Data

> Good programmers worry about data structures and their relationships.
>
> — Linus Torvalds

Control over data is essential to computer science: computational processes are made of data operations that transform input into output. But algorithms usually don't specify *how* their data operations are performed. For instance, `merge` (sec. 3.1) relies on unspecified external code to create lists of numbers, to check if lists are empty, and to append items into lists. The `queens` algorithm (sec. 3.4) does the same: it doesn't care how operations on the chessboard are made, nor how positions are stored in memory. These details are hidden behind what we call **abstractions**. In this chapter, we'll learn:

- ✨ How **abstract data types** keep your code clean,
- 🛠️ **Common abstractions** you need in your toolbox,
- 🏗️ Different ways to **structure data** in memory.

But before we dive into all this, let's first understand what the terms "abstraction" and "data type" mean.

Abstractions

Abstractions let us omit details; they are an interface for reaping the functionality of complex things in a simple way. For instance, cars hide complex mechanics beneath a driving panel, such that anyone can easily learn to drive without understanding any engineering.

In software, **procedural abstractions** hide complexities of a process beneath a procedure call. In the `trade` algorithm

(sec. 3.6), the `min` and `max` procedures hide *how* the minimum and maximum numbers are found, making the algorithm simpler. With abstractions on top of other abstractions, we can build modules[1] that allow us to do complex stuff with single procedures, like this:

```
html ← fetch_source("https://code.energy")
```

In one line of code, we fetched a website's source code, even though the inner workings of that task are extremely complex.[2]

Data abstractions will be a central topic in this chapter. They hide details of data-handling processes. But before we can understand how data abstraction works, we need to solidify our understanding of data types.

Data Type

We distinguish different types of fasteners (like screws, bolts, and nails) according to the operations we can perform on them (like screwing, wrenching, and hammering). Similarly, we distinguish different **types of data** according to the operations that can be performed on the data.

For instance, a data variable that can be split in positional characters, that can be converted to upper or lower case, that can receive appended characters, is of the String type. Strings represent texts. A data variable that can be inverted, that can receive `XOR`, `OR`, `AND` operations, is of the Boolean type. Booleans can be either `True` or `False`. Variables that can be summed, divided, subtracted, are of the Number type.

Every data type is associated with a specific set of procedures. The procedures that work on variables that store Lists are different to the ones that work on variables that store Sets, which are different from the ones that work on Numbers.

---

[1] A **module**, or **library**, is a piece of software that provides generic computational procedures. They can be included on demand in other pieces of software.

[2] It involves resolving a domain name, creating a TCP network socket, doing SSL encryption handshakes, and much more.

1.1 Abstract Data Types

An **Abstract Data Type** (ADT) is the specification of a group of operations that make sense for a given data type. They define an interface for working with variables holding data of a given type—hiding all details of how data is stored and operated in memory.

When our algorithms needs to operate on data, we don't directly instruct the computer's memory to read and write. We use external data-handling modules that provide procedures defined in ADTs.

For example, to operate with variables that store lists, we need: procedures for creating and deleting lists; procedures for accessing or removing the n^{th} item of a list; and a procedure for appending a new item to a list. The definitions of these procedures (their names and what they do) are a List ADT. We can work with lists by exclusively relying on these procedures. That way, we never manipulate the computer's memory directly.

Advantages of Using ADTs

Simplicity ADTs make our code simpler to understand and modify. By omitting details from data handling procedures, you focus on the big picture: the problem-solving process of the algorithm.

Flexibility There are different ways to structure data in memory, leading to different data-handling modules for a same data type. We should choose the best for the situation at hand. Modules implementing the same ADT provide the same procedures. This means we can change the way the data is stored and manipulated *just* by using a different data-handling module. It's like cars: electric cars and gas-powered cars all have the same driving interface. Anyone who can drive a car can effortlessly switch to any other.

