PERFORMANCE-ORIENTED COMPUTING

Optimization (2) – Data Structures

GOALS

- Understand which structures are favorable for which **types of operations**
  - Know which **decision criteria** there are, and how to apply them
- Gain an **overview** of **general-purpose data structures** from a performance perspective
- Distinguish between the **theoretical complexity** of data structure operations, and their **real-world performance**
- An outlook on **specialized** data structures

OPERATIONS & DECISION CRITERIA

INTRODUCTION

- Selecting the proper data structure(s) for a specific purpose is an **essential part of algorithmic optimization**
- Selecting a suitable data structure may:
  1. Allow performing common (in your application) operations at a **reduced complexity class** compared to a less suitable structure
  2. Improve the **memory layout** and/or allow the memory hierarchy of your target platform to work more effectively, significantly improving performance

HOW TO DETERMINE WHETHER A DATA STRUCTURE IS SUITABLE?

Several common decision criteria:

- **Type** and **quantity** of data which needs to be stored
- **Access patterns**
  - **Where** do we perform **which types** of operations?
- Target **hardware** properties

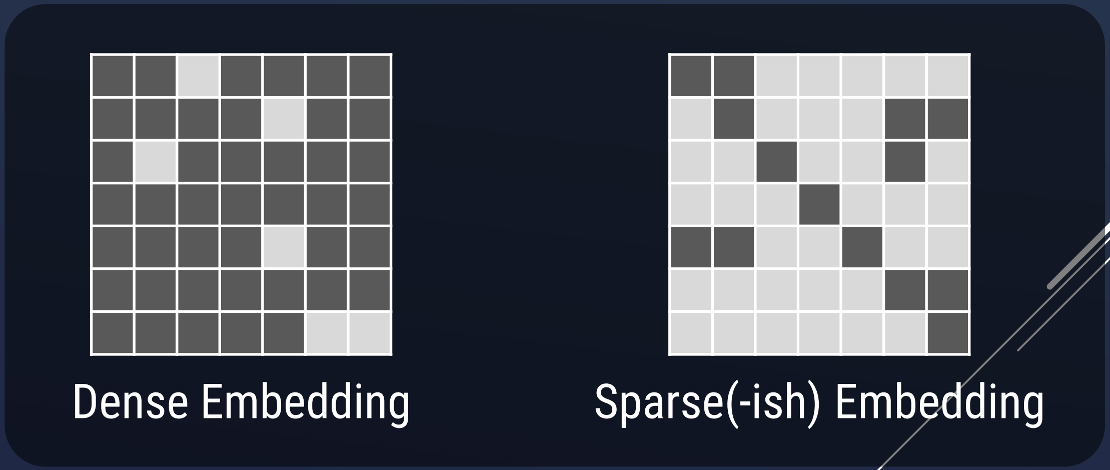
Generally **not as critical as the first two points**, but can be a **tie-breaker**.

TYPES OF DATA

Might be used internally to implement data structures which are not semantically trees (e.g. sets, maps)!

The data type can influence / limit the viable data structures:

- Are data elements **comparable**? Is there an ordering?
  - Required for e.g. many tree-based data structures
- Is a **hash** operation defined on the data elements?
- Are elements **countable**?
  - If so, is there a **dense embedding** in some iteration space?



QUANTITY OF DATA

Both **total quantity** and the **size of individual elements** influence our choice of data structures

- **Total quantity**:
  - **Low** total size → we might want to “waste” some space to save time
  - **High** number of elements → **complexity class** of operations becomes more important than constant overheads or HW suitability
- **Element size**:
  - **Small** individual elements need to be stored **densely** in memory

  → We’ll talk about a concrete example of this later in this lecture

ACCESS PATTERNS

We generally **don’t care** about patterns happening uncommonly, e.g. only during initialization of a long-running program.

- Adapting our data structure selection to the *common* **access patterns** of our application can have significant performance advantages

Categorize by:

- **Type** of access:
  - Read, write, insert, delete, replace, search, ...
- **Position** of access:
  - Front, back, sequential, arbitrary, ...
- **Parallel** access:
  - From how many threads? Multiple readers/writers/both?

