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

0. About

- Introduction and Primary Focus
  - This PreReqs focuses on teaching all the **fundamental data structures and algorithms** that individuals need to know. There is a specific emphasis on preparing for **coding interviews**.
- Target Audience
  - This curriculum is suitable for two main groups:
    - 1. **Beginners**.
    - 2. Anyone who **just needs a refresher**.
- PreReqs Content and Scope
  - The instruction will cover all the common data structures and algorithms. The discussion will include detailed analysis of four key areas related to these structures and algorithms:
    - 1. **Design**
    - 2. **Implementation**
    - 3. **Tradeoffs**
    - 4. **Analysis**
  - Career Impact and Compensation Value
    - Mastering the skills taught in this PreReqs is critical because the ability to perform efficiently can significantly impact compensation. This difference in compensation can amount to **hundreds of thousands of dollars**.
  - The core competencies that yield this high value include:

        - Solving these problems **efficiently**.
        - **Analyzing** them.
        - Discussing their **tradeoffs**.
        - **Communicating the idea** to others.
    - The goal is that the problem solving skills acquired in this course will serve the participant for their **entire career**.
- Next Steps
    - For participants seeking **more details on what is going to be covered**, they should **scroll down**. Once participants are ready, they are encouraged to **get started**.

1. Arrays

1.1. RAM(Random-access memory)

- Data Structures and RAM
    - A data structure is a way of structuring data.
    - In computer science, data structures involve structuring data inside of RAM (Random Access Memory).
    - RAM is where all variables are stored.
    - Example: An array containing the numbers 1, 3, and 5 is information stored in RAM.
- RAM Measurement and Composition
    - RAM is measured in bytes.
    - It is common for many computers to have 8 gigabytes of RAM.
    - Giga means about 10^9, or approximately a billion.
    - A byte is eight bits.`1 byte = _ _ _ _ _ _ _ _ (0 or 1)`
    - eg: `1 Int = 4 Bytes = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _`
    - eg: `'1' decimal = 00000000000000000000000000000001 binary`
    - A bit can be thought of as a position that stores a digit.
    - The restriction on the digit stored in a bit is that it can only be a zero or a one.
    - Zeros and ones are the language of computers.
    - Storage Hierarchy: Individual bits form multiple bits, which form bytes, which ultimately form RAM.
    - RAM is used to store advanced data structures.
- Storing Values and Addressing in RAM

```
[ Array storage in RAM ]


  Array: [ 1 | 3 | 5 ]
            |   |   |
    +-------+   |   +-------+
    |           |           |
    V           V           V
    RAM:
  +-----------+-----+-----+-----+----------+
  | Value     |  1  |  3  |  5  |   ...    |
  +-----------+-----+-----+-----+----------+
```

```
| Address   | $0  | $4  | $8  |  $12 ... | GAP OF 4B
+-----------+-----+-----+-----+----------+

['a','b','c'] =
+-----------+-----+-----+-----+----------+
| Value     |'a'  |'b'  |'c'  | ...      |
+-----------+-----+-----+-----+----------+
| Address   | $0  | $1  | $2  |  $3 ...  | GAP OF 1B
+-----------+-----+-----+-----+----------+
```

    - RAM can be conceptually viewed as a contiguous block of data.
        - i.e. Nothing Can be B/w 2 adjacent blocks
    - RAM has two components: values and addresses.
    - Values are the data being stored.
    - Every value is stored at a distinct location called an address.
    - In representations, a dollar sign ($) may be used in front of every address to distinguish it from the stored values.
    - Example: The first address might be $0.
- Representation of Integers in RAM
    - It is common for integers to be represented by four bytes, not a single byte.
    - Four bytes is equivalent to 32 bits.
    - Example of storing the integer '1' using 32 bits:
        - The representation will be 31 zeros, followed by a single one at the end.
    - The process involves taking the value, representing it in terms of bytes, and then placing that representation into RAM.

1.2. Static Arrays

- Arrays
    - Arrays are considered the most simple data structure.
    - **Definition/Key Property:** Arrays are always contiguous.
    - Contiguous storage means that nothing is stored in between the values of the array (e.g., nothing between the 3 and the 5).
    - Arrays look exactly the same in memory as they are represented conceptually: a contiguous set of values.
    - When storing an array in RAM, the starting location (address) is not always chosen by the user.
    - Arrays can store different types of values, such as integers or characters.
- Array Storage and Address Incrementing (The Size Rule)
    - Values are stored contiguously regardless of how large or small they are.
    - The address must be incremented by the size of the value being stored.
    - **Example: Storing 32-bit Integers (4 bytes)**

```
Array: [ 1 | 3 | 5 ]
          |   |   |
```

```
    +-------+   |   +-------+
    |           |           |
    V           V           V
  +-----------+-----+-----+-----+----------+
  | Value     |  1  |  3  |  5  |   ...    |
  +-----------+-----+-----+-----+----------+
  | Address   | $0  | $4  | $8  |  $12 ... |
  +-----------+-----+-----+-----+----------+
```

- Each integer takes four bytes to store.
- If the address stored the value '1' starting at address $0.
- It uses four bytes ($0, $1, $2, $3).
- The next value ('3') must be stored starting at address $4 (incremented by four).
- The final value ('5') would be stored starting at address $8.
- If each value only took one byte, the addresses would increment by one ($0, $1, $2), but 32-bit integers require four bytes.

