kintsugi-stack-dsa-cpp: THEORY

> "Data Structures and Algorithms (DSA) should be viewed as essential tools, akin to the finely tuned parts of a Formula 1 car. The act of problem-solving with DSA serves as a crucial platform to exhibit both intelligence and creative thinking. The coding challenges themselves are simply various permutations of external factors; like the weather, track, wind, and rain in an F1 race. Ultimately, what dictates success in both domains; coding and Formula 1; is the mastery of planning, strategizing, maintaining flow, and ensuring precise code orchestration." - Siddhant Bali

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

About

Overview

- **Course Overview: Fundamental Data Structures and Algorithms**
  - **Primary Objectives**
    - The course covers all **fundamental data structures and algorithms** required for knowledge in the field.
    - There is a specific **focus on coding interviews**.
  - **Target Audience**

        - The content is designed for **beginners**.
        - It is also intended for **anyone who needs a refresher**.
    - **Core Areas of Study**
        - The course explores four main aspects of common data structures and algorithms:
            - **Design**: How they are conceptualized.
            - **Implementation**: How they are built.
            - **Tradeoffs**: The pros and cons of different approaches.
            - **Analysis**: How to evaluate their performance.
    - **Key Skills for Professional Success**
        - Students will learn to:
            - **Solve problems efficiently**.
            - **Analyze** problems effectively.
            - **Discuss tradeoffs** regarding different solutions.
            - **Communicate ideas** clearly to others.
    - **Value and Impact**
        - Mastering these skills can lead to a **difference of hundreds of thousands of dollars in compensation**.
        - The **problem-solving skills** learned are intended to serve a student for their **entire career**.
    - **Next Steps**
        - For further details on what the course covers, users should **scroll down**.
        - Users are encouraged to **get started** when they are ready.

Arrays

RAM

- **Introduction to Data Structures and RAM**
    - A **data structure** is defined as a specific way of **structuring data**.
    - In the context of computer science, this data is structured and stored inside of **RAM** (Random Access Memory).
    - RAM serves as the primary location where all **variables** are stored during the execution of code.
    - **Arrays** are the first data structure covered in this study.
    - When an array, such as one containing the values 1, 3, and 5, is used in code, that information must be stored within the RAM.

- **RAM Measurement and Binary Basics**
    - **RAM size** is measured in units called **bytes**.
    - It is common for modern computers to possess approximately **8 gigabytes** of RAM.
    - The term **"giga"** represents approximately **10 to the power of 9**, which is roughly **one billion**.
    - A single **byte** is composed of exactly **eight bits**.
    - A **bit** is a storage position for a single digit, with the restriction that the digit must be either a **0 or a 1**.
    - These zeros and ones constitute the fundamental **language of computers**.
    - Storage is hierarchical: individual bits form groups of bits, which form bytes, which collectively form RAM.

    - This RAM structure is then used to house more **advanced data structures**.

- **Representing Integers in Memory**

    - To store an integer like "1" in RAM, it must be represented in terms of bytes.
    - It is a common standard for **integers** to be represented by **four bytes** rather than a single byte.
    - Since one byte is eight bits, a four-byte integer is represented by **32 bits**.
    - To represent the integer **1** using 32 bits, the computer uses **31 zeros** followed by a single **1** at the very end.
    - Once the data is converted into this byte-based representation, it can be placed into RAM.

- **RAM Structure: Values and Addresses**

    - RAM can be visualized as a **contiguous block of data**.
    - RAM consists of two primary components: **values** and **addresses**.
    - Every value is stored at a **distinct location**, which is referred to as its **address**.
    - In technical diagrams, addresses are often distinguished from values by placing a **dollar sign ($)** in front of the address number.
    - The first address in a sequence is typically **0**.

- **Memory Properties of Arrays**

    - Arrays are considered the **most simple data structure** because they are stored in memory in the same way they are represented conceptually.
    - A defining characteristic of arrays is that they are always **contiguous**.
    - **Contiguous** means that values are stored one after another with **nothing in between** them.
    - For an array, the value 1 is stored first, followed immediately by 3, and then 5, without any gaps in the memory sequence.
    - While a programmer does not always choose the specific starting location of an array in RAM, the values will remain in a continuous block.

- **Data Sizes and Address Increments**

    - Because **32-bit integers** take up **four bytes**, the memory addresses for an integer array must **increment by four** for each new element.

```
Example Integer Array Memory Map:
Address: $0 -> Value: 1
Address: $4 -> Value: 3
Address: $8 -> Value: 5
```

    - If the addresses were incremented by only one ($0, $1, $2) for integers, there would not be enough room, as each integer requires a full four-byte slot.
    - Other data types, such as **characters** (e.g., 'A', 'B', 'C'), have different size requirements.
    - **ASCII characters** typically take up only **one byte** of memory.
    - Consequently, the addresses for a character array increment by **one** for each element.

```

```

```
Example Character Array Memory Map:
Address: $0 -> Value: 'A'
Address: $1 -> Value: 'B'
Address: $2 -> Value: 'C'
```

    - The general rule for storing values contiguously is that the address must increment by the **size of the value** being stored.

- **Theory vs. Practice**

    - Understanding how memory and addresses work provides the necessary **theoretical foundation** for data structures.
    - However, the primary focus of this study is **practical knowledge**.
    - This includes understanding how arrays are used in real-world scenarios, as well as their specific **properties and tradeoffs**.