ACCESS PATTERN EXAMPLE

We want to arbitrarily add elements, but always access the one with the highest *priority*

- First idea: need data structure which we can **efficiently sort**
- *Even better*: one which is **alway sorted**

➔ **Priority Queue**

→ Still open implementation choices, e.g. How to store elements

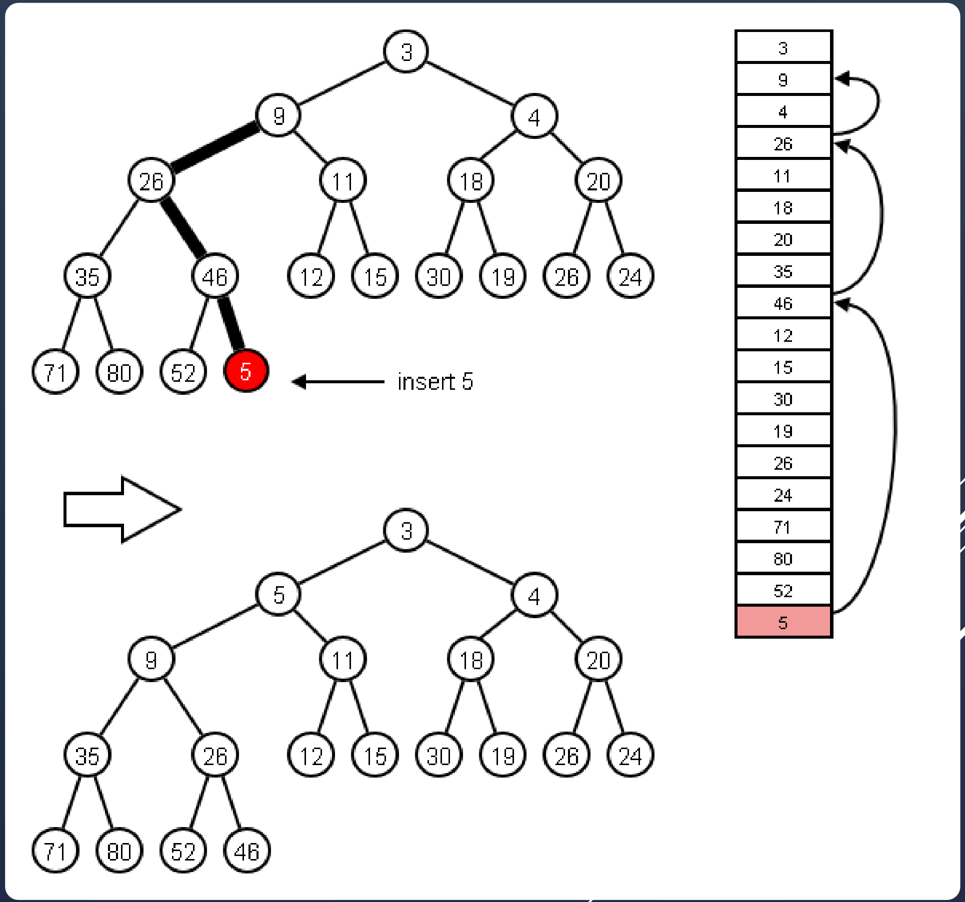


Image source: https://cs.lmu.edu/~ray/notes/pqueues/

TARGET HARDWARE CONSIDERATIONS

- **Additional factor** in the selection of data structures *after* algorithm considerations
- Importance depends on the **flexibility** of the target HW
  - i.e. we need more consideration towards HW when selecting a data structure for use on a GPU than a general-purpose CPU
- Most obvious constraint: **total memory**
  - Influences how much we need to focus on memory space efficiency

TARGET HARDWARE CONT.