Reusability We can use the same data-handling modules in projects that require handling data of the same type. For instance, both `power_set` and `recursive_power_set` from last chapter operate with variables representing sets. This means we can use the same `Set` module in both algorithms.

ORGANIZATION We usually need to operate several data types: numbers, text, geographical coordinates, images, and more. To better organize our code, we create distinct modules that each host code specific to a data type. That's called **separation of concerns**: parts of code that deal with the same logical aspect should be grouped in their own, separate module. When they're entangled with other functionalities, we call it *spaghetti code*.

CONVENIENCE We can get a data-handling module coded by someone else, and learn to use the procedures defined by its ADT. Then we can use these procedures to operate with variables of a new data type right away. Understanding how the data-handling module works isn't required.

BUG-FIXING If you're using a bug-free data-handling module, your code will be free of data-handling bugs. If you find a bug in a data-handling module, fixing it once means you instantly fix all parts of your code affected by the bug.

1.2 Common Abstractions

To solve a computational problem, it is very important to understand the type of data you're working on and the operations you'll need to perform on it. Deciding the ADT you'll use is equally important. Next, we present well known Abstract Data Types you should be familiar with. They appear in countless algorithms. They even come built-in with many programming languages.

Primitive Data Types

Primitive data types are those with built-in support in the programming language you're using—they work without external modules. These always include integers, floating points,[3] and generic operations with them (addition, subtraction, division). Most languages also come with built-in support for storing text, booleans and other simple data types in their variables.

[3] Floating points are a common way to represent numbers that have a decimal.

The Stack

Picture a pile of papers. You can put a sheet onto the top of the pile, or take the top sheet off. The first sheet to be added is always the last to be removed. The **Stack** is used when we have a pile of items, and only work with its top item. The item on top is *always* the pile's most recently inserted one. A Stack implementation must provide at least these two operations:

- **`push(e)`**: add an item `e` to the top of the stack,
- **`pop()`**: retrieve and remove the item on top of the stack.

More "advanced" stacks may provide more operations: to check whether the stack is empty, or to get the number of items currently in the stack.

Processing data this way is know as **LIFO** (**L**ast-**I**n, **F**irst-**O**ut); we only ever remove items from the top, which always has the stack's most recent insertion. The Stack is an important data type that occurs in many algorithms. For implementing the "undo" feature in your text editor, every edition you make is pushed onto a stack. Should you want to undo, the text editor pops an edition from the stack and reverts it.

To implement backtracking (sec. 3.4) without recursive algorithms, you must remember the sequence of choices that got you to the current spot in a stack. When exploring a new node, we push a reference to the node into a stack. To go back, simply `pop()` from the stack to get a reference of where to go back to.

The Queue

The **Queue** is the Stack's antagonist. It's also used for storing and retrieving items, but the retrieved item is always the one in *front* of the Queue, i.e., the one that has been on the queue the longest. Don't be confused, that's just like a real-life queue of people waiting in a restaurant! The Queue's essential operations are:

- **`enqueue(e)`**: add an item `e` to the back of the queue,
- **`dequeue()`**: remove the item at the front of the queue.

The Queue works by organizing data the **FIFO** way (**F**irst-**I**n, **F**irst-**O**ut), because the first (and oldest) item that was inserted in the queue is always the first to leave the queue.

Queues are used in many computing scenarios. If you are implementing an online pizza service, you will likely store the pizza orders in a queue. As a thought exercise, think about what would be different if your pizza restaurant was designed to serve the orders using a Stack instead of a Queue. 🤔

The Priority Queue

The **Priority Queue** is similar to the Queue, with the difference that enqueued items must have an assigned *priority*. People waiting for medical attention in a hospital is a real life example of a Priority Queue. The urgent cases receive top priority and go directly to the front of the queue, whereas the minor cases are added to the bottom of the queue. These are the Priority Queue's operations:

- **`enqueue(e, p)`**: add an item `e` to the queue according to the priority level `p`,
- **`dequeue()`**: remove the item at the front of the queue and return it.