Storing data in RAM is like a **long row of lockers** in a school hallway; each locker has a unique number (the address), and depending on what you are storing, you might occupy just one locker (a character) or a block of four lockers (an integer), but you always make sure your items are in a row without any empty lockers between them.

Static Arrays

- **Static Arrays Overview**

    - **Definition and Memory Storage**

        - An array is a **contiguous block of data**.
        - In RAM, it is stored exactly this way: as one continuous block.
        - Data structures generally involve two primary operations: **reading** and **writing**.

    - **Reading Data from an Array**

        - **The Concept of Indices**
            - Programmers use variables (e.g., `my array`) to represent the allocated array.
            - To read the first element, the intuitive action is to go to the first memory address and read the value.
            - Programmers do not need to know exact memory addresses for every value because they use **indexes**.
            - **Index 0** is always the first value; it does not start at one,.
            - Subsequent values follow in order: index 1, index 2, and so on.
        - **Efficiency of Reading**
            - Any index in an array can be automatically mapped to a location in memory.
            - This allows for **instant reading** of any value as long as the index is known.
            - This operation is called **Big O of 1** (constant time).
            - Even as an array grows larger, reading a specific index happens in the same amount of time in the worst case.
        - **RAM Properties (Random Access Memory)**

        - The ability to go to an arbitrary address and read it instantly is a property of **Random Access Memory (RAM)**.
        - "Random access" means any part of the memory can be accessed in constant time without starting from the beginning and looking through it sequentially.
        - If you want the third value, you do not need to go through the first and second values first.

- **Looping Through an Array**

    - To read every value from start to finish, a programmer starts at index 0.
    - In code, this is typically represented as `my array[i]`.
    - The process involves:
        1. Starting with index `i = 0`.
        2. Reading the value at `i`.
        3. Incrementing `i` to 1, then 2, etc..
        4. Stopping when `i` equals the **length of the array**.
    - This is usually implemented using a **for loop** or a **while loop**.

- **The Fixed Size Property of Static Arrays**

    - **Static arrays are of fixed size**.
    - If you declare an array of three values, it remains that size.
    - **Challenges with Adding Values:**
        - If you want to add a fourth value (e.g., the number 7) to an array of size three, it must be **contiguous** to the existing data.
        - Example: If the array ends at address 11, the new value must be at address 12.
        - However, programmers cannot always decide where RAM is allocated.
        - The next spot in RAM might be:
            - Occupied by another array.
            - Used by the Operating System for another purpose.
        - If you put the new value anywhere else in memory, the array is no longer contiguous, which breaks the definition of an array.
        - If the values are not contiguous, incrementing an index (e.g., going from index 2 to 3) would not lead the computer to the correct memory spot where the value is stored.
    - **Language Differences:**
        - Languages like **Python or JavaScript** often use **dynamic arrays** as the default, so users rarely encounter these fixed-size limitations,.
        - Static arrays are the specific focus of this limitation.

- **Writing and Removing Data**

    - **Initialization:**
        - When an array is first allocated, it might be empty but must store something.
        - Languages may initialize it to all zeros, or it may contain random, arbitrary "garbage" values.
    - **Basic Writing:**
        - Adding a value to an empty spot is an **instant operation (Big O of 1)**.

        - Because indices map directly to RAM positions, the computer knows exactly where to write the value.
    - **Removing Values:**
        - You cannot technically "delete" or deallocate a single spot in memory within a static array.
        - "Removing" a value involves **overwriting** it,.
        - Common methods include replacing the value with a 0, a -1, or another "irrelevant" marker,.
        - Overwriting/removing is a **Big O of 1** (constant time) operation because it just involves going to that specific memory position and changing it.

- **Inserting at Arbitrary Positions (Beginning or Middle)**

    - Arrays are **ordered values**; the position matters.
    - Inserting at the end (if space is available) is efficient (O(1)).
    - Inserting in the middle or beginning is **inefficient** because the order must be preserved.
    - **Example: Inserting 4 before 5 and 6**
        - If the array has `[5, 6, _]` and you want to insert 4 at the start to get ``:
            1. You cannot just overwrite 5 with 4, or you lose the 5.
            2. You must **shift** the existing values to the right.
            3. Step 1: Move the value at index 1 (6) to index 2 (1 + 1).
            4. Step 2: Move the value at index 0 (5) to index 1 (0 + 1).
            5. Step 3: Insert the value 4 into the now-empty index 0.
    - **Complexity of Shifting:**
        - In a long array, inserting at the beginning requires moving **every single value**.
        - This is a **Big O of N** operation, where N is the number of elements (length) of the array.
        - **Worst Case Scenario:**
            - Big O notation refers to the **worst case**.
            - If you insert in the middle, you only shift the values to the right of that point.
            - If you insert at the very beginning, you shift **all N values**.
            - To generalize for the worst case, we say the time complexity is **Big O of N**.

- **Removing from Arbitrary Positions**

    - Removing a value from the middle and wanting to "close the gap" (so the first value is the new start) requires shifting.
    - **Example: Removing the first element from ``**
        - To make 6 the new first value:
            1. Shift the value at index 1 (6) to the left to index 0 (1 - 1).
            2. Shift the value at index 2 (7) to the left to index 1 (2 - 1).
        - The result is ``. The original memory spot for 7 might still hold a value, but we treat it as non-existent or overwrite it with 0.
    - Like insertion, this is a **Big O of N** operation in the worst case because we may have to shift every subsequent value.