- **Example: Storing Characters (1 byte)**

```
Array: [ a | b | c ]
           |   |   |
    +-------+   |   +-------+
    |           |           |
    V           V           V
  +-----------+-----+-----+-----+----------+
  | Value     |  a  |  b  |  c  |   ...    |
  +-----------+-----+-----+-----+----------+
  | Address   | $0  | $1  | $2  |   $3 ... |
  +-----------+-----+-----+-----+----------+
```

- Characters (like A, B, C) are stored continuously.
- Characters typically take only one byte to store in memory.
- This one-byte size is typical when storing ASKY characters, although knowing the specific encoding (ASKY) is not super important.
- If the first character 'A' starts at address $0.
- The next address is incremented by one: 'B' is stored at address $1.
- The next character 'C' is stored at address $2.

- Course Focus

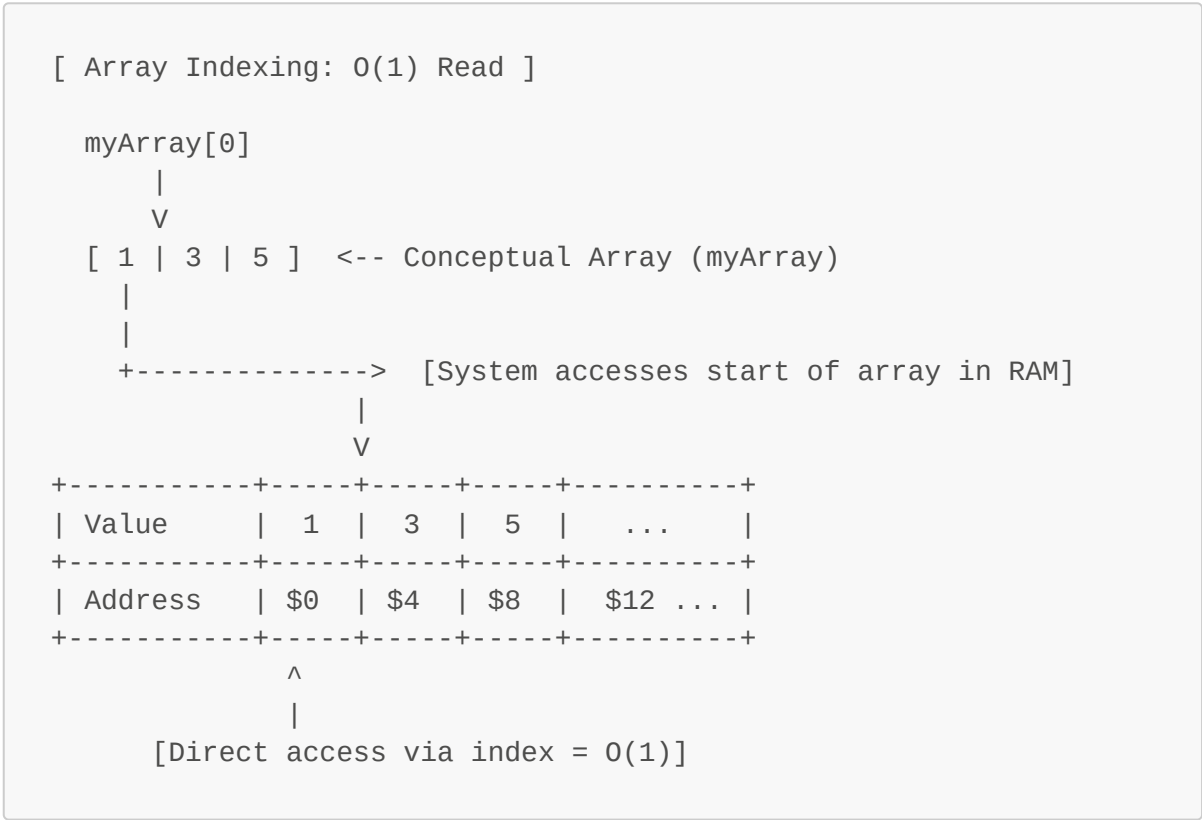
  - The information presented covers theoretical aspects of memory and how arrays are stored.
  - This course will mostly focus on practical knowledge.
  - Upcoming topics will cover array properties, usage, and tradeoffs.