In a computer there are typically many running processes but only one (or a few) CPUs to execute them. An operating system organizes all these processes waiting for execution in a Priority Queue. Each process waiting in the queue is assigned a priority level. The operating system dequeues a process and lets it run for a little while. Afterwards, if the process isn't finished it gets enqueued again. The operating system keeps repeating this.

Some processes are more time-sensitive and get immediate CPU time, others wait in the queue longer. The process that gets input from the keyboard typically receives a super-high priority. If the keyboard stops responding, the user might believe the computer crashed and try to cold-restart it, which is *never* good.

The List

When storing a bunch of items, you sometimes need more flexibility. For instance, you could want to freely reorder the items; or to access, insert and remove items at any position. In these cases, the **List** is handy. Commonly defined operations in a List ADT include:

- **`insert(n, e)`**: insert the item `e` at position `n`,
- **`remove(n)`**: remove the item at position `n`,
- **`get(n)`**: get the item at position `n`,
- **`sort()`**: sort the items in the list,
- **`slice(start, end)`**: return a sub-list slice starting at the position `start` up until the position `end`,
- **`reverse()`**: reverse the order of the list.

The List is one of the most used ADTs. For instance, if you need to store links to the most frequently accessed files in a system, a list is ideal: you can sort the links for display purposes, and remove links at will as the corresponding files become less frequently accessed.

The Stack or Queue should be preferred when the flexibility of List isn't needed. Using a simpler ADT ensures data is handled in a strict and robust way (FIFO or LIFO). It also makes the code easier to understand: knowing a variable is a Stack helps to see how data flows in and out.

The Sorted List

The **Sorted List** is useful when you need to maintain an *always sorted* list of items. In these cases, instead of figuring out the right position before each insertion in the list (and manually sorting it periodically), we use a Sorted List. Its insertions always keep the list sorted. None of its operations allow reordering its items: the list is guaranteed to be always sorted. The Sorted List has fewer operators than the List:

- **`insert(e)`**: insert item `e` at the right position in the list,
- **`remove(n)`**: remove the item at the position `n` in the list,
- **`get(n)`**: get the item at position `n`.

The Map

The **Map** (aka **Dictionary**) is used to store mappings between two objects: a **key** object and a **value** object. You can query a map with a key and get its associated value. For instance, you might use a map to store a user's ID number as key, and its full name as value. Then, given the ID number of a user, the map returns the related name. The operations for the Map are:

- **`set(key, value)`**: add a key-value mapping,
- **`delete(key)`**: remove `key` and its associated value,
- **`get(key)`**: retrieve the value that was associated to `key`.

The Set

The **Set** represents unordered groups of *unique* items, like mathematical sets described in Appendix III. They're used when the order of items you need to store is meaningless, or if you must ensure no items in the group occurs more than once. The common Set operations are:

- **`add(e)`**: add an item to the set or produce an error if the item is already in the set,
- **`list()`**: list the items in the set,
- **`delete(e)`**: remove an item from the set.

With these ADTs, you as a coder learned to interact with data, like a driver uses a car's dashboard. Now let's try to understand how the wires are structured *behind* that dashboard.

1.3 Structures

An Abstract Data Type only describes how variables of a given data type are operated. It provides a list of operations, but doesn't explain *how* data operations happen. Conversely, **data structures** describe *how* data is to be organized and accessed in the computer's memory. They provide ways for implementing ADTs in data-handling modules.

There are different ways to implement ADTs because there are different data structures. Selecting an ADT implementation that uses the best data structure according to your needs is essential for creating efficient computer programs. Next we'll explore the most common data structures and learn their strengths and weaknesses.

The Array

The **Array** is the simplest way to store a bunch of items in computer memory. It consists in allocating a sequential space in the computer memory, and writing your items sequentially in that space, marking the end of the sequence with a special `NULL` token.

