**Introduction to Data Structures:**

Data structures are an essential aspect of programming and software development. In simple terms, data structures refer to the organization and arrangement of data in a way that enables efficient operations on that data. These structures are primarily concerned with how data is stored and accessed in the main memory during program execution.

When we run a program, it consists of a set of instructions that manipulate and operate on data to produce results. Without data, these instructions serve no purpose. Therefore, data is a fundamental component of any program.

During program execution, both the program and the data it operates on are stored in the main memory. The main memory includes the central processing unit (CPU), which executes instructions, and the working memory or primary memory (RAM), which acts as temporary storage. On the other hand, permanent storage, such as a hard disk drive or external storage, holds program files and data files.

Let's consider an example to understand how data structures come into play. Suppose we take the popular application MS Word. When we run MS Word on a computer or mobile phone, the program is loaded into the main memory, and its instructions are executed by the CPU. As a result, we see the MS Word application interface and can start using it.

Now, if we want to open a document file in MS Word, the data from that file needs to be brought into the main memory for processing. This data could be text, multimedia content like images or videos, or any other form of information. Therefore, the program needs to organize and arrange this data in a way that allows for efficient utilization during its execution.

Data structures enable the efficient organization of data inside the main memory. Different data structures, such as arrays, linked lists, trees, or hash tables, can be used based on the requirements of the application. By designing appropriate data structures, the application can work more efficiently and process the data faster.

It's important to note that data structures are not limited to applications running on personal computers or mobile phones. They are integral to any system that deals with data, including databases and data warehouses.

Databases:

Databases are commonly used for managing commercial data in businesses. Commercial data refers to the information used in banks, retail stores, manufacturing farms, and other organizations. These databases store data in organized tables on the disk or hard disk drive.

When an application program accesses a database, the required data has to be brought into the main memory for processing. This retrieval of data from the disk to the main memory also necessitates the use of data structures to organize and arrange the data effectively.

Data Warehouse:

In businesses, data tends to accumulate rapidly, leading to large volumes of data. Operational data, used daily, and legacy data, which may not be frequently accessed, constitute commercial data. Legacy data, which acts as historical or archival data, can be stored on arrays of disks in what is known as a data warehouse. Data warehouses are used for analyzing business trends, making informed decisions, and formulating policies based on historical data.

Big Data:

With the advent of the internet, a vast amount of data has been generated and accumulated over time. This data, known as Big Data, encompasses information about various aspects such as people, places, and things. Analyzing this data can provide valuable insights for management, governance, and business purposes. Big Data involves the storage, management, and utilization of large-scale data sets and often requires specialized techniques and algorithms, such as data mining algorithms, to extract meaningful patterns and information.

To summarize, data structures are essential for efficient data manipulation during program execution. They facilitate the organization and arrangement of data in the main memory, enabling applications to process and operate on the data effectively. Databases, data warehouses, and Big Data represent different contexts where data structures are utilized to manage and utilize data efficiently.

**Title: Static and Dynamic Memory Allocation - Video Summary**

Video Overview:

The video provides an overview of static and dynamic memory allocation. It starts by introducing the concept of memory and its division into smaller addressable units known as bytes. The memory is represented as a block, with each byte having its unique address.

Size of Main Memory:

The presenter assumes a main memory size of 64 kilobytes (64 KB) for the discussion. Although modern computers use memory in gigabytes (GB), a smaller memory size is chosen for better understanding.

Divisions of Main Memory:

The main memory is divided into three sections utilized by a program: code section, stack, and heap. The code section holds the machine code of the program and its size varies depending on the program. The stack and heap sections are used dynamically during program execution.

Code Section:

The program's machine code is loaded into the code section of the main memory. Its size is not fixed and depends on the program being executed.

Stack and Heap Sections:

The remaining memory in the main memory is divided into the stack and heap sections. Typically, the stack is located at the higher memory addresses, while the heap is located at the lower memory addresses. However, for easier explanation, the presenter refers to the stack section as the higher portion and the heap section as the lower portion.

Stack Memory Allocation:

The video presents an example code with two variables: `integerVar` of type `int` and `floatVar` of type `float`. Assuming an integer occupies 2 bytes and a float occupies 4 bytes, the total memory required for the variables is 6 bytes. This memory is allocated on the stack.

Stack Frame and Activation Record:

The block of memory allocated for the variables inside the stack is referred to as the stack frame or activation record of the `main` function. The size of the memory required by a function is determined at compile time, making it a static memory allocation.

Dynamic Memory Allocation:

The video primarily focuses on static memory allocation. However, it briefly mentions dynamic memory allocation, where memory is allocated during runtime from the heap section. This allows for flexible memory management, but it is not covered in detail in the video.

Conclusion:

The video provides an introductory overview of memory allocation, specifically static memory allocation in the stack section. It explains the divisions of main memory, the allocation of memory for variables, and the concept of stack frames or activation records. It sets the foundation for further understanding of memory management in programming languages.

**Title: Memory Allocation in Stack and Heap - Detailed Explanation**

Introduction:

In this discussion, we explore the allocation of memory in the stack and heap for a sequence of function calls. The provided sample code illustrates the memory allocation process and highlights the distinction between the stack and heap memory.

1. Stack Memory Allocation:

- The stack is a section of memory used for static memory allocation and function call management.

- Activation records, also known as stack frames, are created on the stack for each function call.

- Activation records contain variables and parameters specific to each function call.

- Memory is allocated and deallocated automatically as functions are called and return.

