Data Structures and Algorithms (Spring2024)

Week03 (28/29/29-Feb-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Quick Sort

Introduction to Quick Sort:

Quick Sort is a highly efficient and widely used sorting algorithm that follows the Divide and Conquer paradigm. Developed by Tony Hoare in 1960, Quick Sort exhibits remarkable performance for **average and best-case scenarios**. Its efficiency lies in the fact that it doesn't require additional memory space for sorting and is an in-place sorting algorithm.

Motivation for Quick Sort:

The motivation behind Quick Sort lies in its ability to efficiently handle large datasets with an average-case time complexity of $O(n \log n)$, making it one of the fastest sorting algorithms in practice. Unlike Bubble Sort, Selection Sort, and Insertion Sort, Quick Sort does not rely on comparing adjacent elements and, in many cases, outperforms other $O(n \log n)$ algorithms like Merge Sort due to its lower constant factors.

Methodology - Divide and Conquer:

Quick Sort employs a Divide and Conquer strategy to efficiently sort an array or list. The algorithm's methodology can be outlined as follows:

1. **Divide:**
    - Choose a "pivot" element from the array. The choice of the pivot can vary, but a common strategy is to select any of the first, middle, and the last elements.
    - Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
2. **Conquer:**
    - Recursively apply Quick Sort to the sub-arrays created in the Divide step.
3. **Combine:**
    - The sorted sub-arrays are then combined, resulting in the entire array being sorted.

Key Steps:

- **Pivot Selection:**
    - The efficiency of Quick Sort heavily depends on the choice of the pivot. A well-chosen pivot can significantly reduce the number of comparisons.

- **Partitioning:**
  - The partitioning step involves rearranging the elements so that those less than the pivot come before it, and those greater come after it.
- **Recursion:**
  - Quick Sort recursively applies the same process to the sub-arrays created during partitioning until the entire array is sorted.

Advantages:

- **In-Place Sorting:**
  - Quick Sort requires only a constant amount of additional memory space, making it an in-place sorting algorithm.
- **Average-Case Efficiency:**
  - The average-case time complexity of Quick Sort is $O(n \log n)$, making it highly efficient for large datasets.
- **Adaptability:**
  - Quick Sort is adaptive and performs well even on partially sorted arrays.

Conclusion:

Quick Sort's efficiency, adaptability, and in-place nature make it a favored choice for various applications where sorting is a critical operation. However, it's important to note that its worst-case time complexity is $O(n^2)$, which occurs when poorly chosen pivots lead to unbalanced partitions. Nonetheless, in practice, Quick Sort's average-case performance often outweighs the worst-case scenario, making it a preferred sorting algorithm for many real-world scenarios.

Quick Sort in Action

Example Array: [7, 2, 1, 6, 8, 5, 3, 4]

Step 1: Initial Array

```
[7, 2, 1, 6, 8, 5, 3, 4]
```

Step 2: First Recursive Call (Pivot = 4)

```
[2, 1, 3, |4|, 8, 5, 7, 6]
```

- The pivot, 4, is chosen as the last element (consistent with always choosing the last element as the pivot).
- Partitioning: Elements less than 4 go to the left, and elements greater than 4 go to the right.
- Recursive Call on Left Sub-array: [2, 1, 3]
- Recursive Call on Right Sub-array: [8, 5, 7, 6]

Step 3: Recursive Call on Left Sub-array (Pivot = 3)

```
[1, 2, |3|, ||4||, 8, 5, 7, 6]
```

- The pivot, 3, is chosen from the last element of the left sub-array.
- Partitioning: Left sub-array is already sorted.

- Recursive Call on Right Sub-array: [8, 5, 7, 6]

Step 4: Recursive Call on Right Sub-array (Pivot = 6)

```
[1, 2, |3|, ||4||, 5, |6|, 7, 8]
```

- The pivot, 6, is chosen from the last element of the right sub-array.
- Partitioning: Left sub-array is already sorted.
- Recursive Call on Right Sub-array: [8, 7]

Step 5: Recursive Call on Right Sub-array (Pivot = 8)

```
[||1, 2, 3, 4, 5, 6||, 7, 8]
```

- The pivot, 7, is chosen from the last element of the right sub-array.
- Partitioning: Left sub-array is already sorted.
- Recursive Call on Right Sub-array: []

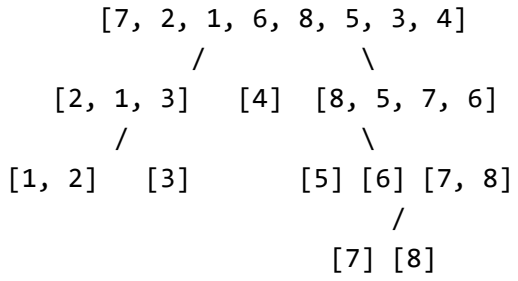
Final Sorted Array:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

This example ensures that the pivot is always the last element during the partitioning step, demonstrating how Quick Sort efficiently sorts the array by recursively choosing and placing the last element as the pivot.