- **Summary of Static Array Operations**

- **Reading:** `O(1)` (Constant time).
- **Writing/Overwriting (at a known index):** `O(1)` (Constant time).
- **Inserting at the end:** `O(1)` (Assuming space is available).
- **Removing from the end:** `O(1)`.
- **Inserting in the middle/beginning:** `O(N)` (Worst case due to shifting).
- **Removing from the middle/beginning:** `O(N)` (Worst case due to shifting).

- **Coding Logic for Access and Loops**

```
// To read an element at index i
my_array[i]

// To loop through an array
for (i = 0; i < length; i++) {
    print(my_array[i]);
}
```

Dynamic Arrays

- DYNAMIC ARRAYS OVERVIEW
  - **Dynamic arrays** are much more common and useful than **static arrays**.
  - They solve the **fixed size problem** inherent in static arrays.
  - **Language Defaults**:
    - In **Python** and **JavaScript**, dynamic arrays are the default array type.
    - In **Java**, they are implemented as an **ArrayList**.
    - In **C++**, they are called a **Vector**.
  - **Initialization**:
    - You do not necessarily have to specify a size when creating one.
    - If no size is specified, it initializes to a **default size**.
    - Example: In Java, the default size of an ArrayList is typically 10, though this number is arbitrary.
- INTERNAL MECHANISMS AND TERMINOLOGY
  - **Capacity (Size)**: The total number of slots allocated in memory (e.g., a size of 3).
  - **Length**: The actual number of elements currently stored in the array (e.g., an empty array of size 3 has a length of 0).
  - **Pointers**:
    - The implementation maintains a **pointer** or variable that tracks the index of the **last element**.
    - Example: If the first element is at index 0 and the next is at index 1, the pointer identifies index 1 as the last element.
    - The **length** is calculated using this pointer; if the pointer is at index 1, the length is 2 (since indices start at 0).
- CORE OPERATIONS

    - **Pushing**:
        - This refers to adding an element to the **end** of the array at the next empty spot.
        - It is a **Big O(1)** time operation because the system knows exactly where the next spot is via the pointer.
    - **Popping**:
        - This refers to removing the last value from the end of the array.
        - It is a **Big O(1)** time operation because the pointer identifies the last element immediately.
        - After removing the value, the pointer shifts left by one index to mark the new end.

- THE RESIZING PROCESS (GROWING THE ARRAY)

    - **The Problem**: When the array runs out of space (e.g., capacity is 3 and you try to push a fourth element), you cannot simply allocate more memory right next to the existing array.
    - **The Solution**: The dynamic array must allocate a brand new array to contain all elements.
    - **Steps for Resizing**:
        1. **Allocation**: A new array is created in a different, random location in memory.
        2. **Capacity Doubling**: The new array's size is typically **double** the original capacity.
            - Example: An original size of 3 becomes size 6. An original size of 6 becomes size 12.
        3. **Copying**: All original values are copied from the old array to the new array starting at the beginning (index 0 to 0, index 1 to 1, etc.).
        4. **Insertion**: The new value (the one that triggered the resize) is added to the new space.
        5. **Deallocation**: The original array is **deallocated** or "freed," telling the operating system it is no longer in use so the memory can be reused.

- TIME COMPLEXITY AND AMORTIZATION

    - **Resizing Complexity**:
        - Allocating new memory takes **O(N)** time, where N is the size of the memory allocated.
        - Copying every value from the old array to the new one is also an **O(N)** operation.
    - **Amortized Time Complexity**:
        - Although a push that triggers a resize takes O(N), resizing is **infrequent**.
        - Because we double the capacity each time, most pushes are simple O(1) operations.
        - **Amortized complexity** is essentially the **average time** of an operation.
        - Pushing a value to a dynamic array has an **amortized time complexity of O(1)**.
    - **Mathematical Explanation (The Power Series)**:
        - If you push 8 elements into an array that started at size 1:
            - Total operations = 8 (last allocation/moves) + 4 + 2 + 1 = 15 operations.
        - This calculation is **dominated by the last term**.
        - The sum of all previous terms will always be less than or equal to the last term.
        - Therefore, the total operations for N elements is always **less than or equal to 2 * N**.
        - In Big O, **O(2 * N)** simplifies to **O(N)**.
        - Since it takes O(N) total time to push N elements, the average (amortized) time per push is **O(1)**.

- BIG O NOTATION AND CONSTANTS

    - **Constants are Ignored**:

        - We do not care about constants multiplied by N (like O(2 * N)) or added to N (like O(N + 8)).
        - Whether the constant is 10, 100, or 1000, it is simplified to O(N).
    - **Why Constants are Ignored**:
        - Big O focuses on **growth rates** as input sizes (N) become very large.
        - We care about how the runtime grows: is it linear (O(N)) or quadratic (O(N^2))?.
        - **Interaction of Functions**:
            - A faster-growing function like N^2 will always eventually intersect and surpass a slower-growing function like N or $2N$, regardless of the starting constant.
            - We ignore small input sizes (like 1, 8, or 1000) because only very large inputs significantly slow down a CPU.