Other relevant hardware aspects:

- **Caching / memory hierarchy**
  - Depending on the sophistication and importancy of the memory hierarchy, we might prefer data structures with fewer indirect accesses
- Impact of **branching**
  - Much more impactful on e.g. GPUs than high performance CPUs

May sound obvious / straightforward, but generally we don't make data structures more indirect or branchy just because it's fun.
Often a **tradeoff between operational/algorithmic efficiency and hardware efficiency**!

OVERVIEW OF SOME DATA STRUCTURES

And their performance properties

ARRAY-LIKE LISTS

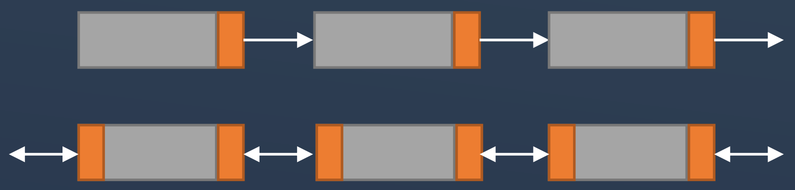
- Perhaps the most basic data structure
- ***Great for***
  - **Iterative traversal**
  - **Random** indexed **access**
  - Insertion at the end ...
    - ... as long as there is some extra space
- **Dense storage**, no extra metadata beyond size

  ➔ Very space-efficient, **great cache properties**

- ***Weaknesses***
  - Frequent arbitrary **insertion/deletion** (not at end)
  - **Search**
    - Can be solved by sorting (if only searching by one criterion)
  - **Pointer invalidation** / lack of reference stability

> Pointers/references to individual elements might be **invalid after operations** on the data structure (as elements were reallocated).
>
> Also known as **iterator invalidation** e.g. in C++.

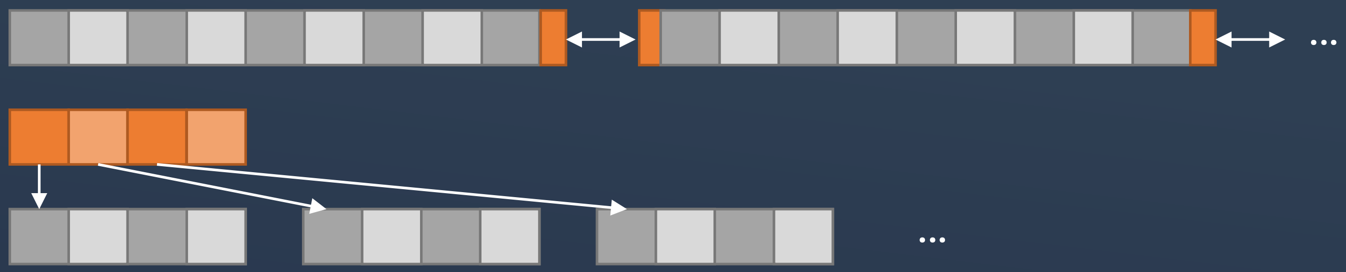
LINKED LISTS



- Either single- or double link, depending on use case
- ***Great for***
  - **Large** amounts of **data** (esp. large individual elements)
  - **Frequent** arbitrary **insertion/deletion**
- Provides **stable references**/pointers to elements

- ***Weaknesses***
  - **Not dense** in memory
    → slower traversal
    (and operations in general)
    due to caching and indirection
  - No random indexed access
  - Much less predictable for the compiler
    → lower optimization potential

The performance impact of these weaknesses is easy to underestimate!

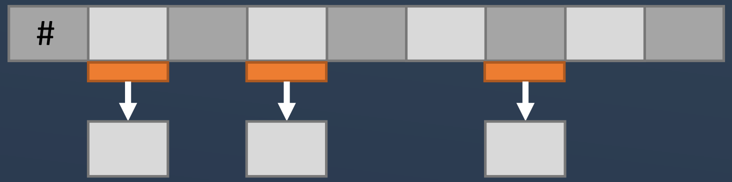


HYBRID LISTS

- Many data structures that seek to **combine** (some of!) the benefits of array-like and linked lists
- Examples:
  - Linked lists of fixed-size multi-element chunks
  - Array of pointers to fixed-size chunks
  - …
- Can provide e.g. faster arbitrary insertion/deletion than arrays with better cache behaviour than linked lists
  - But always a **tradeoff**

A real-world example of this is the C++ **`std::deque`** data structure.