Tree View:

Let's represent the steps of the Quick Sort algorithm on the example array in a tree format:



Quick Sort Implementation

Here's a simple implementation of the Quick Sort algorithm in C++ using an array for the input data:

```cpp
#include <iostream>


int partition(int arr[], int low, int high) {
    // Choose the pivot (for simplicity, we choose the last element)
    int pivot = arr[high];

    // Index of the smaller element
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the pivo
t
        if (arr[j] <= pivot) {
            i++;
            // Swap arr[i] and arr[j]
            swap(arr[i], arr[j]);
        }
    }

    // Swap arr[i+1] and arr[high] (put the pivot in its correct plac
e)
    swap(arr[i + 1], arr[high]);

    // Return the index of the pivot element
    return i + 1;
}



void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Find the partitioning index
        int pivotIndex = partition(arr, low, high);

        // Recursively sort the elements before and after the pivot
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
```

```
int main() {
    int arr[] = {7, 2, 1, 6, 8, 5, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }

    quickSort(arr, 0, n - 1);

    std::cout << "\nSorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }

    return 0;
}
```

This C++ code defines the `quickSort` function for the Quick Sort algorithm, along with a `partition` function to rearrange elements based on a chosen pivot. The `main` function demonstrates the usage of the algorithm on an example array.

Analyzing Quick Sort

The time complexity of Quick Sort is analyzed based on its recursive divide-and-conquer approach. The key steps are as follows:

1. **Partitioning:**
    - Partitioning an array of size n takes $O(n)$ time. This is because each element is compared to the pivot exactly once during the partitioning process.
2. **Recursion:**
    - After partitioning, Quick Sort is applied recursively to the two sub-arrays. The recurrence relation for the time complexity is $T(n) = 2T(n/2) + O(n)$.
3. **Combining:**
    - Combining the sorted sub-arrays does not contribute significantly to the time complexity.

The time complexity of Quick Sort is determined as follows:

$$T(n) = 2T(n/2) + O(n)$$

The time complexity of Quick Sort is $O(n \log n)$ in the average and best-case scenarios.

Conclusion: In conclusion, Quick Sort exhibits excellent average-case performance with a time complexity of $O(n \log n)$. However, it is essential to note that the worst-case time complexity of

Pivot Selection in Quick Sort:

The choice of the pivot in Quick Sort significantly impacts the algorithm's performance. The efficiency of Quick Sort is evident when a well-chosen pivot leads to balanced partitions during each recursion step. The common strategies for pivot selection are:

1. **First/Last Element:**
    - Choose the first or last element of the array as the pivot. This is simple but can lead to suboptimal performance for already sorted or nearly sorted arrays.
2. **Random Pivot:**
    - Randomly select a pivot element. This reduces the chances of encountering worst-case scenarios and improves average-case performance.
3. **Median-of-Three:**
    - Choose the pivot as the median of the first, middle, and last elements. This aims to provide a more balanced partitioning, especially for partially sorted arrays.

The impact of pivot selection on complexity is mainly observed in the worst-case scenario, where poorly chosen pivots result in unbalanced partitions. In such cases, the time complexity degrades to $O(n^2)$. However, on average, Quick Sort exhibits a time complexity of $O(n \log n)$ due to its divide-and-conquer nature.

Taxonomy of Data Structures

The taxonomy of data structures provides a systematic classification of these fundamental components, aiding in the understanding and organization of the vast array of structures used in computer science. It's important to recognize that varying taxonomies exist, and the perspective from which data structures are viewed greatly influences their categorization.

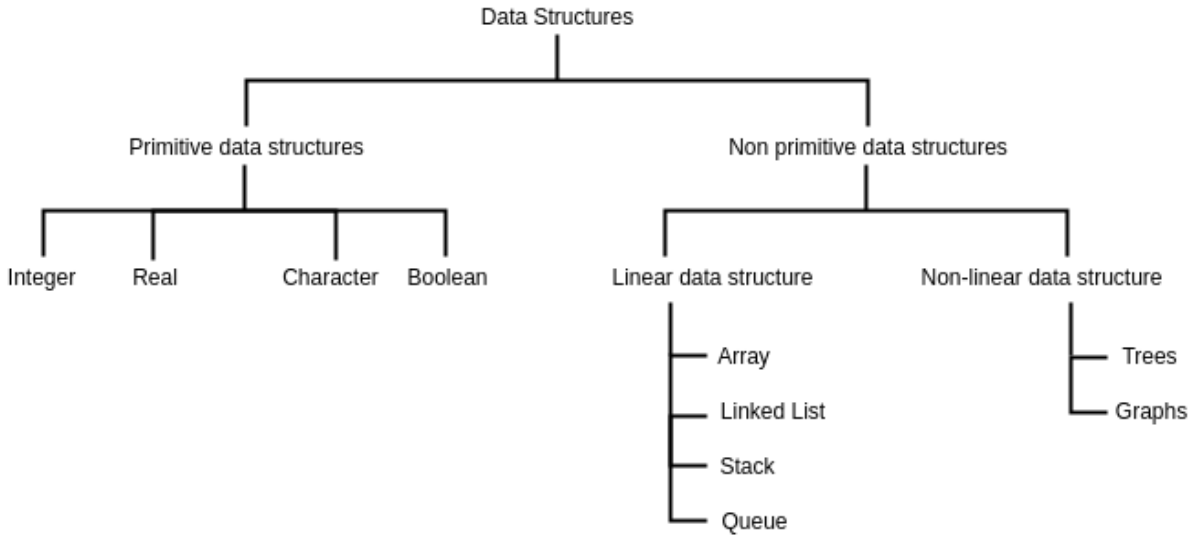
Varying Taxonomies:

1. **Linear vs. Non-Linear:**
    - One common taxonomy classifies data structures into linear and non-linear categories. Linear structures, like arrays and linked lists, organize data sequentially, while non-linear structures, such as trees and graphs, have a more hierarchical organization.
2. **Primitive vs. Non-Primitive:**
    - Another perspective distinguishes between primitive and non-primitive data structures. Primitive structures include basic types like integers and floats, while non-primitive structures encompass arrays, linked lists, and more complex entities.
3. **Specialized vs. General-Purpose:**
    - Data structures are also categorized based on specialization. Specialized structures, like heaps and bloom filters, serve specific purposes, while general-purpose structures, such as arrays and linked lists, offer versatility.

Perspective Matters:

1. **Usage in Algorithms:**
    - Data structures are often categorized based on their utility in algorithms. For instance, arrays and linked lists are crucial for sorting algorithms, while trees and graphs play pivotal roles in searching and traversing algorithms.
2. **Memory Organization:**
    - From a memory organization perspective, structures are classified as contiguous (arrays) or non-contiguous (linked lists) in storage. This perspective is crucial for understanding memory efficiency and access patterns.
3. **Behavioral Characteristics:**
    - Behavioral characteristics, like dynamic or static behavior, influence the taxonomy. Arrays exhibit static behavior with fixed sizes, while linked lists demonstrate dynamic behavior with variable sizes.

In conclusion, the taxonomy of data structures offers a lens through which we can organize, understand, and apply these fundamental components in computer science. Here is one possible organization:



Data Structures

Definition: A data structure is a specialized format for organizing and storing data to perform efficient operations on that data. It defines the way data is organized, stored, and manipulated, allowing for easy access and modification. Data structures are essential in computer science and programming for managing and organizing data to meet specific computational needs.

Key Characteristics:

1. **Organization:** Data structures define the organization and layout of data in a systematic manner.
2. **Operations:** They provide operations or functions for manipulating the stored data efficiently.
3. **Efficiency:** Data structures are designed to optimize various operations, such as insertion, deletion, searching, and sorting.
4. **Abstraction:** They often provide a level of abstraction, hiding the underlying implementation details.
5. **Memory Management:** Data structures may involve memory management to allocate and deallocate memory efficiently.

Examples:

- Arrays, Linked Lists, Stacks, Queues, Trees, Graphs, Hash Tables, Heaps

Data Types

Definition: A data type is a classification of data that specifies the type of values a variable can hold and the operations that can be performed on it. It defines a set of values along with the allowable operations on those values. Data types are fundamental in programming languages for ensuring type safety and providing a foundation for variable declarations.

Key Characteristics:

1. **Representation:** Data types define how data is represented in a computer's memory.
2. **Operations:** They specify the operations that can be performed on variables of that type.
3. **Size:** Each data type has a specific size, determining the amount of memory it occupies.
4. **Default Values:** Data types often have default values assigned when a variable is declared.
5. **Type Safety:** Data types help in enforcing type safety, preventing unintended operations on variables.

Examples:

- Integer, Float/Double, Character, Boolean, String, Array, Pointer, Structure/Class (in object-oriented programming)

Data Structures vs Data Types

Commonalities:

- **Abstraction:** Both data types and data structures provide a level of abstraction, hiding implementation details and allowing users to interact with them at a higher level.
- **Memory Usage:** Both concepts involve considerations related to memory, including allocation, deallocation, and access.

Differences:

- **Purpose:** Data types primarily define the nature of individual variables, specifying the type of values they can hold. Data structures, on the other hand, organize and structure multiple variables to facilitate efficient operations.
- **Level of Complexity:** Data types are relatively simpler, defining individual variables, while data structures involve more complexity in organizing and managing multiple variables.
- **Operations:** While data types specify operations that can be performed on individual variables, data structures provide operations for managing and manipulating a collection of variables.

In summary, data types are fundamental building blocks that define the nature of individual variables, whereas data structures organize and structure multiple variables to efficiently perform operations on them. Both are crucial concepts in programming, working together to enable effective data handling and manipulation.

Where do Arrays, Strings, Structures, Classes stand?

Arrays and Strings:

- **Data Type Perspective:**
  - Arrays and strings are often considered data types because they represent a particular way of organizing and storing data.
  - In this context, they define the type of values a variable (or set of variables) can hold.
- **Data Structure Perspective:**
  - From another viewpoint, arrays and strings can be seen as basic data structures because they organize and structure multiple elements to form a cohesive unit.
  - Especially when considering dynamic arrays, multidimensional arrays, or advanced string manipulation, they exhibit characteristics of data structures.