- DYNAMIC ARRAY PERFORMANCE SUMMARY

    - **Accessing an Element**: **O(1)** (can access any index anywhere in constant time).
    - **Push/Pop at the End**: **O(1)** (pushing is O(1) amortized).
    - **Resizing**: Allows for growth even when initial space is full.
    - **Inserting/Removing in the Middle**: **O(N)**.
        - This is the downside of dynamic arrays.
        - You must shift every subsequent value over to make room or fill a gap.
        - There is **no amortization** for this; it is always O(N) in the worst case.

- ANALOGY

    - Imagine a dynamic array as a growing dinner party table. You start with a small table (the original capacity). When more guests arrive than there are chairs, you don't just squeeze them in; you move the entire party to a new room with a table twice as big. While the move itself (copying elements) takes a lot of effort, it happens so rarely compared to guests just sitting down (pushing elements) that on average, seating a guest is still a very quick process.

Stacks

- **Stacks Overview**

    - A stack is a common data structure in programming that typically supports three specific operations.
    - **Core Operations**
        - **Push**: Adding an element to the end of the stack.
        - **Pop**: Removing an element from the end of the stack.
        - **Peek**: Looking at the last element in the stack to read it without removing it, unlike a pop operation.
    - **Efficiency Requirements**
        - These operations must run efficiently in **constant time**.
        - Pushing should be a constant time operation.
        - Popping should be a constant time operation.
        - Peeking at the last element should be a constant time operation.

- **Implementation with Dynamic Arrays**

    - There is no need to design a stack data structure from scratch because **dynamic arrays** satisfy all efficiency requirements.
    - A stack can be implemented using a dynamic array.
    - In standard programming, developers typically use the default dynamic array provided by the language they are using.
    - **Space Management**
        - Because stacks are implemented with dynamic arrays, users do not need to worry about running out of space.
        - The system will automatically allocate more space when needed.

- **Visual and Physical Representation**

    - **Array Representation**: A stack is essentially an array where elements are pushed onto the end.
    - **Vertical Representation**: Some visualize stacks as vertical structures where values are "thrown" into the top.
        - If the stack is empty, the first element goes to the bottom.
        - Subsequent elements are added to the "top," which is considered the "end" of the array.
    - **Tracking Elements**
        - A **pointer** or a count of the number of elements added so far is maintained.
        - This allows the system to know exactly where to push the next element into the next empty spot.

- **The Popping Mechanism**

    - Elements can only be removed from the end of the array (the top of the stack).
    - Removing from the end is an efficient process.
    - **Internal Memory Handling**
        - When an element is popped, the system does not necessarily delete that portion of memory.
        - Instead, it internally updates the marker to indicate a new "end" of the array or a new "top" of the stack.

- **LIFO (Last-In-First-Out) Principle**

    - **Definition**: Stacks are a **LIFO** data structure, meaning "Last In, First Out".
    - **The Reverse Order Property**: The order in which elements are inserted is the reverse of the order in which they are removed.
        - The last element added is the first element removed.
        - The second-to-last element inserted is the second-to-last removed if all elements are popped.
    - **Step-by-Step Example**
        1. Push a **1**.
        2. Push a **2**.
        3. Push a **3**.
        4. Pop the **3**.
        5. Pop the **2**.
        6. Pop the **1**.

    - This sequence demonstrates that the order of removal (3, 2, 1) is the reverse of the insertion order (1, 2, 3).

- **Use Cases**

    - **Reversing Sequences**: Because of the LIFO property, stacks can be used to reverse a sequence of values, such as numbers or a string of characters like "ABC".
    - **Complex Problem Solving**: Stacks are used for many other complex programming use cases and problems.

Analogy for Understanding: Think of a **stack of cafeteria trays**. When you add a new tray, you place it on the very top (Push). When someone needs a tray, they take the one that was most recently placed on top (Pop). The tray at the very bottom of the stack—the first one placed there—will be the very last one to be used.

Linked Lists

Singly Linked Lists

- **Singly Linked Lists Overview**

    - Linked lists have significant overlap with arrays but also many differences.
    - They are comprised of **list nodes**, sometimes simply called **nodes**.

- **List Node Definition and Structure**

    - A list node is an object that encapsulates at least two specific things:
        - **Value**: This can be an integer, a character, another object, or anything else (e.g., the value 4).
        - **Next**: This is a **pointer** that indicates what the next list node in the linked list is.
    - This pointer system is the method for connecting multiple list nodes together to form a linked list.
    - If a `next` pointer points at **null**, it means it points at nothing, signifying the end of the list or a single-node list.

- **Building and Connecting a Linked List**

    - **Initialization**:
        - To build a list, you first create a list node (e.g., a node with the string value "red").
        - By default, the `next` pointer is empty or set to **null**, meaning it is not yet connected to another node.
    - **Memory Allocation**:
        - When a node object is created, it is stored somewhere in memory.
        - Programmers do not necessarily control the exact memory location.
        - Multiple nodes (e.g., "red", "blue", and "green") can be created separately in memory.
    - **Implementation of Connections**:
        - In most programming languages, the `next` pointer is implemented as a **reference** to another list node object.
        - To connect nodes in code, you set the `next` property of one node to the variable representing the next node.

        - **Example**: `ListNode1.next = ListNode2`.
    - **Low-Level Mechanics**:
        - Under the hood, the second node (e.g., "blue") is stored at a specific memory address (e.g., "xx") while the first node is at another (e.g., "y").
        - The pointer tells the system the specific address of the next node.
        - Programmers usually use references to objects and do not need to worry about manual address management.