HASHSETS/MAPS

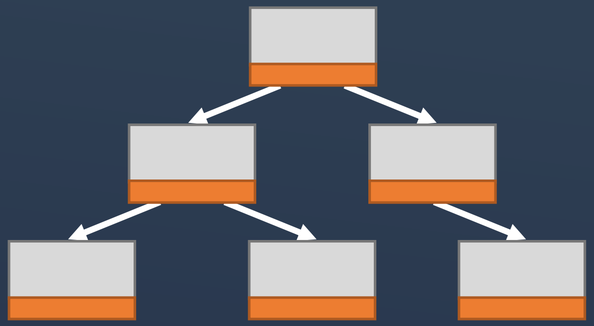


- Ideal for very **fast lookup** of specific elements
  - $O(1)$ when everything goes well
- **Requirement**: elements must be **efficiently hash-able**
- Also helpful, but not necessary: **prior knowledge** of roughly **how many** elements will be stored
  - Alleviates the need to change the number of buckets or re-hash

- Several performance-relevant details to hash out:
  - Open or closed addressing
  - Initial size
  - Tradeoff between collisions and hash function

➔ No change to fundamental properties / use cases

TREE STRUCTURES



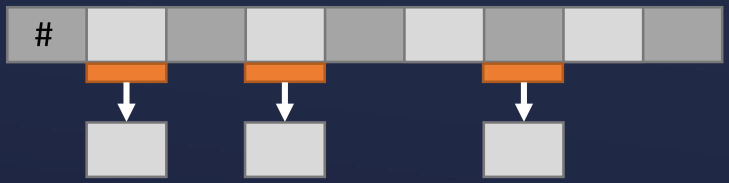
- Flexible for **many kinds of lookup / search purposes**
- Great when **sorted traversal** is frequently performed
- **Requires an ordering** on items

- **Disadvantage**: generally requires **balancing** operations in order to maintain performance after many insertions / deletions

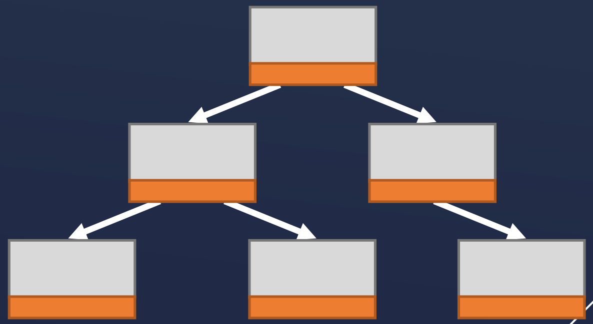
TREES VS HASHES

Both data structures are good for fast data retrieval. How to decide? – Comparison:

- Very fast **lookup**
- Potential for **performance degradation**
  Usually $\boldsymbol{O(1)}$, but can degrade to $\boldsymbol{O(n)}$
- Comparatively easier to implement correctly
- Tradeoff between memory efficiency and performance



- Very fast sorted **traversal** (since already sorted)
- **Predictable** performance
  $\boldsymbol{O(log(n))}$ for most operations
- Can be complex to implement (esp. balancing)
- Predictable memory utilization



RING BUFFERS

- Very **efficient insertion and removal** of elements at **start/end**
- With **pointer stability**!
- Fast **traversal**, indexed lookup
- Very well suited for transporting data in a **parallel producer/consumer** scenario
  - Only need to update 2 indexes/pointers, can be performed atomically

- ***Disadvantages / constraints:***
  - Maximum **number of entries** needs to be known **a priori**
  - **Fixed memory allocation** of all elements, regardless of actual use

Relatively small *number* of constraints, but they are very significant for many use cases.