Structures and Classes:

- **Data Type Perspective:**
  - Structures and classes are often considered composite or user-defined data types because they allow the grouping of variables with different data types under a single name.
  - They define the blueprint for creating objects.
- **Data Structure Perspective:**
  - From a broader perspective, structures and classes can be viewed as user-defined data structures. They encapsulate data and behavior within a single entity.
  - In object-oriented programming, classes are fundamental to creating complex data structures.

In practice, the terminology might be used interchangeably based on the specific context. What's crucial is understanding the characteristics and use cases of these constructs, whether classified as data types or as part of data structures. It's worth noting that the distinction can be subtle, and different educational materials or programming languages might use varied classifications.

Representation of Primitive Data Types

In C++, primitive data types are stored in memory, and their representation depends on the type of data and the architecture of the system. In the context of x86 architecture, let's delve into the representation of common primitive data types: `int`, `float`, `double`, `char`, and `bool`.

1. Integer (`int`):

In C++, the `int` data type is commonly used to represent integer values. On x86 architecture, `int` is typically 4 bytes.

Representation:

- Stored in little-endian format, where the least significant byte comes first.
- Example: Let's consider the value `42` (decimal).

```
| Byte 3 | Byte 2 | Byte 1 | Byte 0 |
|--------|--------|--------|--------|
| 00     | 00     | 00     | 2A     |
```

  The hexadecimal representation of `42` is `2A` in little-endian format.

Range:

- Signed `int` has a range from -2,147,483,648 to 2,147,483,647.

2. Floating-Point (`float`):

The `float` data type is used for single-precision floating-point numbers, typically 4 bytes on x86 architecture.

Representation:

- Follows IEEE-754 standard for floating-point representation.
- Consists of a sign bit, an 8-bit exponent, and a 23-bit fraction (mantissa).
- Example: Let's consider the value `3.14` (decimal).

```
| Sign | Exponent            | Mantissa             |
|------|---------------------|----------------------|
| 0    | 10000000001 (129)   | 10010010000111111011 |
```

The binary representation is `01000000010010010000111111011011`.

Range:

- Approximately ±3.4e38 with 7 significant digits of precision.

3. Double-precision Floating-Point (`double`):

The `double` data type is used for double-precision floating-point numbers, typically 8 bytes on x86 architecture.

Representation:

- Also follows IEEE-754 standard.
- Consists of a sign bit, an 11-bit exponent, and a 52-bit fraction (mantissa).
- Example: Let's consider the value `3.14` (decimal).

```
| Sign | Exponent                | Mantissa
|
|------|-------------------------|--------------------------------
------|
| 0    | 10000000000 (1023)      | 1001001000011111101101010100
01000 |
```

The binary representation is `0100000000001001001000011111101101101010001000000000000000000`.

Range:

- Approximately ±1.7e308 with 15 significant digits of precision.

4. Character (`char`):

The `char` data type is used to represent single characters, typically 1 byte on x86 architecture.

Representation:

- Stores the ASCII value of the character.
- Example: Let's consider the character `'A'` (ASCII value 65).

```
| Byte 0 |
|--------|
| 41     |
```

The hexadecimal representation is `41`.

Range:

- Signed `char` has a range from -128 to 127.
- Unsigned `char` has a range from 0 to 255.

5. Boolean (`bool`):

The `bool` data type is used to represent Boolean values, typically 1 byte on x86 architecture.

Representation:

- Usually represented as `0` for `false` and `1` for `true`.
- Example: Let's consider `true`.

```
| Byte 0 |
|--------|
| 01     |
```

The hexadecimal representation is `01`.

Range:

- `bool` can only take values `true` or `false`.

It's important to note that the actual sizes and representations may vary across different compilers and architectures, but the provided examples illustrate common representations on x86. Additionally, C++ offers fixed-size integer types (`int32_t`, `int64_t`, etc.) to ensure a specific size regardless of the platform.

Arrays

Arrays in memory are contiguous blocks of storage that allow for the storage of multiple elements of the same data type. The representation of arrays varies based on the data type, and memory allocation for arrays is typically done in a way that ensures efficient access to individual elements.

Representation of Arrays in Memory:

Let's consider arrays of different data types – `int`, `float`, `char`, and `double`.

1. Integer Array (`int[]`):

```
int arrInt[4] = {10, 20, 30, 40};
```

- Representation in memory (assuming 4-byte `int`):

Index	Memory Address	Value
0	0x1000	10
1	0x1004	20
2	0x1008	30
3	0x100C	40

2. Float Array (`float[]`):

```
float arrFloat[3] = {1.1, 2.2, 3.3};
```

- Representation in memory (assuming 4-byte `float`):

Index	Memory Address	Value
0	0x2000	1.1
1	0x2004	2.2
2	0x2008	3.3

3. Char Array (`char[]`):

```
char arrChar[5] = {'a', 'b', 'c', 'd', 'e'};
```

- Representation in memory (assuming 1-byte `char`):

Index	Memory Address	Value
0	0x3000	'a'
1	0x3001	'b'
2	0x3002	'c'
3	0x3003	'd'
4	0x3004	'e'

4. Double Array (`double[]`):

```
double arrDouble[2] = {5.5, 6.6};
```

- Representation in memory (assuming 8-byte `double`):

Index	Memory Address	Value
0	0x4000	5.5
1	0x4008	6.6

Memory Allocation for Arrays:

- Memory for arrays is allocated in a contiguous block.
- The starting memory address of the array is the address of the first element.
- Subsequent elements are located at consecutive memory addresses.
- The size of the array is the product of the number of elements and the size of each element.