- **Memory Layout Differences**
    - **Linked Lists**: Nodes may be stored in **random order** in memory and are only connected via pointers.
        - A "red" node can point to a "blue" node even if the blue node is stored "behind" it or elsewhere in a "big mess" in memory.
    - **Arrays**: Arrays are stored in memory in the exact same sequence as they are used by the programmer.

- **Traversing a Linked List**
    - **Goal**: Start at the beginning and loop through every node until the end is reached.
    - **Detection of End**: The end is identified when a node's pointer points to **null**.
    - **Traversal Logic and Code**:
        - A variable named `cur` (current) is initialized to the first node.
        - A `while` loop runs as long as `cur` is not pointing at **null**.

```
cur = head
while cur != null:
    cur = cur.next
```

    - **Step-by-Step Execution Example**:
        1. `cur` starts at the "red" node.
        2. The condition `cur != null` is true, so the loop enters.
        3. `cur` is set to `cur.next`, following the pointer to the "blue" node.
        4. The condition `cur != null` is checked again; it is still true.
        5. `cur` is set to `cur.next`, moving to the "green" node.
        6. The condition is checked; it is still true.
        7. `cur` is set to `cur.next`, which is **null** (the green node points to nothing).
        8. The condition `cur != null` becomes false; the loop terminates.

- **Complexity and Edge Cases**
    - **Infinite Loops**: If a node's `next` pointer points back to a previous node (e.g., the third node points back to the first), the program will loop forever and eventually crash.
    - **Time Complexity of Traversal**: Traversing the entire list is **O(n)**, where **n** is the size of the linked list. This is identical to array traversal.

- **Head and Tail Management**

    - It is helpful to keep track of two specific pointers:
        - **Head**: The first element of the list.
        - **Tail**: The last element of the list.
    - In a list of **size one**, both the Head and Tail pointers point to the same single node.

- **Operations: Insertion at the End**

    - **Scenario**: Adding a new node (e.g., "purple") to the end of the list.
    - **Steps**:
        1. Create the new node.
        2. Set the current `tail.next` to the new node (`tail.next = ListNode4`).
        3. Update the `tail` pointer to the new node (`tail = ListNode4` or `tail = tail.next`).
    - **Time Complexity**: This is a **constant time operation O(1)** because the tail pointer allows direct access without iterating through the list. This is the same as an array.

- **Operations: Removal of Nodes**

    - **Constant Time Removal O(1)**: Removing a node (beginning or end) is **O(1)**, provided there is a pointer to the **previous node** of the one being removed.
    - **Procedure for Middle Removal**:
        - To remove a node between "head" and a third node, set `head.next` to `head.next.next`.
        - This "chains" the fields: `head.next` refers to the second node, and `head.next.next` refers to the third node.
        - By setting the head's next pointer to the third node, the second node is bypassed.
    - **Memory Management**:
        - In languages with **garbage collection**, the system automatically frees the memory of a node once no pointers are pointing to it.
    - **Comparison to Arrays**: Linked list removal is beneficial because it does not require shifting all subsequent elements like an array does.
    - **Arbitrary Removal**: If you must find a specific element first without a pointer, removal takes **O(n)** time because you must search the list.

Analogy for Understanding A singly linked list is like a **scavenger hunt**. Each location (node) contains a prize (value) and a slip of paper (pointer) telling you exactly where to find the next location. You can't jump straight to the third location unless you go to the first and second locations to get the clues. Even if the locations are scattered randomly across a city (memory), the clues keep them in a specific logical order.

Doubly Linked Lists

- **DOUBLY LINKED LISTS**

    - **Overview and Structure**

        - A doubly linked list is a variation of a singly linked list.
        - The primary difference is the presence of two pointers, known as **double pointers**.
        - Each node contains:
            - A **next pointer** that points to the next node in the list.

        - A **previous pointer** that points to the previous node in the list.
    - **Structural Example with Three Nodes (1, 2, 3):**
        - The next pointers connect each node to the one following it.
        - The previous pointers connect each node to the one before it.
        - **Boundary Pointers:**
            - The first node has no previous node, so its previous pointer is set to **null**.
            - This null value indicates the beginning of the list.
            - The last node has its next pointer set to **null**, indicating the end of the list.
    - **Navigation Pointers:**
        - Like singly linked lists, doubly linked lists typically use a **head** pointer for the first node and a **tail** pointer for the last node.

- **Operations: Adding a New Node to the End**

    - Adding a node to the end of a doubly linked list is very similar to doing so in a singly linked list.
    - **Example: Adding Node 4 to a list ending at Node 3:**
        1. Take the next pointer of the current tail (Node 3) and assign it to the new node (Node 4).
        2. Set the previous pointer of the new node (Node 4) to point at the current tail (Node 3) to preserve doubly linked properties.
        3. Shift the tail pointer to the new node (Node 4).
    - **Importance of Order:**
        - The order of operations is critical.
        - The pointers must be connected (setting the previous pointer to the current tail) **before** updating the tail pointer to the new node.
    - **Code Logic for Addition:**
        - `tail.next = newNode`
        - `newNode.prev = tail`
        - `tail = tail.next` (or `tail = newNode`)

- **Operations: Removing the Last Node**

    - Removing the last node is easier with a doubly linked list because it allows looking backwards.
    - **Efficiency Advantage:**
        - In a singly linked list, you would have to start at the beginning and iterate forward to reach the second-to-last node.
        - In a doubly linked list, if you have a pointer to the tail, you can simply follow the previous pointer to find the preceding node.
    - **Example: Removing Node 3 from a list of three nodes:**
        1. Start at the tail (Node 3) and follow its previous pointer to reach the previous node (Node 2).
        2. Take the next pointer of Node 2 and set it to **null**.
        3. Update the tail pointer to point at Node 2.
    - **Post-Removal State:**
        - The list size is reduced (e.g., from three nodes to two).