Each object in an array occupies the same amount of space in memory. Imagine an array starting at memory address s, where each item occupy b bytes. The n^{th} item in the array can be obtained fetching b bytes starting from memory position $s + (b \times n)$.

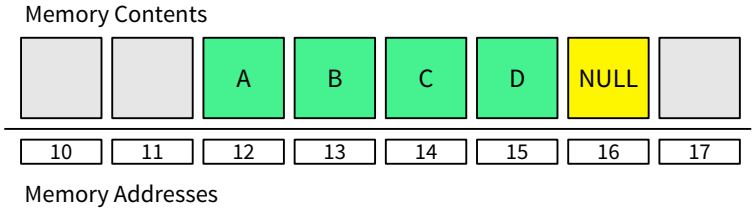

Figure 1.1 An array in the computer's memory.

This lets us access any item from an array *instantly*. The Array is especially useful for implementing the Stack, but can also used to implement Lists and Queues. Arrays are simple to code and have the advantage of instant access time. But they also have disadvantages.

It can be impractical to allocate large amounts of sequential space in the memory. If you need to grow an array, there might not be enough free space adjacent to it in the memory. Removing an item in the middle is problematic: you have to push *all* subsequent items one step back, or mark the removed item's memory space as "dead". Neither option is desirable. Similarly, adding an item causes you to push *all* subsequent items one step forward.

The Linked List

With **Linked Lists**, items are stored in a chain of cells that don't need to be at sequential memory addresses. Memory for each cell is allocated as needed. Each cell has a pointer indicating the address of the next cell in the chain. A cell with an empty pointer marks the end of the chain.

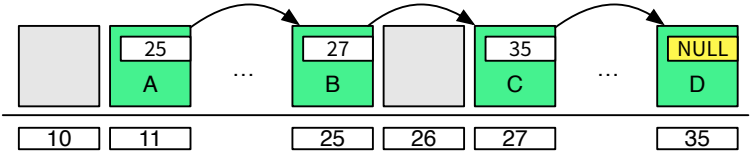

Figure 4.2 A Linked List in the computer's memory.

Linked lists can be used to implement Stacks, Lists, and Queues. There's no problem growing the list: each cell can be kept at any part of the memory. We can create lists as big as the amount of free memory we have. It's also easy to insert items in the middle or delete any item by changing the cell pointers:

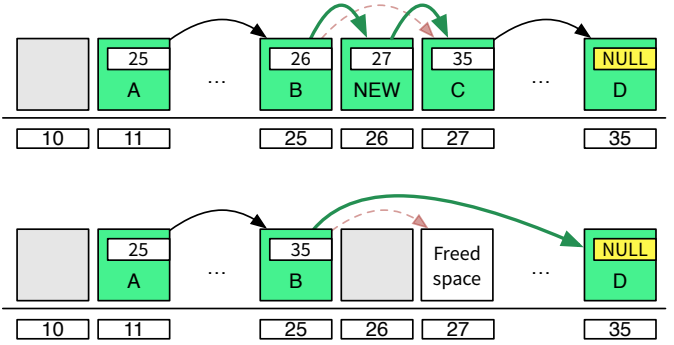

Figure 1.3 Adding an item between B and C; deleting C.

The Linked List also has its drawbacks: we can't instantly retrieve the n^{th} item. For that, we have to start searching at the first cell, use it to get the address of the second cell, then get the second cell, use its pointer to the next cell and so on, until we get to the n^{th} cell.

Also, if we're only given the address of a single cell, it's not easy to remove it or move backwards. With no other information, we can't know the address of the previous cell in the chain.

The Double Linked List

The **Double Linked List** is the Linked List with an extra: cells have two pointers: one to the cell that came before it, and other to the cell that comes after.