Limits on Array Size:

- The size of an array in C++ is determined by the data type and the available memory.
- The language itself does not impose a strict limit on the size of an array.
- However, practical limits may be determined by factors such as available RAM and system architecture.
- Arrays that are too large might lead to stack overflow or segmentation faults.

Example:

```
// Declaration of a large array (the actual size depends on the syste
m)
int arrLarge[1000000];
```

It's essential to be mindful of the system's memory limitations and choose appropriate data structures for handling large amounts of data, such as dynamic data structures like linked lists or dynamic arrays.

In summary, arrays are represented in memory as contiguous blocks, and memory allocation is determined by the size and data type of the array. Practical limits on array size depend on the available system memory.

Address Calculation in Arrays

Address calculation in arrays involves determining the memory location of individual elements within the array. In C++, the memory address of an array element can be calculated using pointer arithmetic. Let's explore this concept in general and with C++ examples.

General Address Calculation in Arrays:

1. **Base Address:**
    - The memory address of the first element in the array is considered the base address.
2. **Element Size:**
    - The size of each element in the array is crucial for calculating the address of subsequent elements.
3. **Index:**
    - The index of the element indicates its position within the array.
4. **Address Calculation:**
    - The memory address of an element can be calculated using the formula:

```
Address_of_element_i = Base_address + (i * Size_of_each_elemen
t)
```

Where:

- `Base_address` is the memory address of the first element.
- `i` is the index of the element.
- `Size_of_each_element` is the size of each element in the array.

C++ Example of Address Calculation in Arrays:

```cpp
#include <iostream>

int main() {
    // Integer array
    int arrInt[5] = {10, 20, 30, 40, 50};

    // Base address of the array
    int* baseAddress = arrInt;

    // Calculate the size of each element
    size_t sizeOfElement = sizeof(arrInt[0]);

    // Calculate the address of the third element (index 2)
    int* addressOfElement2 = baseAddress + 2;



    // Calculate the byte offset using sizeof
    size_t byteOffset = 2 * sizeOfElement;

    // Print the details
    std::cout << "Base Address of arrInt: " << baseAddress << std::end
l;
    std::cout << "Size of each element: " << sizeOfElement << " bytes"
<< std::endl;
    std::cout << "Calculated address of arrInt[2]: " << addressOfEleme
nt2 << std::endl;
    std::cout << "Byte offset: " << byteOffset << " bytes" << std::end
l;
    std::cout << "Value at arrInt[2]: " << *addressOfElement2 << std::
endl;

    return 0;
}
```

Let's go through the important parts of the code:

```
// Base address of the array
int* baseAddress = arrInt;
```

- **Line 8:** Declares a pointer `baseAddress` of type `int*` and assigns it the address of the first element of `arrInt`. This is the base address of the array.

```
// Calculate the size of each element
size_t sizeOfElement = sizeof(arrInt[0]);
```

- **Line 11:** Uses the `sizeof` operator to calculate the size (in bytes) of each element in the array and stores it in the variable `sizeOfElement`.
  - `size_t` is used for `sizeOfElement` because it represents the size of a memory block and ensures compatibility across different platforms and architectures.
  - `size_t` is an unsigned integer type in C++ that is guaranteed to be able to represent the size of any object. It is commonly used for variables that hold sizes or indices of objects in memory.

```
// Calculate the address of the third element (index 2)
int* addressOfElement2 = baseAddress + 2;
```

- **Line 14:** Calculates the address of the third element (index 2) by adding `2 * sizeOfElement` to the base address. This demonstrates pointer arithmetic.

```
// Calculate the byte offset using sizeof
size_t byteOffset = 2 * sizeOfElement;
```

- **Line 17:** Calculates the byte offset by multiplying the index (`2`) with the size of each element (`sizeOfElement`). This provides the number of bytes to offset from the base address.
  - `size_t` is used for `byteOffset` because it represents the size of a memory offset in bytes, and using `size_t` ensures non-negativity and consistency with memory-related operations.
  - Using `size_t` is a good practice when dealing with sizes, indices, or memory-related calculations to enhance portability and maintain code consistency across platforms.

Important Notes:

1. **Pointer Arithmetic:**
   - Pointer arithmetic is used to calculate addresses in C++.
   - Adding an integer to a pointer advances it by that many elements.
2. **Array Indexing:**
   - Array indexing is a syntactic sugar for pointer arithmetic.
   - `arrInt[i]` is equivalent to `*(arrInt + i)`.
3. **Byte Offset:**
   - The actual memory address is calculated in bytes, so the size of each element is crucial for correct address calculation.

4. **Out-of-Bounds:**

    - Accessing elements beyond the array bounds is undefined behavior and should be avoided.