        - Technically, the removed node (Node 3) might still exist in memory with its previous pointer still pointing to Node 2.
        - However, because there are no references to it from the list and the tail has been moved, it is effectively gone.
        - Depending on the programming language, **garbage collection** may delete this node automatically.
    - **Code Logic for Removal:**
        - `node2 = tail.prev`
        - `node2.next = null`
        - `tail = node2`

- **Time Complexity Analysis**

    - **End Operations:**
        - Both inserting and removing a value at the end are **constant time operations (O(1))**.
        - These operations do not require looping through the entire list regardless of its size.
    - **Accessing Elements:**
        - Accessing an arbitrary or random element (e.g., the second, third, or seventh element) is a **linear time operation (O(n))**.
        - You cannot jump to an index; you must follow pointers from the beginning (or end) until you reach the desired element.
    - **Middle Operations:**
        - Inserting or removing from the middle is technically a **constant time operation (O(1))** once you are at the location.
        - Unlike arrays, you do not need to shift all subsequent elements; you only change the pointers to skip or include a node.
        - **The Caveat:** To perform this O(1) removal or insertion, you must first arrive at that element. Since arriving usually requires iterating from the start, the total process is typically a **linear time operation (O(n))**.

- **Comparison: Linked Lists vs. Dynamic Arrays**

    - **Stacks:**
        - Stacks can be implemented with linked lists because they support O(1) push and pop at the end.
        - However, dynamic arrays are much more common for stacks because they offer the same efficiency for end operations while also allowing arbitrary element access.
    - **Summary Table of Complexities:**
        - **Arrays:**
            - Accessing any arbitrary index: **O(1)**.
            - Insert/Remove at end: **O(1)**.
            - Insert/Remove in middle: **O(n)** (requires shifting elements).
        - **Linked Lists (Singly or Doubly):**
            - Accessing arbitrary element: **O(n)**.
            - Insert/Remove at end: **O(1)**.

            - Insert/Remove in middle: **O(n)** (due to the need to iterate to the element first).
        - **Utility:**
            - Arrays are generally more useful for problem-solving because efficient arbitrary access is highly important.
            - Linked lists have a slight theoretical benefit for middle operations, but this is often negated by the need to iterate through the list to find the insertion/removal point.

Analogy to Solidify Understanding Think of a **Singly Linked List** like a **one-way train** where each car only knows the car in front of it; if you want to find the person second-to-last, you have to walk from the engine all the way down. A **Doubly Linked List** is like a **train with walkie-talkies between every car**; every car knows who is in front *and* who is behind. If you are at the very last car (the tail) and need to unhook it, you can just talk to the car right behind you to tell it that it's now the new end of the train.

Queues

- Queues
    - Definition and Overview
        - Queues are an important data structure that share similarities with stacks.
        - The defining characteristic of a queue is that it follows the FIFO principle, which stands for "First In First Out".
        - Under FIFO, the first element added to the queue is the first element that is removed.
        - Elements are always removed in the same order they were originally added to the queue.
    - Core Operations
        - Enqueue (NQ)
            - This operation refers to adding elements to the queue.
            - Enqueuing is similar to the "push" operation used in stacks.
            - When you enqueue an element, you add it to the end of the queue.
        - Dequeue (DQ)
            - This operation refers to removing a value from the queue.
            - Unlike a stack, dequeuing involves removing the element from the beginning of the queue rather than the end.
    - Efficiency and Time Complexity
        - A primary goal is to perform queue operations efficiently.
        - Adding an element to the end of the queue should ideally be done in constant time, O(1).
        - Removing an element from the beginning of the queue should also be done in constant time, O(1).
    - Comparison: Arrays vs. Linked Lists
        - Array Implementation
            - Constant time removal from the beginning is not possible with standard arrays.
            - If an element is removed from the beginning of an array, every other element must be shifted over to fill the gap.
            - This shifting process results in a Big O of N (O(n)) time operation.
            - While it is technically possible to build a queue with an array, the implementation is significantly more complicated.
        - Linked List Implementation

        - Linked lists can achieve O(1) constant time for both adding and removing elements.
        - Implementing a queue is much simpler using a linked list compared to an array.
    - Step-by-Step Linked List Implementation
        - Initial State
            - Start with an empty queue where the current pointer is set to null.
            - To manage the queue, two specific pointers are maintained: a head pointer and a tail pointer.
        - Adding the First Element
            - Example: Adding the value "red" to the queue.
            - Create a list node for the value.
            - Both the head and tail pointers are set to point at this initial node.
            - The current pointer is set to this node because it is the only element in the queue.
        - Adding Subsequent Elements
            - Example: Adding the value "blue" to the queue.
            - To insert the new node, take the "next" pointer of the current tail node and set it to point to the new node.
            - The head pointer remains at the first element added.
            - The tail pointer is updated to point at the newest node, which is now the end of the queue.
        - Removing Elements (Dequeue)
            - Removing from the head of the list is a simple and efficient process.
            - To dequeue, take the current pointer (the head) and set it to `current.next`.
            - By moving the pointer to the next node, the original first node is bypassed and can be treated as if it no longer exists in the list.
            - This method allows for constant time removal from the beginning.
    - Usage
        - Queues are a very commonly used data structure in computer science.