FLYWEIGHT PATTERN

- Useful when there are a relatively **small number of distinct**, complex **elements**
  - But they are arranged in **complex relationships** with many individual instances (e.g. graphs)
  - Represent them by flyweight instances which refer to the actual elements

- Similar to using pointers, but can deal with instances as **values**
  - Less error prone memory management
  - Potentially better performance

BEYOND THIS SELECTION

- There are many more *relatively* general purpose data structures
  - Bitsets
  - Interval containers
  - Bimaps
  - …

➔ When there is a performance issue in your application that relates to operations on a container / data structure, determine the **decision criteria** we discussed (access patterns etc.) and then **search for a good fit**!

PERFORMANCE THEORY VS PRACTICE IN DATA STRUCTURES

PERFORMANCE THEORY

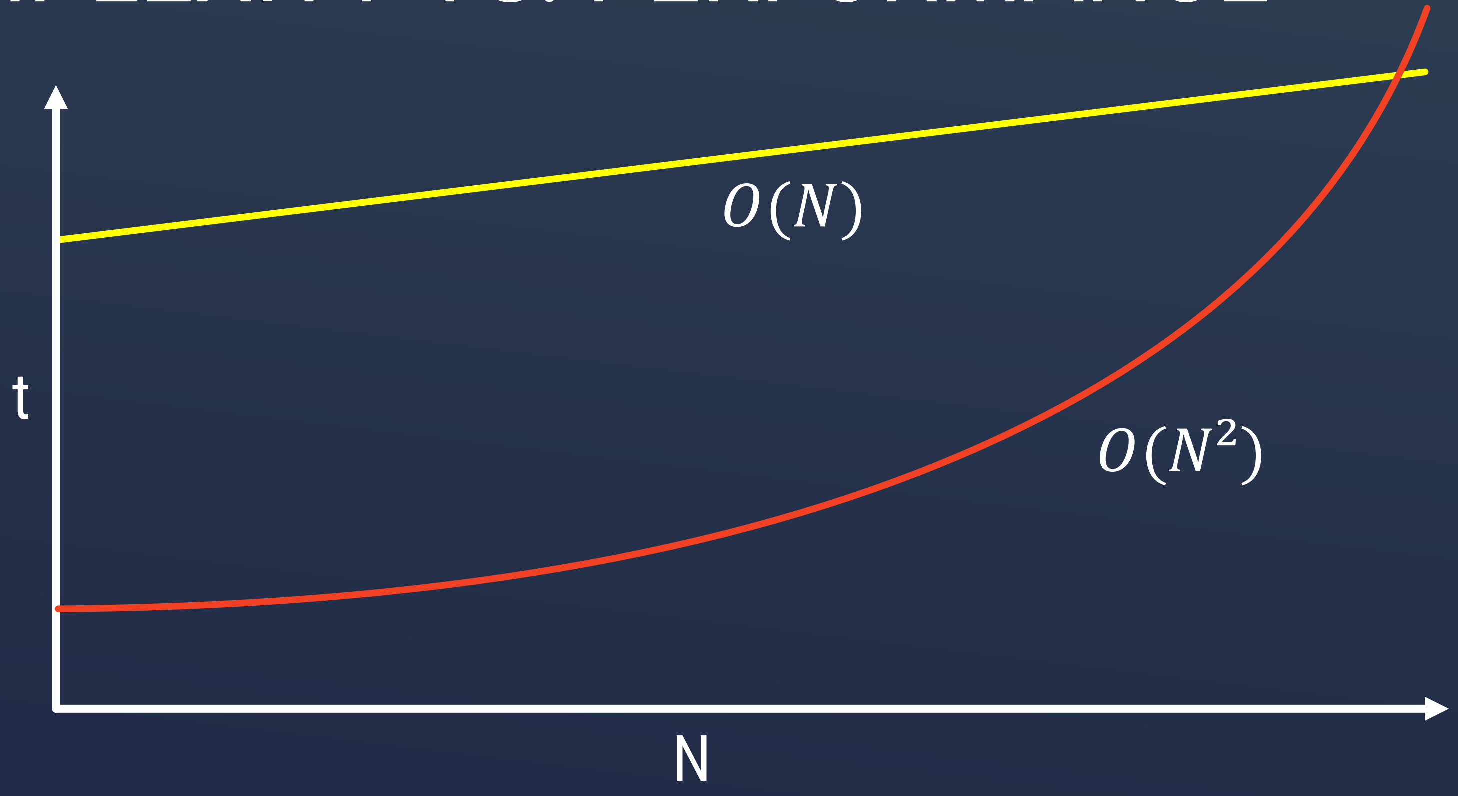
In **theory**, we generally express the expected performance of data structures by specifying the asymptotic bound of each potential operation using **big-O notation**