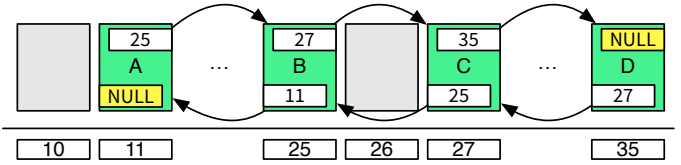

Figure 1.1 A double Linked List in the computer's memory.

It has the same benefits as the Linked List: no big chunk of memory preallocation is required, because memory space for new cells can be allocated on demand. And the extra pointers let us walk the chain of cells forwards *and* backwards. And if we're only given the address of a single cell, we're able to delete it.

Still, there's no way to access the n^{th} item instantly. Also, storing two pointers in each cell directly translates to more code complexity and more required memory to store our data.

Arrays vs. Linked Lists

Feature-rich programming languages often come with built-in implementations for List, Queue, Stack and other ADTs. These implementations often resort to a default data structure. Some of these implementations can even switch data structures automatically during runtime, based on how data is being accessed.

When performance isn't an issue, we can rely on these generic ADT implementations and not worry about data structures. But when performance must be optimal, or when working with a lower level language that doesn't have such features, *you* must decide which data structures to use. Analyze the operations your data must

undergo, and choose an implementation that uses an appropriate data structure. Linked Lists are preferable to Arrays when:

- You need insertions/deletions in the list to be extremely fast,
- You don't need random, unordered access to the data,
- You insert or delete items in the middle of a list,
- You can't evaluate the exact size of the list (it needs to grow or shrink throughout the execution).

Arrays are preferable over Linked Lists when:

- You frequently need random, unordered access to the data,
- You need extreme performance to access the items,
- The number of items doesn't change during execution, so you can easily allocate contiguous space of computer memory.

The Tree

Like the Linked List, the **Tree** employs memory cells that do not need to be contiguous in physical memory to store objects. Cells also have pointers to other cells. Unlike Linked Lists, cells and their pointers are not arranged as a linear chain of cells, but as a tree-like structure. Trees are especially suitable for hierarchical data, such as a file directory structure, or the command chain of an army.

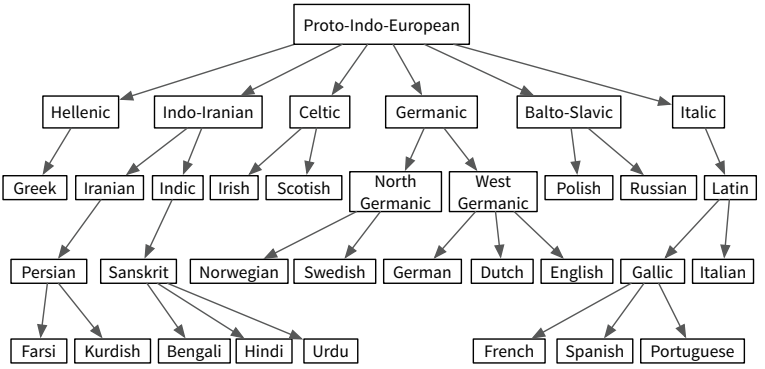

Figure 1.5 A tree with the origins of Indo-European languages.

In the Tree terminology, a cell is called a **node**, and a pointer from one cell to another is called an **edge**. The topmost node of a tree is the **Root Node**: the only node that doesn't have a parent. Apart from the Root Node, nodes in trees must have exactly *one* parent.[4]

Two nodes that have the same parent are siblings. A node's parent, grandparent, great-grandparent (and so on all the way to the Root Node) constitute the node's ancestors. Likewise, a node's children, grandchildren, great-grandchildren (and so on all the way to the bottom of the tree) are the node's descendants.

Nodes that do not have any children are **leaf nodes** (think of leaves in an actual tree 🌳). And a **path** between two nodes is a set of nodes and edges that can lead from one node to the other.

A node's **level** is the size of its path to the Root Node. The tree's **height** is the level of the deepest node in the tree. And finally, a set of trees can be referred to as a **forest**.

