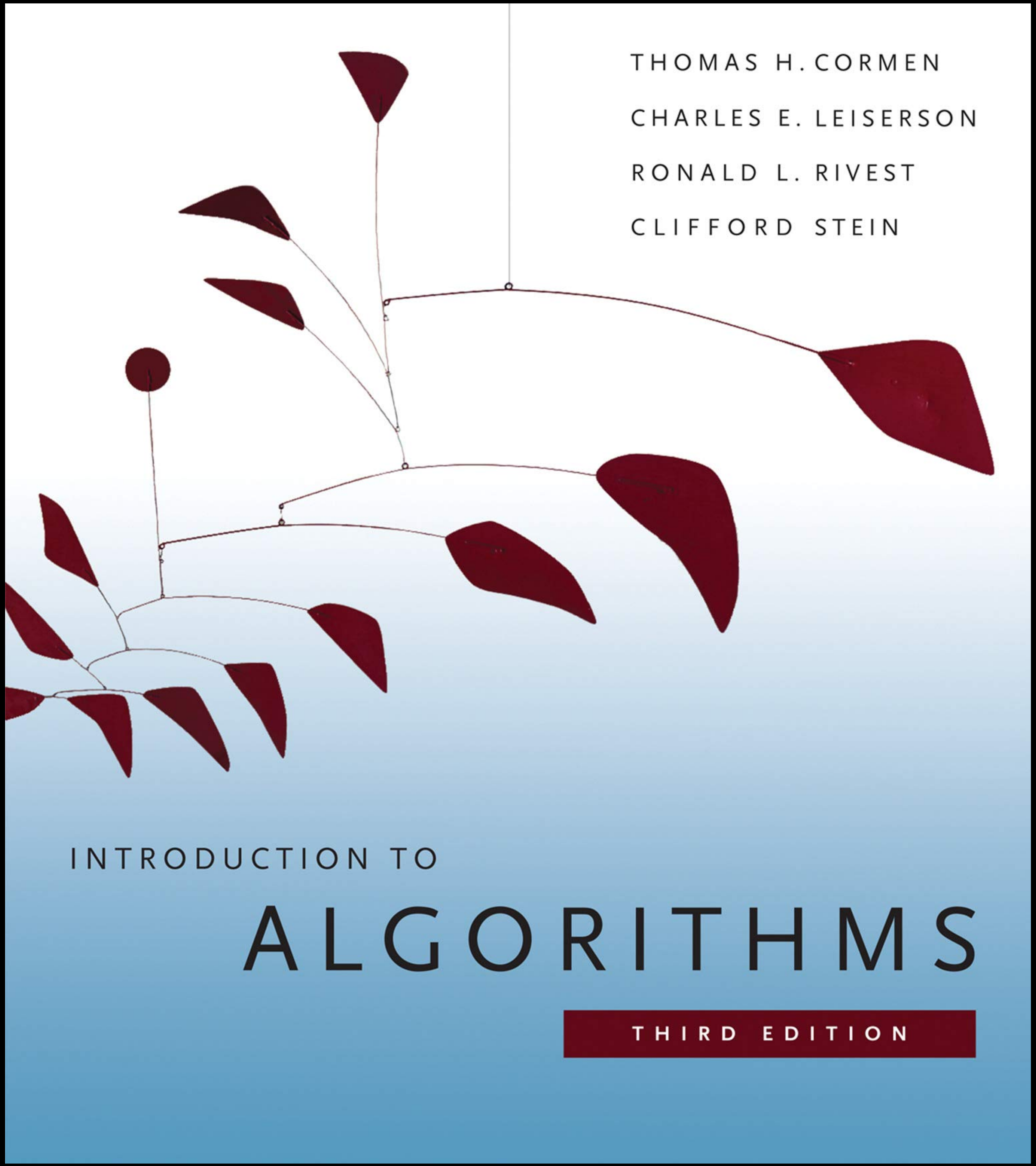
David Musser coded the best C++ *implementation* for the API Stepanov designed:
`std::map` & `std::set`





The Book

If you want to dig deep,
I highly recommend this classic:



Red-black tree

Red-black trees are very advanced data structures, that are beautifully wrapped in a very easy to use API:

`std::map` & `std::set`

Red-black tree

Red-black trees are very advanced data structures, that are beautifully wrapped in a very easy to use API:

`std::map` & `std::set`

... and this is where things get interesting 😈

Let's see!

Code dive

We'll explore together these properties, by building a search engine index in C++

... Really 😄

Let's see what we want to build.

Search engine index

Google Autocomplete

As you type in the browser search box, you can find information quickly by seeing search predictions that might be similar to the search terms you're typing.

The suggestions that Google offers all come from how people actually search.