- Crucially, when we specify a complexity class, we **ignore constant factors**
  - This is often sensible and appropriate; but sometimes it isn't
- Also, it's important to distinguish between **average-case** and **worst-case** analyses

PERFORMANCE PRACTICE

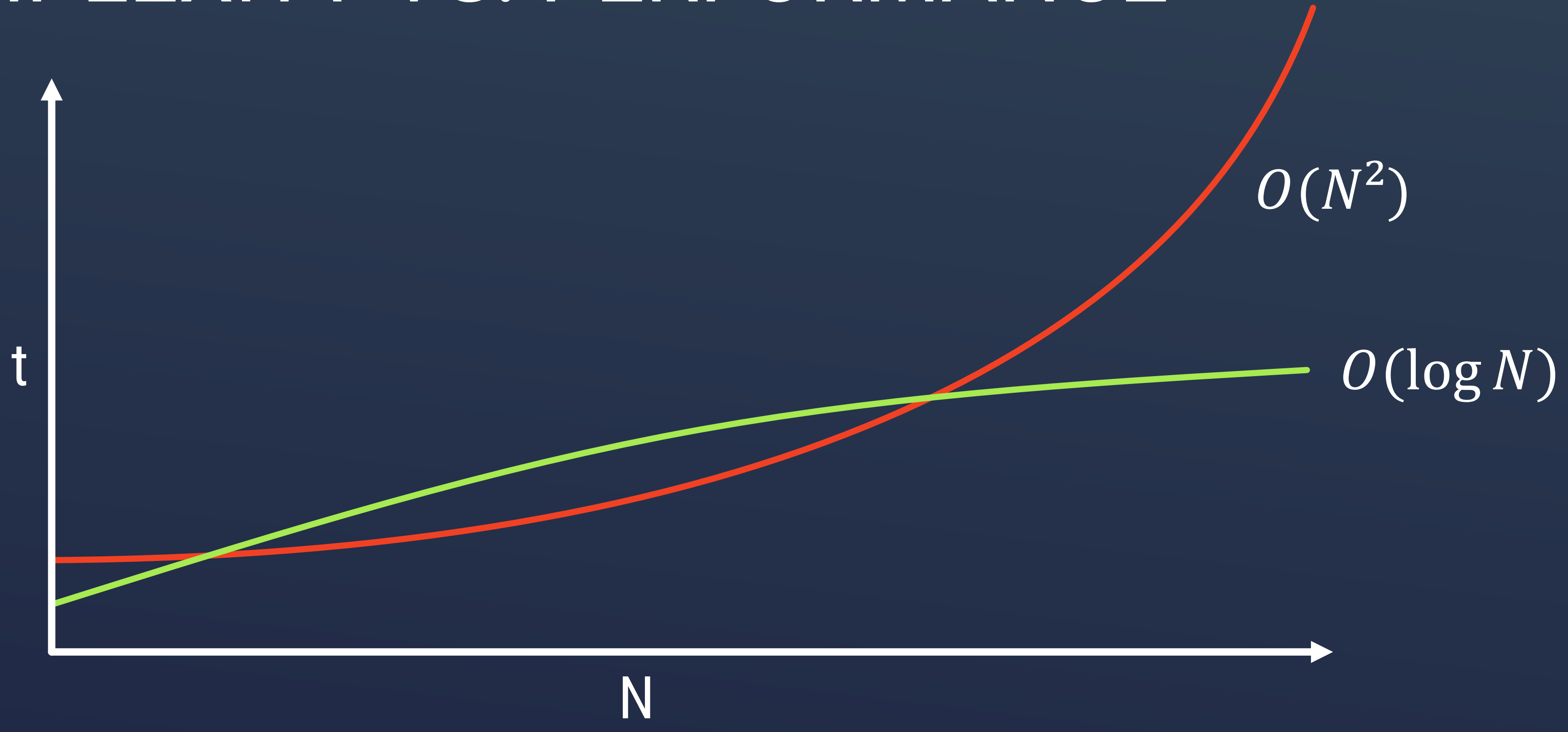
- In practice, **constant factors** – and even constant overheads – ***can*** make a large difference
  - And they are frequently **influenced by hardware factors**
- Rather obvious when the operations have the **same** complexity class
  - But depending on the degree of overhead and the expected N, *even differences in asymptotic complexity might not matter*
  - **Important**: decisions made based on this need to take into account the **expected evolution of N**, and might need to be revisited!