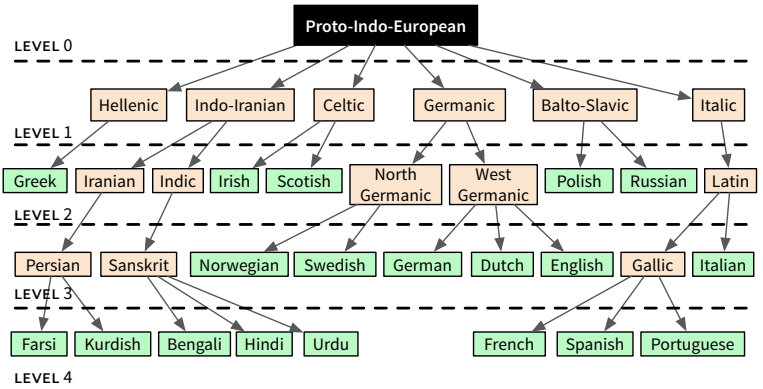

Figure 1.6 Leaves on this tree are present-day languages.

Binary Search Tree

A **Binary Search Tree** is a special type of Tree that can be efficiently searched. Nodes in Binary Search Trees can have at most two children. And nodes are positioned according to their value/key. Children nodes to the left of the parent must be smaller than the parent, children nodes to the right must be greater.

[4] If a node violates this rule, many search algorithms for trees won't work.

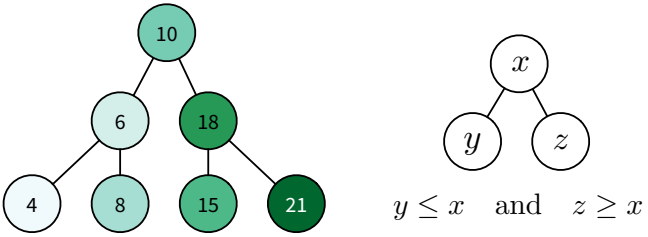

Figure 1.7 A sample binary search tree.

If the tree respects this property, it's easy to search for a node with a given key/value within the tree:

```
function find_node(binary_tree, value)
    node ← binary_tree.root_node
    while node:
        if node.value = value
            return node
        if value > node.value
            node ← node.right
        else
            node ← node.left
    return "NOT FOUND"
```

To insert an item, we search the value we want to insert in the tree. We take the last node explored in that search, and make its right or left pointer point to the new node:

```
function insert_node(binary_tree, new_node)
    node ← binary_tree.root_node
    while node:
        last_node ← node
        if new_node.value > node.value
            node ← node.right
        else
            node ← node.left
    if new_node.value > last_node.value
        last_node.right ← new_node
    else
        last_node.left ← new_node
```

Tree Balancing If we insert too many nodes in a Binary Search Tree, we end up with a tree of very high height, where many nodes have only one child. For example, if we insert nodes with keys/values always greater than the previous one, we end up with something that looks more like a Linked List. But we can rearrange nodes in a tree such that its height is reduced. This is called *tree balancing*. A perfectly balanced tree has the minimum possible height.

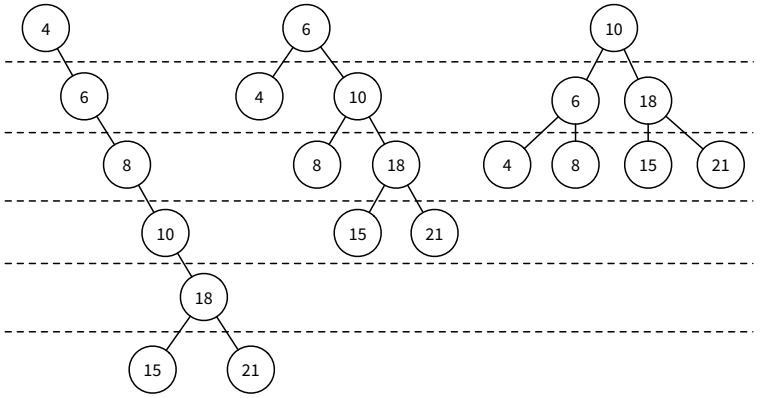

Figure 1.8 The same binary search tree in a very badly balanced form, in a somewhat balanced form, and in a perfectly balanced form.