- ARRAY PROPERTIES AND OPERATIONS (STATIC ARRAYS)

```
+--------------------+-------------+
|      Operation     | Big-O Time  |
```

```
+---------------------+-------------+
| r / w i-th element  |     O(1)    |
| Insert / Remove End |     O(1)    |
| Insert Middle       |     O(n)    |
| Remove Middle       |     O(n)    |
+---------------------+-------------+
```

- Array Definition and Structure
  - An array is defined as a **contiguous block of data**.
  - Arrays are stored in RAM (Random Access Memory) as a contiguous block of data.
- Common Array Operations
  - The two most common operations for any data structure are reading to the data or writing to the data.
- I. Reading Data from Arrays



  - Indexing
    - Programmers use indexes to access values in an array.
    - The first value is **always at index zero**, not index one.
    - Subsequent values are at index one, two, etc..
  - Reading Mechanism
    - To read the first element of an array (e.g., `my array`), the intuitive action is to go to the first address in memory and read the value.
    - Under the hood, this is exactly what happens.
    - Programmers do not need to know the exact address of every single value.
    - When reading the value at index zero, the system automatically goes to that location in memory and reads the stored value.
  - Efficiency of Reading

        - Since any index in the array can be automatically mapped to a location in memory, any value can be read **instantly** if the index is available.
        - This efficient operation is represented by **big O of one (O(1))**.
        - O(1) means that in the worst case, reading a value (assuming the index is known, e.g., index 2) is an **instant operation**.
        - Reading happens in **constant time**, regardless of how large the array becomes.
    - Role of RAM
        - The ability to go to any arbitrary address in RAM and read the value is a property of RAM.
        - **RAM stands for Random Access Memory**.
        - Random access means memory can be accessed randomly in constant time.
        - We do not have to start from the beginning of RAM and keep looking for a value; we can access it automatically.
        - This is important because if there is a lot of RAM, we avoid going through the entire RAM every time.
        - If accessing the third value, we do not have to go through the first and second values.
    - Looping Through Arrays (Reading All Values)
        - After reading the first value, we read the second, third, and continue until the end of the array.
        - In code, reading a value at index `i` is typically represented by `my array at index i`.
        - The index `i` is incremented (to one, then two, etc.).
        - The loop stops when `i` equals the length of the array.
        - This process loops through the entire input array.
        - In most languages, this can be done using a `for loop` or a `while loop`.
- II. Writing Data and Limitations of Static Arrays
    - `ARRAY IS NOT DYNAMIC, YOU CAN'T APPEND SPACE TO ADD MORE ELEMENTS`
    - Fixed Size Property
        - Arrays possess the property of **fixed size**.
        - Example: If an array was declared to hold three values, it is fixed size.
    - The Contiguity Constraint for Adding Values
        - If we want to add a fourth value (e.g., a seven) to an array of size three, the new value must be added to maintain contiguity.
        - Example: If the existing data ends just before address 12, the seven would need to be added precisely at address 12 to maintain a contiguous array of size four.
        - Limitations in RAM allocation prevent simple expansion.
            - We do not always get to decide exactly where RAM is allocated or where values are stored.
            - The required spot in RAM might already be used by another array or the operating system.
        - If a value is put anywhere else in memory, the array loses its contiguous property, resulting in two separate arrays, which is not the intended outcome.
        - Loss of Contiguity Breaks Operations
            - If contiguity is lost, looping through the array breaks.

            - Example: If a loop is at index two and increments to index three, it will attempt to access the expected contiguous spot; if the value is stored elsewhere, the operation will fail.
        - Static Arrays Defined
            - This fixed-size constraint is the biggest limitation of **static arrays**.
            - Static arrays are defined as **fixed size arrays**.
            - **Recommendation is to first define upperbound static arrays with initialisation with zeros or use vector array**
        - Static vs. Dynamic Arrays
            - Programming languages like Python or JavaScript often use **dynamic arrays** by default, which means programmers rarely encounter the fixed-size limitation or worry about running out of space.
        - Initialization
            - When a static array (e.g., size three) is allocated, there must be something stored in memory.
            - Default initialization varies:
                - Languages usually initialize values to all zero.
                - Sometimes they do not initialize them at all, resulting in random arbitrary values.
    - III. Overwriting and Efficient Operations (O(1))
        - Writing/Adding at the End
            - Adding a new value to the end of the array (at the next empty spot being tracked) is an **instant operation**.
            - We know the index (e.g., index 2) and thus instantly know the position in RAM.
            - **Writing to any position** is a **big O of one (O(1)) operation**; it is constant time.
            - Inserting a value at the end of the array is also a **big O of one (O(1))** time operation, assuming an empty space is available.
        - Removing/Overwriting
            - When "removing" a value in a static array, we cannot actually delete or deallocate it from memory.
            - Removal means **overwriting** the value (e.g., replacing it with a zero).
            - We effectively declare that the original value is "no longer relevant," even though it remains in memory.
            - **Removing a value** is also efficient: **big O of one (O(1))**.
            - It is a constant time operation because it involves going to that memory position and overwriting it (e.g., replacing it with a zero or negative 1).
            - Removing the value at the end of the array is a **big O of one (O(1))** time operation, assuming at least one element exists to remove.
        - Summary of Efficient Operations
            - Reading or overwriting the arbitrary element E (meaning any index) is done in **big O of one (O(1))** time, or constant time.
    - IV. Arbitrary Insertion and Removal (Inefficient Operations: O(N))
        - Ordered Values
            - In arrays, including static arrays, the **order of the values matters**.
            - Values are ordered because of the way indexes work (e.g., five comes before six).
        - Inserting into the Middle or Beginning