COMPLEXITY VS. PERFORMANCE



- *Need to be aware of:*
  how much overhead (if any) there is → **where** the transition happens

COMPLEXITY VS. PERFORMANCE



- *Need to be aware of:*
  where our program will be on the **N** scale

OPERATION MIX

- In real programs, your interaction with a given data structure is almost **never a single operation**
- Have to take into account **aggregate** performance of ***operation mix***
- Also important to consider distinct **phases** of a program
  - I.e. expensive building of a data structure justified if it happens **once** at the start and is then queried very **frequently**

THE DANGERS OF INTUITION

- Between **array-like** lists and **linked lists**, which data structure would you expect to perform best in the following cases?

Insertions/Deletions	Reads	Elem. Size	N
1%	99%	8 Byte	100
10%	90%	8 Byte	100
50%	50%	8 Byte	100
1%	99%	512 Byte	1000
10%	90%	512 Byte	1000
…			

➔ We'll explore the answer to this in the exercises!

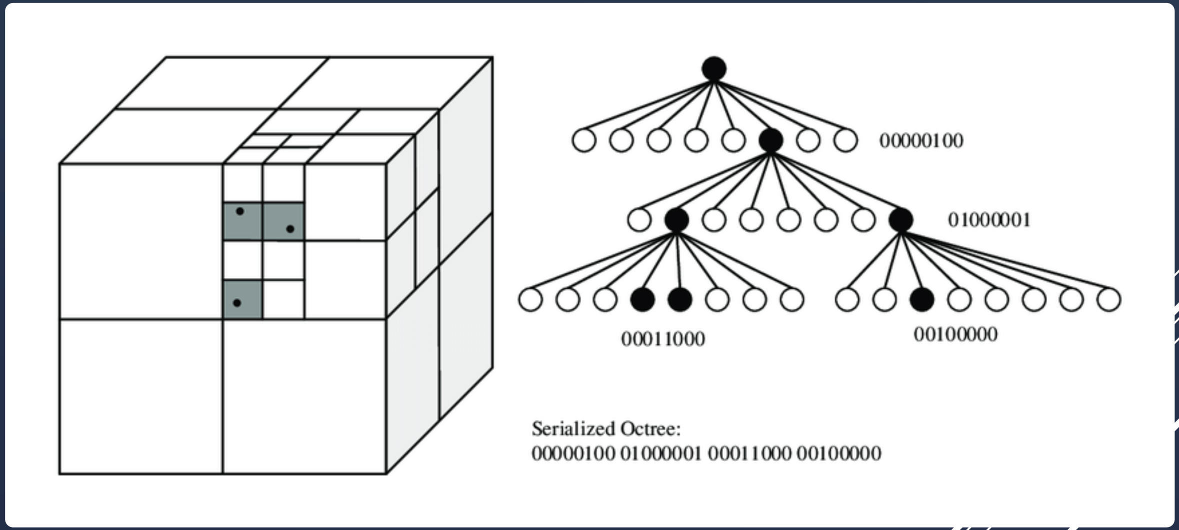
SPECIALIZED DATA STRUCTURES

WHAT DOES “SPECIALIZED” MEAN?