Most operations with trees involve following links between nodes until we get to a specific one. The higher the height of the tree, the longer the average path between nodes, and the more times we need to access the memory. Therefore, it's important to reduce tree height. Building a perfectly balanced binary search tree from a sorted list of nodes can be done as follows:

```
function build_balanced(nodes)
    if nodes is empty
        return NULL
    middle ← nodes.length/2
    left ← nodes.slice(0, middle - 1)
    right ← nodes.slice(middle + 1, nodes.length)
    balanced ← BinaryTree.new(root=nodes[middle])
    balanced.left ← build_balanced(left)
    balanced.right ← build_balanced(right)
    return balanced
```

Consider a Binary Search Tree with n nodes. Its maximum height is n, in which case it looks like a Linked List. The minimum height, with the tree perfectly balanced, is $\log_2 n$. The complexity of searching an item in a binary search tree is proportional to its height. In the worst case, the search must descend to the lowest level, reaching all the way to the tree's leaves in order to find the item. Searching in a balanced Binary Search Tree with n items is thus $\mathcal{O}(\log n)$. That's why this data structure is often chosen for implementing Sets (which requires finding if items are already present) and Maps (which requires finding key-values).

However, tree balancing is an expensive operation, as it requires sorting all nodes. Rebalancing a tree after each insertion or deletion can greatly slow down these operations. Usually, trees are undergo balancing after several insertions and deletions take place. But balancing the tree from time to time is only a reasonable strategy for trees that are rarely changed.

To efficiently handle binary trees that change a lot, **self-balancing binary trees** were invented. Their procedures for inserting or removing items directly ensure the tree stays balanced. The **Red-Black Tree** is a famous example of a self-balancing tree, which colors nodes either "red" or "black" for its balancing strategy.[5] Red-Black Trees are frequently used to implement Maps: the map can be heavily edited in an efficient way, and finding any given key in the map remains fast because of self-balancing.

The **AVL Tree** is another breed of self-balancing trees. They require a bit more time to insert and delete items than Red-Black Trees, but tend to have better balancing. This means they're faster than Red-Black Trees for retrieving items. AVL Trees are often used to optimize performance in read-intensive scenarios.

Data is traditionally stored in magnetic disks that read data in big chunks. In these cases, the **B-Tree**, a generalization of Binary Trees, is used. In B-Trees, nodes may store more than one item and can have more than two children, making it efficient to operate with data in big chunks. As we'll soon see, B-Trees are commonly used in database systems.

[5] Self-balancing strategies are out of the scope of this book. If you are curious, there are videos on the Internet showing how they work.

The Binary Heap

The **Binary Heap** is a special type of Binary Search Tree, in which we can find the smallest (or highest) item instantly. This data structure is especially useful for implementing Priority Queues. In the Heap it costs $\mathcal{O}(1)$ to get the minimum (or maximum) item, because it is always the Root Node of the tree. Searching or inserting nodes still costs $\mathcal{O}(\log n)$. It has the same node placement rules as the Binary Search Tree, plus an extra restriction: a parent node must be greater (or smaller) than *both* its child nodes.