For example, type in the word “cruise” and you get suggestions like:

```
Keyword: cruise

Suggested searches for: "cruise"
   -> cruise line
   -> cruise ship
   -> carnival cruise
   -> caribbean cruise
   -> princess cruise
   -> disney cruise
   -> celebrity cruise
   -> norwegian cruise
   -> alaska cruise
   -> ship cruise
```

Search engine index

```
Keyword: cruise

Suggested searches for: "cruise"
    -> cruise line
    -> cruise ship
    -> carnival cruise
    -> caribbean cruise
    -> princess cruise
    -> disney cruise
    -> celebrity cruise
    -> norwegian cruise
    -> alaska cruise
    -> ship cruise
```

These are all real searches that have been done by other people.

Popularity is a factor in what Google shows.
If lots of people who start typing in "cruise" then go on to type " line" that can help make "cruise line" appear as a suggestion in the future.

The task

We have a keywords "database" in the form of a large text file (keywords.db) containing search terms (phrases) used by people in the past.
(consider this an *active search cache*)

Here is a *small fragment* from this text file:

```
---------------- keywords.db -------------------
philips lcd 15
15 lcd cheap monitor
cheap 15 lcd monitor
dell e153fp 15 lcd midnight grey 36
lcd tv 15
samsung lcd 15
sony 15 lcd monitor
15 dvd lcd tv
15 inch lcd plasma monitors
...
```

Assumptions

You may assume the following simplifying preconditions:

- the text file contains only ASCII alphanumeric characters (English words)
- keywords are separated by space or CR/LF
- keywords database file is to be considered an immutable (read-only) snapshot of the current query cache
- each line in the file represents a search phrase used in the past
- consider the whole “database” as a continuous chain of keywords, separated by whitespace
- a keyword is a sequence of non-whitespace characters (words)

Search phrase

For simplicity, we shall define a search phrase as a pair of just two consecutive keywords in the query database.

E.g.

"cruise line"
"dell e153fp"
"cruise ship"
"samsung lcd"
"norwegian cruise"
"lcd cheap"
"sony 15"
"cheap monitor"

First task

First, we have to load and rank the keywords database.

That means ordering all search phrases according with their frequency in the cache (database).

We should be able to print the Top 1000 search phrases with their respective ranks (occurrence frequency).

First task

E.g.

Top 10 search phrases from keywords.db with their respective # ranks

```
real estate # 43298
for sale # 38022
new york # 27302
how to # 25068
web site # 21073
las vegas # 19039
cell phone # 17657
of the # 15012
credit card # 14278
web hosting # 11037
```

Second task

Our second task is to implement our own auto-suggestion engine for 10 related searches, based on top search phrases containing the input keyword.

See previous *example* with suggested searches for keyword: "cruise".

This operation should be super-fast.

* This interactive mode should be active only when the program receives a `/search` command-line switch.

The Code

We're going to see **2** completely different implementations for this program.

We're going to analyze the PROs & CONs of each and see some hints for a potential 3rd implementation => your homework assignment. 💻

Solution 1

Data Structures

Data structures used by the algorithm are designed to store the minimal amount of information in memory (no redundancy, no keyword copies).

Data structures leverage STL container iterators that are stable (valid) under the used algorithm operations.

We use node-based data structures (red-black trees): `std::set` & `std::map`

Solution 1

The Algorithm

Loading the keyword database into our data structures (counting search phrase occurrences).

=> filling a `std::map` from each phrase combination to its frequency

=> using std::set & std::map iterators everywhere, to avoid copying strings (keywords)

=> keywords are stored & referenced from a single location in an `std::set` (unique)

=> ranking is done automatically by means of a custom `std::set` comparator predicate

Solution 1

DEMO TIME

Let's dive into the code...

Solution 1 - analysis

PROs

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: `std::set` & `std::map`

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: `std::set` & `std::map`
- is idiomatic STL usage

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: `std::set` & `std::map`
- is idiomatic STL usage
- is type-safe and memory safe

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: `std::set` & `std::map`
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: `std::set` & `std::map`
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets
- stores unique keywords (no data duplication - space efficient)

Solution 1 - analysis

PROs

- is a very good showcase for STL usage (serves its didactical purpose)
- is succinct in implementation
- is relatively easy to explain/understand
- uses simple/ordered STL tree data structures: `std::set` & `std::map`
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets
- stores unique keywords (no data duplication - space efficient)
- offers good memory working set scaling for long search phrases

Solution 1 - analysis

CONs

Solution 1 - analysis

CONNs

- is not cache-friendly (uses tree/cell-based data structures spread all over memory)

Solution 1 - analysis

CONs

- is not cache-friendly (uses tree/cell-based data structures spread all over memory)
- tree data structures (sets/maps) are memory inefficient (a lot of waste in storing 64-bit pointers for tree nodes)

Solution 1 - analysis

CONs

- is not cache-friendly (uses tree/cell-based data structures spread all over memory)
- tree data structures (sets/maps) are memory inefficient (a lot of waste in storing 64-bit pointers for tree nodes)
- it uses the notoriously slow I/O streams for data input

Solution 1 - analysis

CONs

- is not cache-friendly (uses tree/cell-based data structures spread all over memory)
- tree data structures (sets/maps) are memory inefficient (a lot of waste in storing 64-bit pointers for tree nodes)
- it uses the notoriously slow I/O streams for data input
- for simplicity, our implementation uses case-sensitive compare for keywords

Solution 2

Data Structures

Are not designed to store the minimal amount of information in memory, having considerable redundancy in storing the keywords (allows for storing duplicate instances of keywords).