- When a function is called, its activation record is added to the top of the stack.

- As each function finishes execution, its activation record is removed from the stack.

2. Memory Allocation Example:

- The sample code includes a main function, fun1, and fun2, with associated local variables and parameters.

- The main function allocates memory for variables a and b in its activation record.

- When fun1 is called from main, a new activation record is created for fun1 with the variable x.

- Upon calling fun2 from fun1, another activation record is added to the stack for fun2, containing variables i and a.

- The topmost activation record represents the currently executing function.

- As functions complete execution, their activation records are sequentially removed from the stack.

3. Stack Memory Behavior:

- The stack follows a last-in-first-out (LIFO) behavior, similar to a stack of items.

- Activation records are pushed onto the stack when a function is called and popped off when a function returns.

- The stack memory organization enables efficient memory management for function calls.

4. Heap Memory Allocation:

- Heap memory is used for dynamic memory allocation.

- Unlike the stack, memory allocation and deallocation in the heap are manual.

- Programs cannot directly access heap memory but must use pointers to access the allocated memory.

- Memory allocation in the heap is achieved using the "new" keyword in C++ or the "malloc()" function in C.

- The allocated memory remains allocated until explicitly deallocated using the "delete" keyword in C++ or "free()" function in C.

5. Treating Heap Memory as a Resource:

- Heap memory should be treated as a resource, similar to other system resources like a printer.

- Programs should request memory from the heap when needed and release it when no longer required.

- Failure to release heap memory can lead to memory leaks and eventual exhaustion of heap memory.

Conclusion:

Understanding memory allocation in the stack and heap is essential for efficient program execution. Stack memory is automatically managed and used for function calls, while heap memory allows dynamic allocation using pointers. By following good programming practices and correctly managing heap memory, memory leaks and wastage can be avoided, leading to more effective memory utilization in programs.

**Introduction to Physical Data Structures:**

- Physical data structures determine how memory is organized and allocated.

- The two main physical data structures are Arrays and Linked Lists.

- Arrays are directly supported by programming languages like C, C++, and Java.

- Arrays are a collection of contiguous memory locations, with fixed size.

- Arrays can be created either inside the stack or the heap.

- Arrays are suitable when the maximum number of elements is known in advance.

- Linked Lists are dynamic data structures composed of nodes.

- Each node contains data and a reference to the next node in the list.

- Linked Lists have a variable length and are always created in the heap.

- Linked Lists are suitable when the size of the list is unknown or can change dynamically.

Introduction to Logical Data Structures:

- Logical data structures determine how values are utilized and operations are performed on them.

- The main logical data structures include Stacks, Queues, Trees, Graphs, and Hash Tables.

- Physical data structures (Arrays and Linked Lists) are used to implement logical data structures.

- Stacks follow the Last In First Out (LIFO) discipline.

- Queues follow the First In First Out (FIFO) discipline.

- Trees are non-linear hierarchical structures composed of nodes and links.

- Graphs are also non-linear structures consisting of nodes connected by edges.

- Hash Tables are tabular data structures used for efficient key-value storage and retrieval.

Implementation of Logical Data Structures:

- Logical data structures are implemented using physical data structures (Arrays or Linked Lists).

- Throughout the course, each data structure will be learned and implemented using both arrays and linked lists.

- There are various subtopics within each data structure that will be covered in detail.

- Learning about arrays and linked lists will be the starting point of the course.

Note: It's important to note that the provided text is a script for a video or lecture on data structures. It is advisable to revise the content and structure it in a more organized and comprehensive manner when creating detailed notes.

**Abstract data type**

Abstract Data Type (ADT) is a concept in programming that combines data representation and operations on the data into a single unit. It is particularly relevant in object-oriented programming languages. ADTs allow us to define our own data types by hiding the internal details of how the data is represented and operated upon.

To illustrate the concept of ADT, let's consider an example of a list. A list is a collection of elements, and its representation requires three things: space for storing the elements, the maximum capacity of the list (capacity), and the number of elements currently in the list (size).

There are different ways to represent a list, such as using an array or a linked list. However, the choice of representation is not critical to understanding the concept of ADT. The focus is on the operations performed on the list.

Some common operations on a list include:

1. Add/Append: Adding an element to the end of the list.

2. Add/Insert: Adding an element at a specific index, requiring shifting of existing elements.

3. Remove: Removing an element at a given index, requiring shifting of remaining elements.

4. Set/Replace: Changing the value of an element at a given index.

5. Get: Retrieving the value of an element at a given index.

6. Search/Contains: Finding the index of a given element in the list.

7. Sort: Arranging the elements of the list in a specific order.

8. Reverse: Reversing the order of elements in the list.

9. Merge/Combine/Split: Operations involving multiple lists, such as merging or splitting them.

These operations demonstrate the functionality provided by the list ADT. When implementing an ADT, the internal details of how these operations are performed are hidden, allowing users to work with the ADT without concerning themselves with the underlying implementation.

In the context of data structures, ADTs are commonly used to represent various data structures like arrays, linked lists, stacks, queues, graphs, trees, and hash tables. By defining these data structures as ADTs, we encapsulate their data representation and operations, making them reusable and easily understandable.

In programming languages like C++ and others, ADTs are often implemented using classes. The class definition includes the data representation and the operations on the data, providing a complete ADT.

By understanding ADTs, you can work with higher-level data structures and focus on the functionality they offer, without needing to know the intricate details of their internal workings.

<Time and space complexity & asymptotic notation> --> see from DAA exercise book