Use Cases of Arrays as Data Structures:

Arrays are versatile data structures widely used in various applications due to their simplicity and efficiency. Here are some common use cases, along with their advantages and disadvantages:

1. **Use Cases:**

    - **Sequential Access:** Arrays are efficient for sequential access, making them suitable for applications where data is accessed in a linear manner, such as iterating through a list of items.
    - **Random Access:** Arrays provide constant-time random access to elements based on their index, making them suitable for scenarios where quick access to any element is required.
    - **Fixed-Size Collections:** When the size of the collection is known and fixed in advance, arrays are a suitable choice due to their fixed size.

2. **Advantages:**

    - **Constant-Time Access:** Accessing any element in the array takes constant time, O(1), as it can be directly calculated using the index.
    - **Memory Efficiency:** Arrays are memory-efficient, as they store elements in contiguous memory locations, allowing for better cache locality.
    - **Simplicity:** Arrays are simple and easy to use, making them suitable for scenarios where complexity needs to be minimized.
3. **Disadvantages:**

    - **Fixed Size:** The main disadvantage is that arrays have a fixed size, making them less suitable for dynamic data that may grow or shrink.
    - **Insertion and Deletion:** Inserting or deleting elements in the middle of an array can be inefficient, as it may require shifting elements to maintain order.
    - **Wasted Memory:** If the array size is larger than the actual number of elements, memory may be wasted.

Operations and Time Complexity:

Operation	Time Complexity
Access by Index	O(1)
Search (Unsorted Array)	O(n)
Search (Sorted Array)	O(log n)
Insertion (at the End)	O(1)
Insertion (at Arbitrary Position)	O(n)
Deletion (from the End)	O(1)

Operation	Time Complexity
Deletion (from Arbitrary Position)	O(n)
Resize	O(n)

- **Access by Index:** Retrieving an element from a specific index is a constant-time operation, O(1).
- **Search (Unsorted Array):** Searching for an element in an unsorted array requires checking
- **Search (Sorted Array):** Searching in a sorted array can be more efficient, with a time complexity of O(log n) using binary search.
- **Insertion (at the End):** Inserting an element at the end of the array is a constant-time operation, O(1).
- **Insertion (at Arbitrary Position):** Inserting an element at an arbitrary position requires shifting subsequent elements, resulting in a linear time complexity, O(n).
- **Deletion (from the End):** Deleting an element from the end of the array is a constant-time operation, O(1).
- **Deletion (from Arbitrary Position):** Deleting an element from an arbitrary position requires shifting subsequent elements, resulting in a linear time complexity, O(n).
- **Resize:** Resizing an array involves creating a new array and copying elements, resulting in a linear time complexity, O(n).

Understanding these operations and their associated time complexities helps in choosing the right data structure based on the specific requirements of an application.

Dynamic Arrays

A dynamic array, also known as a resizable array or a dynamically allocated array, is a data structure that allows for flexible and efficient management of a collection of elements. Unlike static arrays, the size of a dynamic array can be changed during runtime, enabling the addition or removal of elements as needed.

In addition to general array features, key characteristics and concepts related to dynamic arrays include:

1. **Dynamic Memory Allocation:**
    - Dynamic arrays are created by allocating memory on the heap using mechanisms like `new` in C++.
    - This allocation allows for a more flexible size, as the memory is not fixed at compile-time.
2. **Resizable Capacity:**
    - Dynamic arrays have a variable capacity that can be adjusted to accommodate changes in the number of elements.
    - As elements are added, the capacity may be increased to prevent running out of space.

In languages like C++, the `std::vector` class is an example of a dynamic array implementation that encapsulates these concepts, offering a convenient and safe way to work with dynamic arrays. Understanding dynamic arrays is fundamental to efficient memory

management and data structure design.

**Implementation Example:

Here's a C++ code example demonstrating dynamic arrays performing resizing. The example includes detailed code explanations for each step.

```
#include <iostream>

int main() {
    // Initialize variables
    int* dynamicArray = nullptr;  // Pointer for dynamic array
    int size = 0;                 // Current size of the dynamic array
    int capacity = 5;             // Initial capacity

    // Allocate memory for the initial dynamic array
    dynamicArray = new int[capacity];



    // Function to print the elements of the dynamic array
    auto printArray = [&]() {
        std::cout << "Dynamic Array: ";
        for (int i = 0; i < size; ++i) {
            std::cout << dynamicArray[i] << " ";
        }
        std::cout << "\n";
    };

    // Function to resize the dynamic array
    auto resizeArray = [&](int newCapacity) {
        int* newArray = new int[newCapacity];  // Allocate a new array
with the new capacity

        // Copy elements from the old array to the new array
        for (int i = 0; i < size; ++i) {
            newArray[i] = dynamicArray[i];
        }
```

```
            // Deallocate memory for the old array
            delete[] dynamicArray;

            // Update the pointer to point to the new array
            dynamicArray = newArray;

            // Update the capacity
            capacity = newCapacity;
        };

        // Add elements to the dynamic array
        for (int i = 1; i <= 10; ++i) {
            // Check if resizing is needed
            if (size == capacity) {
                // Double the capacity when resizing is needed
                resizeArray(2 * capacity);
            }



            // Add the element to the dynamic array
            dynamicArray[size++] = i;

            // Print the dynamic array after each addition
            printArray();
        }

        // Deallocate memory for the dynamic array
        delete[] dynamicArray;

        return 0;
    }
```

Code Explanation:

1. **Memory Allocation:** The program starts by allocating memory for the initial dynamic array with a specified capacity (`5` in this case).
2. **Print Function:** A lambda function `printArray` is defined to print the elements of the dynamic array.
3. **Resize Function:** A lambda function `resizeArray` is defined to resize the dynamic array when its size reaches its capacity. It doubles the capacity and copies elements to the new array.
4. **Element Addition:** A loop adds elements to the dynamic array. If the size exceeds the current capacity, the array is resized using the `resizeArray` function.
5. **Printing After Each Addition:** The `printArray` function is called after each element addition to visualize the dynamic array's state.