We use an STL `unordered_map` container to store all search phrases and their occurrences.

We store each keyword pair as a *concatenated* string "keyword1 keyword2" (map-first) with its corresponding counter (map-second).

This is where our data redundancy stems from (duplicated keywords from search pairs).

Solution 2

Data Structures

We chose this advanced data structure for our algorithm, because it is a hash map.
We leverage this fact for its speed in storing a new search phrase and finding an existing tuple to increment its frequency (in constant time).

Usage of the `CompareKeywordTupleCount` custom binary predicate is ***optional***, because it is not mandatory to perform a stable sort (*lexicographic*) with regards to search phrases (keyword pairs) that have the same rank/frequency.

Solution 2

The Algorithm

Loading the keyword database into our data structures (counting search phrase occurrences).

=> filling a `std::unordered_map` from each phrase combination to its frequency

"keyword1 keyword2" # 24

=> ranking keyword database using an auxiliary `std::vector` and applying `std::sort()` algorithm with a *custom predicate* (lexicographic stable sort, optional)

DEMO TIME

Let's dive into the code...

Solution 2 - analysis

PROs

Solution 2 - analysis

PROs

- is succinct in implementation

Solution 2 - analysis

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)

Solution 2 - analysis

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)
- is idiomatic STL usage

Solution 2 - analysis

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)
- is idiomatic STL usage
- is type-safe and memory safe

Solution 2 - analysis

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets

Solution 2 - analysis

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets
- it's very fast (due to hash-based lookup)

Solution 2 - analysis

PROs

- is succinct in implementation
- is relatively easy to explain (to someone who is familiar with hashed containers)
- is idiomatic STL usage
- is type-safe and memory safe
- offers good performance characteristics for large data sets
- it's very fast (due to hash-based lookup)
- although it **duplicates** data, its memory usage is lower than [Solution 1], because we have short keywords in our database and [Solution 1] has a lot of memory waste due to tree node 64-bit pointers

Solution 2 - analysis

CONs

Solution 2 - analysis

CONs

- it stores duplicated keywords (cannot help but feel uncomfortable about this ?!)

Solution 2 - analysis

CONs

- it stores duplicated keywords (cannot help but feel uncomfortable about this ?!)
- offers poor memory working set scaling for long search phrases

  (due to data duplication)

Solution 2 - analysis

CONs

- it stores duplicated keywords (cannot help but feel uncomfortable about this ?!)
- offers poor memory working set scaling for long search phrases

  (due to data duplication)
- for simplicity, our implementation uses case-sensitive compare for keywords

Solution 2 - analysis

CONs

- it stores duplicated keywords (cannot help but feel uncomfortable about this ?!)
- offers poor memory working set scaling for long search phrases

  (due to data duplication)
- for simplicity, our implementation uses case-sensitive compare for keywords
- it uses the notoriously slow I/O streams for data input

Solution 3 - Hints

Alternative solutions and further improvements:

- We could use a memory mapped file to map the keyword database directly into process memory, so that we could avoid using I/O streams and string parsing, processing
- We could perform a partial_sort of the keyword tuples (just Top N search phrases) and perform our lookup for suggestions in that pool

Solution 3 - Hints

Alternative solutions and further improvements:

- We could use a much more cache-friendly data structure, like an `std::vector` to store the tuple counts more compactly (array).
  - we would sort the array
  - count adjacent equal pairs
  - store counts and tuples in another array that we (partially) sort
  - read out the range desired

Solution 3 - Hints

Alternative solutions and further improvements:

- Because we are dealing strictly with English words, we could cut off (truncate) keywords at 8 bytes each and store them in a `uint64_t` integer.

  ⚠️ This is not functionally equivalent, but good enough because most keywords in the database are smaller than 8 characters.
  Using integers instead of strings would be a huge performance boost when performing comparisons and would also be much more space efficient.

General Techniques

- Graph theory
- Aggressive pruning input domain (restrict to realistic values in the natural workloads)
  - group classes of input values based of frequency of occurrence in the real-world
- Parallelize operations that can be split |> reduce
- Arrays FTW! (indexing more powerful than you'd think)
  - structs of arrays vs. array of structs (DoD - Mike Acton)
- Always think about alignment, padding and cache lines
- Choose a data structure based on the algorithm memory access patterns
- Replace high order logical op with equivalent bit level ops (encode bitfields if possible)

Solution++

Try using these hints to build an even better solution for our task

💻 **HAVE FUN !**