Analogy: Think of a queue like a line at a grocery store checkout. The first person to join the line is the first person served and the first to leave (FIFO). If the store had to shift every single person forward physically every time the person at the front left (like an array implementation), it would be very inefficient. Instead, the cashier just looks at whoever is next in line (like moving a head pointer in a linked list), which is a quick and constant process regardless of how long the line is.

Recursion

Factorial

- **Recursion and Factorial Fundamentals**
    - Recursion is a vital concept for data structures and algorithms that will be used throughout the course.
    - It is often difficult to understand initially, so it is introduced using **one branch recursion** through a math formula: **n factorial**.
    - **Definition of n Factorial:**
        - It is a shorthand for writing $n * (n - 1) * (n - 2)$ all the way until reaching the base number, which is 1.

    - **Example:** 5 factorial (5!) is $5 * 4 * 3 * 2 * 1$, which equals 120.
    - While recursion is useful for learning, you do not strictly need it for this calculation; a loop is often the easiest way.

- **The Logic of Recursion and Sub-problems**
    - Recursion is centered on the idea of **sub-problems**.
    - A complex problem is broken down into a slightly smaller version of the same problem.
    - **Example Logic:** 5 factorial is the same as $5 * 4!$ because 4 factorial is $(4 * 3 * 2 * 1)$.
    - **General Equation:** $n! = n * (n - 1)!$.
    - By using this equation, a big problem (n!) is reduced to a smaller sub-problem ((n - 1)!).

- **Visualizing Recursion with Decision Trees**
    - The best way to represent recursion is through a **decision tree**.
    - In the case of factorials, it is called **one branch recursion** because there is only one decision/path to follow at each step.
    - To compute 5!, the tree shows that we must first solve the sub-problem of 4! before we can multiply it by 5.

- **Recursive Implementation and Code Structure**
    - A recursive function for factorials typically takes an integer and returns an integer.
    - **The Recursive Step:**
        - This is when the function calls itself to solve the sub-problem.
        - In the code, this looks like calling the same function you are currently inside.
        - **Logic:** `return n * factorial(n - 1)`.
    - **The Base Case:**
        - This is the point where the series ends and the multiplication stops.
        - In math, **1 factorial is mapped to the constant 1**, so no further calculation is needed at that point.
        - **Math Note:** 0 factorial is also defined as 1.
        - **Importance:** Without a base case, the function would call itself infinitely, leading to increasingly negative numbers, and it would never return to compute the original result.
    - **Recursive Code Example:**

```
def factorial(n):
    # Base Case
    if n <= 1:
        return 1

    # Recursive Step
    return n * factorial(n - 1)
```

        - The `n <= 1` condition is used to handle both the base case of 1 and the zero case.

- **Step-by-Step Execution Trace (Using 5 Factorial)**
    - **1. Going Down (The Calls):**

        - Input 5: Not <= 1, so it calls `5 * factorial(4)`.
        - Input 4: Not <= 1, so it calls `4 * factorial(3)`.
        - Input 3: Not <= 1, so it calls `3 * factorial(2)`.
        - Input 2: Not <= 1, so it calls `2 * factorial(1)`.
        - Input 1: **Base Case reached**. It returns 1.
    - **2. Going Back Up (The Returns):**
        - The call for 2! receives 1, calculates $2 * 1$, and returns **2** to its parent.
        - The call for 3! receives 2, calculates $3 * 2$, and returns **6** to its parent.
        - The call for 4! receives 6, calculates $4 * 6$, and returns **24** to its parent.
        - The original call for 5! receives 24, calculates $5 * 24$, and returns the final result: **120**.

- **Complexity Analysis**

    - **Time Complexity:**
        - The complexity is **Big O of N (O(n))**, where n is the input value.
        - This is because there are n total steps or multiplications to perform.
        - This is identical to the time complexity of a standard for or while loop.
    - **Memory (Space) Complexity:**
        - The recursive solution has **Big O of N (O(n)) memory complexity**.
        - This is **worse** than a loop because each function call must be saved in memory.
        - Every function call is "put on hold" while waiting for its sub-problem to return.
        - When the code reaches the base case (1!), all n function calls are "alive" and taking up space in memory at the same time.
        - Each function call stores its own input parameter and tracking information.

- **Iterative (Non-Recursive) Solution**

    - The non-recursive approach is more memory-efficient as it typically uses a single variable.
    - **Iterative Logic:**
        - Initialize a `result` variable to 1.
        - Use a `while` loop that runs as long as `n > 1`.
        - Multiply the `result` by `n`, then decrement `n` by 1 in each iteration.
        - This calculates $n * (n-1) * (n-2)...$ until n reaches 1.
    - **Iterative Code Example:**

```
def factorial_iterative(n):
    result = 1
    while n > 1:
        result = result * n
        n = n - 1
    return result
```

    - This approach avoids the O(n) memory overhead of the recursive call stack.