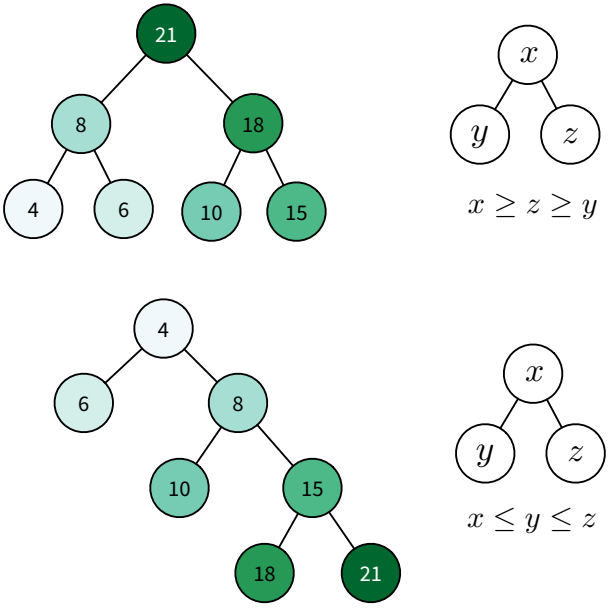

Figure 1.9 Nodes organized as a binary max-heap (top) and min-heap (bottom).

Remember to use the Binary Heap whenever you must frequently work with the maximum (or minimum) item of a set.

The Graph

The **Graph** is similar to the Tree. The difference is that there's no children or parent nodes, and therefore, no Root Node. Data is

freely arranged as nodes and edges, and any node can have multiple incoming and outgoing edges.

This is the most flexible data structure there is, and it can be used to represent almost any type of data. For example, graphs are ideal for representing a social network, where nodes are people and edges represent friendships.

The Hash Table

The **Hash Table** is a data structure that allows finding items in $\mathcal{O}(1)$ time. Searching for an item takes a constant amount of time, whether you're searching among 10 million or just 10 items.

Similarly to the Array, the Hash requires preallocating a big chunk of sequential memory to store data. But unlike the Array, items are not stored in an ordered sequence. The position an item occupies is "magically" given by a **hash function**. That's a special function that takes the data you want to store as input, and outputs a random-looking number. That number is interpreted as the memory position the item will be stored at.

This allows us to retrieve items instantly. A given value is first run through the hash function. The function will output the exact position the item should be stored in memory. Fetch that memory position. If the item was stored, you'll find it there.

There is a problem with Hash Tables: sometimes the hash function returns the same memory position for two different inputs. That's called a **hash collision**. When it happens, both items have to be stored at the same memory address (for instance, by using a Linked List that starts at the given address). Hash collisions are an extra overhead of CPU and memory, so we try to avoid it.

A proper hash function will return random-looking values for different inputs. Therefore, the larger the range of values the hash function can output, the more data positions are available, and the less probable it is for a hash collision to happen. So we ensure at least 50% of the space available to the Hash Table is free. Otherwise, collisions would be too frequent, causing a significant drop in the Hash Table's performance.

Hash Tables are often used to implement Maps and Sets. They allow faster insertions and deletions than tree-based data structures.

However, they require a very large chunk of sequential memory in order to work properly.

Conclusion

We learned data structures provide concrete ways to organize data in computer memory. Different data structures require different operations for storing, deleting, searching, and running though stored data. There's no silver bullet: you should choose which data structure to use according to the situation at hand.

We learned that instead of using data structures directly in our code, it's better to use Abstract Data Types. This isolates your code from data manipulation details, and lets you easily switch the data structure of your programs without changing any of the code.

Don't reinvent the wheel by trying to create the basic data structures and abstract data types from scratch. Unless if you're doing it for fun, for learning, or for research. Use third-party data handling libraries that were already well tested. Most languages have built-in support for these structures.

Reference

- Balancing a Binary Search Tree, by Stoimen
  - See it at https://code.energy/stoimen
- Cornell Lecture on Abstract Data Types and Data Structures
  - See it at https://code.energy/cornell-adt
- IITKGP nodes on Abstract Data Types
  - See it at https://code.energy/iitkgp
- Search Tree Implementation by "Interactive Python"
  - See it at https://code.energy/python-tree