- Here, “specialized” means that these structures are built to either store one **particular type of data**, or support **particular kinds of operations**
- Especially in the context of a specific **application domain**
- Like many of these distinction, there is **no hard line** between “general” and “specialized”
  - E.g. interval containers or the flyweight pattern mentioned previously are clearly *more* specialized than a basic list, but still *less* specialized than some of the examples we’ll look at now

SPACE PARTITIONING STRUCTURES

- One example of a class of domain-specific data structures
  - Specialized for **queries in 3D space**
- Includes
  - **Octrees**
  - Kd-trees
  - Portal graphs
  - …



➔ When a domain is widely explored in a performance-focused context then there is often **a wealth of data structures** for the same (or very similar) purpose!

Image source: Joacim Dybedal

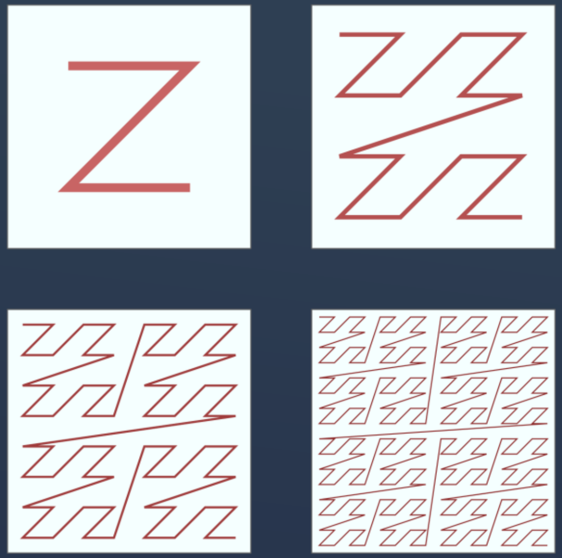
HARDWARE SUPPORT FOR SPECIFIC DATA FORMATS

- Sometimes, domain-specific data formats are sufficiently **critical in a domain** to reach widespread **hardware support**
- In such cases, it's almost always advantageous to make use of these formats
- Hardware implementation is generally:
  - **Faster**
  - More **efficient** (in terms of energy use)
  - Very likely to be **correct**

HARDWARE SUPPORT EXAMPLE
VIDEO AND AUDIO

- **Media files** are perhaps the most well-known data formats with widespread hardware acceleration
- E.g. all kinds of devices, from PCs over mobile phones to TVs support a variety of **audio and video codecs** in hardware
  - For several of them, allowing playback of files that wouldn't even be possible in software on the same platform
- In terms of performance, it might well be worth it to e.g. use a "worse" format with HW support rather than a "better" format in software
- Potential issues with **licensing**, but that's not a performance constraint
  - Unencumbered high-end formats now starting to become available in HW!

TEXTURES & COMPRESSION



- Textures on GPUs stored with **2D locality**
  - Uses a mapping to memory which follows a **space-filling curve**
    - Usually **Z-curve** due to efficient implementation
  - Highly **efficient** access, texture caching and filtering in HW

- Also HW support for various **block compression formats**
  - BC1 – BC7 (as named in D3D)
  - Also some less commonly supported formats like
    ASTC (Adaptive Scalable Texture Compression – primarily mobile)

CONCLUSION

SUMMARY

- **Operations & Decision Criteria**
  - Types of Data
  - Quantity of Data
  - Access Patterns
  - Hardware Considerations
- **Performance Theory vs Practice**
  - The impact of **constant factors** and overhead *before* the asymptotic limit
  - Operation Mix
  - The Dangers of Intuition
- **Overview of Data Structures**
  - Array-like Lists
  - Linked Lists
  - Hybrid Lists
  - Hashsets/maps
  - Tree structures
  - Trees vs. Hashes
  - Ring Buffers
  - Flyweight Pattern
  - Specialized Data Structures

QUESTIONS ?