Fibonacci

- **TWO-BRANCH RECURSION: THE FIBONACCI EXAMPLE**
    - **General Mathematical Formula**

    - To calculate the nth Fibonacci number, the formula is: $F(n) = F(n-1) + F(n-2)$.
    - This means you take the $n-1$ Fibonacci number and the $n-2$ Fibonacci number and add them together to get the nth value.
- **The Necessity of Base Cases**
    - Without base cases, calculating a Fibonacci number (like the fifth one) would continue forever.
    - **Defined Base Cases**:
        - The zeroth Fibonacci number is 0.
        - The first Fibonacci number is 1.
    - **Example: Calculating the Second Fibonacci Number (F(2))**:
        - Formula: $F(2) = F(2-1) + F(2-2)$.
        - This simplifies to $F(1) + F(0)$.
        - Using the base cases: $1 + 0 = 1$.
        - Therefore, the second Fibonacci number is 1.
    - **Example: Calculating the Third Fibonacci Number (F(3))**:
        - Formula: $F(3) = F(3-1) + F(3-2)$.
        - This simplifies to $F(2) + F(1)$.
        - Since $F(2) = 1$ and $F(1) = 1$, the calculation is $1 + 1$.
        - Therefore, the third Fibonacci number is 2.
- **Recursion vs. Looping**
    - Fibonacci numbers can be calculated in a straightforward way by looping through them, which allows for calculating values as high as desired.
    - The recursive solution is actually less efficient than looping, but it is used to illustrate "two-branch recursion".
- **The Recursive Decision Tree for F(5)**
    - A decision tree is used to visualize the recursive subproblems.
    - **Level 1**: To compute $F(5)$, the problem is broken into $F(4)$ and $F(3)$.
    - **Level 2 (Left Path)**: $F(4)$ is broken into $F(3)$ and $F(2)$.
    - **Level 2 (Right Path)**: $F(3)$ is broken into $F(2)$ and $F(1)$.
    - **Further Breakdown**:
        - $F(3)$ breaks into $F(2)$ and $F(1)$.
        - $F(2)$ breaks into $F(1)$ and $F(0)$.
    - This process continues until every branch reaches a base case (0 or 1).
- **Code Implementation and Logic**
    - **Base Case Code**:
        - Individual cases: `if n == 0: return 0` and `if n == 1: return 1`.
        - **Condensed version**: `if n <= 1: return n`.
        - This works because if n is 1, it returns 1, and if n is 0, it returns 0.
    - **Recursive Case Code**:
        - The function must compute the $n-1$ and $n-2$ values, add them together, and return that integer value.

```
# Representative logic based on the description
if n <= 1:
    return n
return recursive_fib(n - 1) + recursive_fib(n - 2)
```

- **Executing the Recursive Steps for F(5)**
  - The code executes down the tree and returns values back up once base cases are hit.
  - **Calculating F(2)**: Reaches base cases $F(1)$ (returns 1) and $F(0)$ (returns 0). Adding them ($1 + 0$) returns 1.
  - **Calculating F(3)**: Uses results from $F(2)$ (which is 1) and $F(1)$ (which is 1). Adding them returns 2.
  - **Calculating F(4)**: Uses results from $F(3)$ (which is 2) and $F(2)$ (which is 1). Adding them returns 3.
  - **Calculating F(5)**: Uses results from $F(4)$ (which is 3) and $F(3)$ (which is 2). Adding them returns the final result: 5.
- **Time Complexity Analysis**
  - **Loop Technique**: The time complexity is $O(N)$.
  - **Recursive Technique**: The complexity is much higher and requires analyzing the decision tree.
  - **Tree Height**:
    - The height of the tree (number of levels) is n.
    - For $F(5)$, the longest path is $5 \to 4 \to 3 \to 2 \to 1$, resulting in roughly 5 levels.
  - **Branching and Doubling**:
    - Each node generally breaks into two subproblems because there are two branches in the recursive tree.
    - Theoretically, the number of terms doubles at every level: $1 \to 2 \to 4 \to 8...$.
    - The number of values in the last level is roughly 2^n.
  - **The Power Series and Upper Bounds**:
    - In a series where values double (1, 2, 4, 8), the series is dominated by the last term.
    - The last term (2^n) is greater than or equal to all previous terms combined.
    - Therefore, 2^n serves as an upper bound for the total number of values in the decision tree.
  - **Big O Constants**:
    - Precise values like $n+1$ or $n-1$ in the exponent do not matter in Big O notation.
    - Multiplying by a constant (like 2×2^n) also results in the same Big O complexity.
    - **Final Complexity**: The recursive solution is bounded by $O(2^n)$.

Prompt for Notes Formatting

```
Create super depth notes in Markdown (.md) format with 100% information
preserved, no loss. Use simple grammar and keep everything clear, direct,
and well-structured. using headings, subheadings,paragraphs, statements and
code blocks when needed. Include every detail, definition, example, and
step exactly from the source. transform the given content into clean,
readable .md format.
and no #, just nested - lines plaintext
```

```
\to => =>
\times => *
$ => ** or `
```

End-of-File

The kintsugi-stack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with 💚 Kintsugi-Programmer