6. **Memory Deallocation:** Finally, memory is deallocated for the dynamic array using `delete[]` to free the allocated memory.

Vectors

Here's a revised version of the code example used for dynamic arrays using vectors in C++. Vectors encapsulate dynamic arrays and provide a higher-level interface for dynamic resizing. The code explanations are provided for each step.

```cpp
#include <iostream>
#include <vector>

int main() {
    // Initialize a vector with an initial capacity of 5
    std::vector<int> dynamicVector;

    // Function to print the elements of the dynamic vector
    auto printVector = [&]() {
        std::cout << "Dynamic Vector: ";
        for (const int& element : dynamicVector) {
            std::cout << element << " ";
        }
        std::cout << "\n";
    };



    // Add elements to the dynamic vector
    for (int i = 1; i <= 10; ++i) {
        // Add the element to the dynamic vector
        dynamicVector.push_back(i);

        // Print the dynamic vector after each addition
        printVector();
    }

    return 0;
}
```

Code Explanation:

1. **Vector Initialization:** The program starts by initializing a `std::vector<int>` named `dynamicVector`. Vectors dynamically manage their memory and automatically resize when needed.
2. **Print Function:** A lambda function `printVector` is defined to print the elements of the dynamic vector.

3. **Element Addition:** A loop adds elements to the dynamic vector using the `push_back()` method. Vectors automatically handle resizing and memory management.
4. **Printing After Each Addition:** The `printVector` function is called after each element addition to visualize the dynamic vector's state.
5. **Memory Management:** Unlike the manual memory management required for dynamic arrays, vectors handle memory allocation and deallocation automatically.

By using vectors, the code becomes more concise and safer, as the vector class encapsulates the complexities of dynamic resizing and memory management. Vectors are a powerful and convenient alternative to dynamic arrays, providing a higher-level abstraction while maintaining

Vectors and Memory

The memory allocated to vectors in C++ is contiguous like arrays. This is because vectors are implemented as dynamic arrays, and the elements are stored in a contiguous block of memory. When a vector is created, it typically allocates an initial block of memory to store its elements. As elements are added to the vector and its size exceeds the allocated capacity, the vector dynamically reallocates memory to ensure sufficient space for additional elements. This dynamic reallocation may involve copying the existing elements to a new contiguous block of memory with a larger capacity.

Operations and Time Complexity

Operation	Time Complexity (Arrays)	Time Complexity (Vectors)
Access by Index	O(1)	O(1)
Search (Unsorted Array)	O(n)	O(n)
Search (Sorted Array)	O(log n)	O(log n)
Insertion (at the End)	O(1)	O(1) (amortized)*
Insertion (at Arbitrary Position)	O(n)	O(n)
Deletion (from the End)	O(1)	O(1)
Deletion (from Arbitrary Position)	O(n)	O(n)
Resize	O(n)	Amortized O(1)

This revised table includes a column for vectors, highlighting the differences in time complexity between arrays and vectors for various operations. Note that vector insertion at the end is stated as amortized O(1) because vectors may occasionally need to resize, leading to occasional O(n) insertions, but these are infrequent on average.
*Here amortize means 'on average'.

Reflection MCQs:

1. What is the primary focus of data types in programming languages?

- A. Organization of data in memory
- B. Representation of algorithms
- C. Design of data structures

- D. Handling of input/output operations
- **Correct Option: A**

2. In the context of C++, what is the purpose of the sizeof operator?

- A. Calculate the size of an array
- B. Determine the size of each array element
- C. Obtain the total number of elements in an array
- D. Retrieve the base address of an array
- **Correct Option: B**

3. How is the base address of an array represented in C++?

- A. int arrBase = &arr[0];
- B. int* arrBase = &arr;
- C. int* arrBase = arr[0];
- D. int* arrBase = arr;
- **Correct Option: D**

4. Which data type is commonly used for single-precision floating-point numbers in C++?

- A. float
- B. double
- C. int
- D. char
- **Correct Option: A**

5. What does the sizeof operator return in C++?

- A. Number of elements in an array
- B. Total size of an array
- C. Size of each element in bytes
- D. Base address of an array
- **Correct Option: C**

6. In C++, what is the range of a signed char?

- A. 0 to 255
- B. -128 to 127
- C. -32768 to 32767
- D. 0 to 127
- **Correct Option: B**

7. What is a major disadvantage of arrays in C++?

- A. Constant-time access
- B. Dynamic size
- C. Wasted memory
- D. Variable-size collections