        - Inserting at an arbitrary position (middle or beginning) is **not efficient**.
        - If we simply overwrite an existing value to insert a new one, the desired order is lost (e.g., replacing five with four results in four, six, not four, five, six).
    - The Shifting Mechanism for Insertion
        - To insert a value (e.g., four) while maintaining order (e.g., four, five, six), the existing elements must be **shifted to the right**.
        - **Step 1: Shift existing values.** We must shift the elements starting from the end towards the right.
            - Example: Before shifting the five, the six must be shifted to prevent loss of data.
            - Six (at index 1) moves to index 1 + 1 (index 2).
            - Five (at index 0) moves to index 0 + 1 (index 1).
        - **Step 2: Insert the new value.** Only after shifting is completed can the new value (four) be placed in the now-available position (index 0).
        - **Downside:** This requires moving every existing value in the array.
    - Insertion Time Complexity (Worst Case)
        - Shifting many elements is **not very efficient**.
        - This operation is called **big O of N (O(N))**.
        - **N** refers to the **number of elements** in the array (the length), not the size of the array.
        - **Big O** notation refers to the **worst case** scenario.
        - **Worst Case for Insertion:** If inserting a value at the beginning of a filled array, we must shift every value in the array (N values) over to the right by one.
        - **Worst Case Time Complexity:** Inserting into the middle of an array is **big O of N**.
            - Note: While inserting into an empty array would be O(1), the worst case determines the Big O notation.
        - Example of a better case: If inserting between five and six to achieve five, four, six, only the six would need to be shifted (one value), not every value. However, the generalized worst case is still O(N).
    - Arbitrary Removal Time Complexity
        - Removing a value at an arbitrary index requires similar shifting.
        - Arbitrary removal means we want to act as if the value no longer exists, not just replace it with a zero.
    - The Shifting Mechanism for Removal
        - Removal requires shifting values to the **left**.
        - Example: If removing the value at index 0 from (5, 6, 7), the goal is to shift the 6 and 7 forward.
            - Six (at index 1) moves to index 1 - 1 (index 0).
            - Seven (at index 2) moves to index 2 - 1 (index 1).
        - Memory Note: The memory allocated for the element that was removed still exists and is reflected in RAM, even if the resulting array (6, 7) appears smaller.
        - **Worst Case Time Complexity:** Removing in the middle of an array is **big O of N (O(N))**, as we might have to shift every value.

- ARRAY PROPERTIES AND OPERATIONS (STATIC ARRAYS) CODE

```cpp
#include <iostream>

using std::cout;
using std::endl;

// Insert n into arr at the next open position.
// Length is the number of 'real' values in arr, and capacity
// is the size (aka memory allocated for the fixed size array).
void insertEnd(int arr[], int n, int length, int capacity) {
    if (length < capacity) {
        arr[length] = n;
    }
}

// Remove from the last position in the array if the array
// is not empty (i.e. length is non-zero).
void removeEnd(int arr[], int length) {
    if (length > 0) {
        arr[length - 1] = 0;
    }
}

// Insert n into index i after shifting elements to the right.
// Assuming i is a valid index and arr is not full.
void insertMiddle(int arr[], int i, int n, int length) {
    for (int index = length - 1; index >= i; index--) {
        arr[index + 1] = arr[index];
    }
    arr[i] = n;
}

// Remove value at index i before shifting elements to the left.
// Assuming i is a valid index.
void removeMiddle(int arr[], int i, int length) {
    for (int index = i + 1; index < length; index++) {
        arr[index - 1] = arr[index];
    }
}

void printArr(int arr[], int capacity) {
    for (int i = 0; i < capacity; i++) {
        cout << arr[i] << ' ';
    }
    cout << endl;
}
```

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