- **Correct Option: C**

8. Which operation has a time complexity of O(log n) in a sorted array?

- A. Access by Index
- B. Search
- C. Insertion
- D. Deletion
- **Correct Option: B**

9. Which category of data structures includes trees and graphs?

- A. Linear Data Structures
- B. Non-Linear Data Structures
- C. Specialized Data Structures
- D. Advanced Data Structures
- **Correct Option: B**

10. What is the primary advantage of arrays in terms of memory efficiency?

- A. Dynamic size
- B. Wasted memory
- C. Contiguous storage
- D. Variable-size collections
- **Correct Option: C**

11. In C++, how is the random access time complexity for arrays characterized?

- A. O(1)
- B. O(log n)
- C. O(n)
- D. O(n^2)
- **Correct Option: A**

12. Which operation has a time complexity of O(n) in an unsorted array?

- A. Search
- B. Insertion
- C. Deletion
- D. Access by Index
- **Correct Option: A**

13. What is a limitation of arrays in C++ when it comes to dynamic data?

- A. Fixed size
- B. Constant-time access
- C. Efficient insertion
- D. Contiguous storage
- **Correct Option: A**

14. What is the time complexity for resizing an array in C++?

- A. O(1)
- B. O(log n)
- C. O(n)
- D. O(n^2)
- **Correct Option: C**

15. What does the base address of an array represent in C++?

- A. The value of the first element
- B. The index of the first element
- C. The memory address of the first element
- D. The size of the array
- **Correct Option: C**

Code Exercises:

Exercise 1: Array Operations

Write a C++ program that performs the following operations on an integer array:

1. Accepts user input to fill an array of size N.
2. Prints the elements of the array.
3. Calculates and prints the sum of all elements.
4. Finds and prints the maximum and minimum elements.
5. Searches for a specific element entered by the user and prints its index if found, otherwise prints a message indicating its absence.

```
#include <iostream>

int main() {
    const int N = 5;  // TODO: Change the size as needed
    int arr[N];

    // TODO: Implement code to fill the array with user input

    // TODO: Implement code to print the elements of the array

    // TODO: Implement code to calculate and print the sum of all elem
ents

    // TODO: Implement code to find and print the maximum and minimum
elements

    // TODO: Implement code to search for a specific element and print
its index if found

    return 0;
}
```

Exercise 2: Array Manipulation and Sorting

Implement a C++ program that does the following:

1. Accepts user input to fill two integer arrays of the same size.
2. Combines the two arrays into a third array, interleaving elements from the first and second arrays.
3. Prints the merged array.
4. Sorts the merged array in ascending order using any sorting algorithm.
5. Prints the sorted array.

```
#include <iostream>

int main() {
    const int N = 5;  // TODO: Change the size as needed
    int arr1[N];
    int arr2[N];
    int mergedArr[2 * N];

    // TODO: Implement code to fill arr1 and arr2 with user input

    // TODO: Implement code to combine the two arrays into mergedArr

    // TODO: Implement code to print the merged array

    // TODO: Implement code to sort the merged array in ascending orde
r

    // TODO: Implement code to print the sorted array

    return 0;
}
```

Exercise 3: Dynamic Array Resizing

Write a C++ program that dynamically resizes an integer array as follows:

1. Accepts user input to fill an initial array.
2. Prints the initial array.
3. Allows the user to enter additional elements to extend the array.
4. Prints the extended array after each addition.

```
#include <iostream>

int main() {
    // feel free to use vectors instead
    int* dynamicArray = nullptr;
    int size = 0;
    int capacity = 5;

    // TODO: Implement code to dynamically allocate memory for the ini
tial dynamic array

    // TODO: Implement code to print the initial array

    // TODO: Implement code to allow the user to enter additional elem
ents and extend the array

    // TODO: Implement code to print the extended array after each add
ition

    // TODO: Implement code to deallocate memory for the dynamic array

    return 0;
}
```

Exercise 4: Searching in a Sorted Array

Implement a C++ program that performs binary search on a sorted array:

1. Accepts user input to create a sorted integer array.
2. Prints the sorted array.
3. Accepts a target value from the user.
4. Performs binary search to determine if the target value is present and prints the result.

```
#include <iostream>

int main() {
    const int N = 5;  // TODO: Change the size as needed
    int sortedArray[N];

    // TODO: Implement code to fill the sorted array with user input

    // TODO: Implement code to print the sorted array

    int target;

    // TODO: Implement code to accept a target value from the user

    // TODO: Implement code to perform binary search and print the res
ult

    return 0;
}
```

Exercise 5: Memory Representation

Explore memory representation in C++:

1. Write a program that prints the memory addresses of elements in an integer array.
2. Modify the program to print the size of each element and the calculated memory address of each element.
3. Explore the use of pointers to access and manipulate array elements.

```
#include <iostream>

int main() {
    const int N = 5;  // TODO: Change the size as needed
    int arr[N];

    // TODO: Implement code to fill the array with user input

    // TODO: Implement code to print the memory addresses of elements
in the array

    // TODO: Implement code to print the size of each element and the
calculated memory address of each element

    // TODO: Implement code to explore the use of pointers to access a
nd manipulate array elements

    return 0;
}
```

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind