Introduction to Data Structure

Unit 1

Review of Arrays:

Arrays are most frequently used in programming.
Mathematical problems like matrix, algebra and etc can be easily handled by arrays.

An array is a collection of homogeneous data elements described by a single name.

Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.

If an element of an array is referenced by single subscript, then the array is known as one dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two dimensional array and so on.

Analogously the arrays whose elements are referenced by two or more subscripts are called multi dimensional arrays.

Review of Class:

In C++ programming it is possible to separate program specific data types through the use of classes.

Classes define types of data structures and the functions that operate on those data structures.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.

Instances of these data types are known as objects and can contain member variables, constants, member functions, and operators defined by the programmer.

Syntactically, classes are extensions of the C struct, which cannot contain functions or overloaded operators. Following example shows the definition of class

Class member functions: A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.

Class access modifiers: A class member can be defined as public, private or protected. By default members would be assumed as private.

Constructor & destructor: A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

Pointer to C++ classes: A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

Pointer to structure

struct Books *struct_pointer;

Now you can store the address of a structure variable in the above defined pointer variable

struct_pointer = &Book1;

Need of Data Structures

The creation of software depends heavily on data structures and algorithms, which are core ideas in computer science. Data structures and algorithms are necessary for the following reasons:

- Data structures offer effective methods for storing and organizing data. To suit certain needs like quick insertion, deletion, or optimal memory use, several data structures are created. You may maximize performance and raise the general efficacy of your programs by selecting the appropriate data format for a certain job.

Need of Data Structures

- Algorithm Design and Analysis: Algorithms are systematic approaches to problem-solving. They define a solution's rationale and progression. You may assess an algorithm's effectiveness by understanding and evaluating its time complexity (how long it takes to execute) and space complexity (how much memory it needs). This study enables you to compare and choose the best method for a particular problem, resulting in quicker and more flexible solutions.

Need of Data Structures

- Data structures and algorithms offer a methodical method for solving problems. You can efficiently tackle difficult issues by decomposing them into smaller, more manageable components and then using the right data structures and methods. Strong problem-solving abilities are essential for software development since they allow you to handle a variety of difficulties and create reliable solutions.

- Performance optimization: Using effective data structures and algorithms will greatly increase the speed and responsiveness of your software programs. Even small increases in efficiency can have a significant impact on speed and resource usage in large-scale systems. Understanding data structures and algorithms can help you optimize crucial processes, speed up their completion, and use less resources.

Need of Data Structures

- Data structures and algorithms serve as a common language for software engineers to communicate with one another. Developers can utilize standardized data structures and methods to convey their thoughts clearly and concisely while discussing solutions or working on projects. The effective cooperation, code review, and maintenance of software systems are made possible by this shared vocabulary.

- Data structures and algorithms are routinely assessed in technical interviews for careers in software engineering. Candidates' problem-solving skills, computational thinking, and familiarity with basic data structures are evaluated during interviews. Your chances of succeeding in technical interviews and advancing your career in software development can be greatly increased by proficiency in these areas.

Characteristics of Data Structures

1. Storage and Organization: Data structures offer tools for the storage and organization of data in a certain format. They decide how information is accessibly stored in memory and how it may be changed.
2. Operations: A variety of operations, including insertion, deletion, retrieval, searching, sorting, and traversal, are supported by data structures. Depending on the data structure being utilized, these operations can have varying degrees of efficiency.
3. Memory Efficiency: By reducing the amount of space needed to store data, data structures can improve the use of memory. Some data structures do this by leveraging dynamic memory allocation or by encoding the data in a small amount of space.
4. Data structures specify the guidelines and processes for granting and removing access to data. Whether accessing items sequentially or randomly, they influence how well they may be accessed.
5. Mutability: Some data structures allow for dynamic changes to the stored data, such as adding or removing elements, while others are immutable and cannot be modified after creation.

Data Structure

- *Data Structure* can be defined as the group of *data* elements which provides an efficient way of storing and organising *data* in the computer so that it can be used efficiently. Some examples of *Data Structures* are arrays, Linked List, Stack, Queue, etc.
- Data structures are used in all areas of computer science such as Artificial Intelligence, graphics, Operating system etc
- For eg: the data structure QUEUE is used by printer for printing its assigned jobs

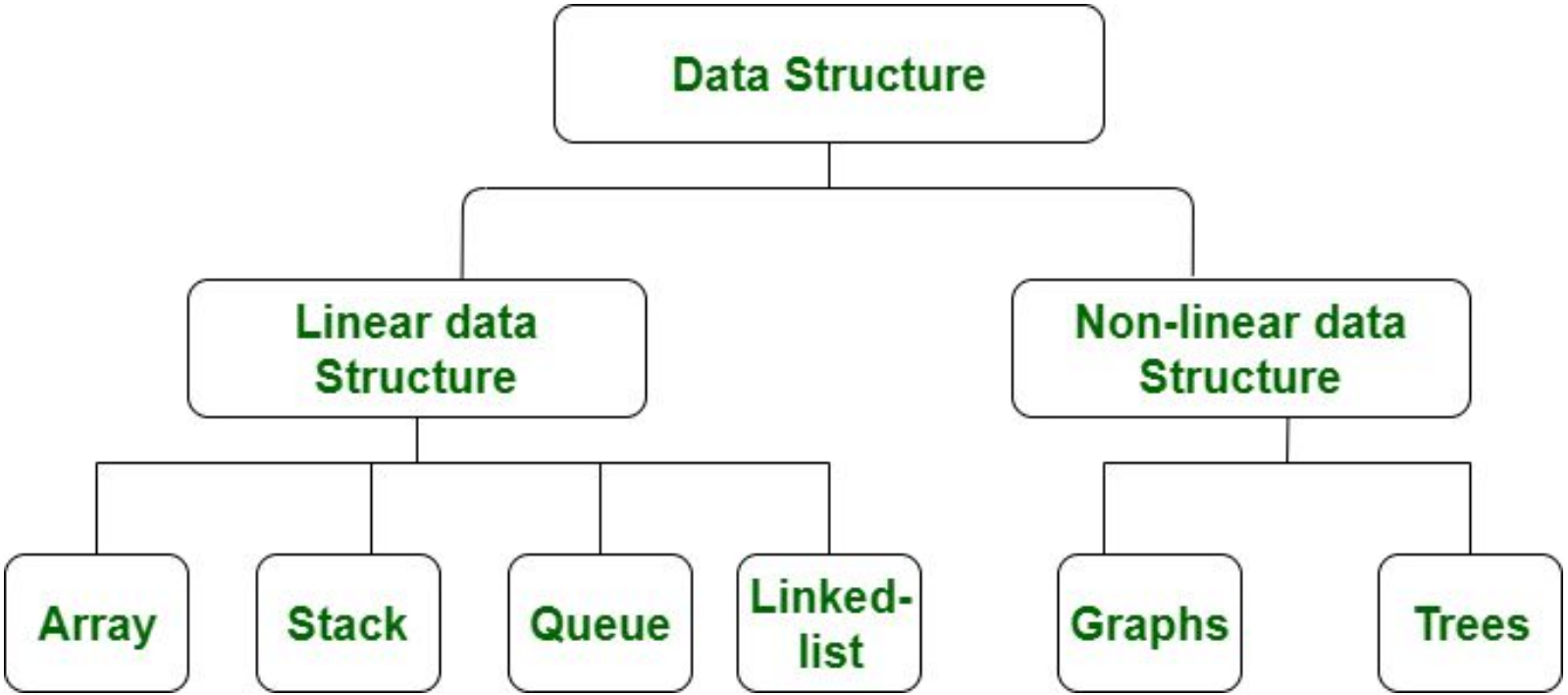
Data Structure (cont.)

Choice of data model depends on two considerations.

It must be rich enough in structure to mirror the actual relationship of the data in the real world.

The structure should be simple enough that one can effectively process the data when necessary.

Types or Classification of Data Structure



1. Arrays: Arrays are a fundamental data structure that stores elements of the same type in contiguous memory locations. They provide constant-time access to elements but have a fixed size.
2. Linked Lists: Linked lists are data structures consisting of nodes, where each node contains data and a reference to the next node. They allow dynamic insertion and deletion of elements but have slower access time compared to arrays.
3. Stacks: Stacks follow the Last-In-First-Out (LIFO) principle, where elements can be added or removed only from one end. They are commonly used in programming languages for function calls, recursion, and expression evaluation.
4. Queues: Queues adhere to the First-In-First-Out (FIFO) principle. Elements are added at the rear and removed from the front. Queues are used in scenarios such as process scheduling, task management, and breadth-first search.
5. Trees: Trees are hierarchical data structures composed of nodes. Each node can have child nodes, forming a parent-child relationship. Trees are used in various applications like file systems, database indexing, and representing hierarchical relationships.
6. Graphs: Graphs consist of a set of vertices (nodes) and edges that connect these vertices. They are used to represent networks, social connections, transportation systems, and more. Graphs can be directed (edges have a specific direction) or undirected (edges have no direction).
7. Hash Tables: Hash tables (also known as hash maps) provide fast data retrieval by mapping keys to values using a hash function. They are efficient for search, insertion, and deletion operations and are widely used for caching, indexing, and data lookup.

Linear Data Structure

Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a **linear data structure**.

In linear data structure, single level is involved.

Therefore, we can traverse all the elements in single run only.

Linear data structures are easy to implement because computer memory is arranged in a linear way.

Its examples are array, stack, queue, linked list, etc.

Non-Linear Data Structure

Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**.

In a non-linear data structure, single level is not involved.

Therefore, we can’t traverse all the elements in single run only.

Non-linear data structures are not easy to implement in comparison to linear data structure.

It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

Abstract Data Type(ADT)

A data type is a collection of values and set of operation on those values.

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.

It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

It is called abstract because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction

Abstract Data Type(ADT)

That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software data structure.

The term “Abstract Data type” refers to the basic mathematical concept that defines the data type.

Formally, An abstract data type is a data declaration packaged together with the operations that are meaningful on the data type.

ADT

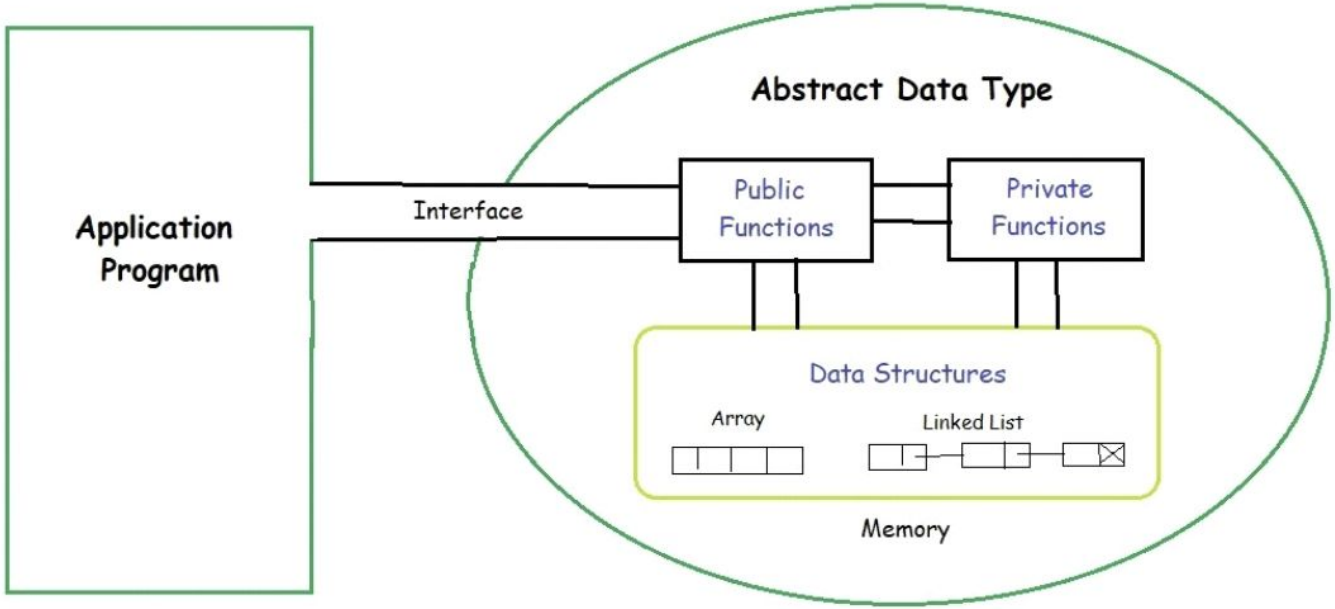

Application Program
Interface
Abstract Data Type
Public Functions
Private Functions
Data Structures
Array
Linked List
Memory

Fig : ADT

Divide and Conquer Algorithm

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems.

These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Ø Divide the problem into a number of sub problems.

Ø Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.

Ø Combine the solutions to the sub problems into the solution for the original problem.

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Ø Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

Ø Conquer: Sort the two subsequences recursively using merge sort.

Ø Combine: Merge the two-sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.

To perform the merging, we use an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that $p \le q < r$.

The procedure assumes that the subarrays A[p □ q] and A[q + 1 □ r] are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray A[p □ r].

Greedy Algorithm

A greedy algorithm is an approach to problem-solving that involves making locally optimal choices at each step with the hope that these choices will lead to a globally optimal solution.

Greedy algorithms are particularly useful for optimization problems where you're trying to find the best possible solution from a set of available choices.

Greedy Algorithm

Greedy algorithms are efficient and easy to implement, but they may not always yield the globally optimal solution. The locally optimal choices might lead to suboptimal results in some cases.

Few examples that use Greedy Algorithm

Huffman Coding: Given a set of characters and their frequencies, create a binary tree that minimizes the total encoding length.
Prim's Algorithm: Find the minimum spanning tree of a connected, undirected graph.
Dijkstra's Algorithm: Find the shortest paths from a single source vertex to all other vertices in a weighted graph.

Backtracking

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Backtracking can also be said as an improvement to the brute force approach. So basically, the idea behind the backtracking technique is that it searches for a solution to a problem among all the available options. Initially, we start the backtracking from one possible option and if the problem is solved with that selected option then we return the solution else we backtrack and select another option from the remaining available options.

Backtracking

There also might be a case where none of the options will give you the solution and hence we understand that backtracking won't give any solution to that particular problem.

We can also say that backtracking is a form of recursion.

This is because the process of finding the solution from the various option available is repeated recursively until we don't find the solution or we reach the final state.

So we can conclude that backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.

Algorithm Analysis

2. Best Case Analysis (Very Rarely used)

In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Omega(1)$

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Growth Rate

We try to estimate the approximate computation time by formulating it in terms of the problem size N. If we consider that the system dependent factor (such as the compiler, language, computer) is constant; not varying with the problem size we can factor it out from the growth rate. The growth rate is the part of the formula that varies with the problem size. We use a notation called O-notation ("growth rate", "big-O"). The most common growth rates in data structure are:

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as :

1. Constant time O(1)
2. Logarithmic time Olog(n)
3. Linear time O(n)
4. Polynomial time $O(n^c)$
5. Exponential time $O(c^n)$

If we calculate these values we will see that as N grows log(N) remains quite small and Nlog(N) grow fairly large but not as large as N^2 . Eg. Most sorting algorithms have growth rates of Nlog(N) or N^2 . Following table shows the growth rates for a given N.

N	Constant	log(N)	Nlog(N)	N^2
1	1	0	0	1
2	1	1	2	4
4	1	2	8	16
8	1	3	24	64
32	1	5	160	1024
256	1	8	2048	65536
2048	1	11	22528	4194304

Estimating the growth rate

Algorithms are developed in a structured way; they combine simple statements into complex blocks in four ways:

Ø Sequence, writing one statement below another

Ø Decision, if-then or if-then-else

Ø Loops

Ø Subprogram call

Let us estimate the big-O of some algorithm structures.

Simple statements: We assume that statement does not contain a function call. It takes a fixed amount to execute. We denote the performance by O(1), if we factor out the constant execution time we are left with 1.

Sequence of simple statements: It takes an amount of execution time equal to the sum of execution times of individual statements. If the performance of individual statements are O(1), so is their sum.

Decision: For estimating the performance, then and else parts of the algorithm are considered independently. The performance estimate of the decision is taken to be the largest of the two individual big Os. For the case structure, we take the largest big O of all the case alternatives. *Simple counting loop*: This is the type of loop in which the counter is incremented or decrement each time the loop is executed (for loop). If the loop contains simple statements and the number of times the loop executes is a constant; in other words, independent of the problem size then the performance of the whole loop is O(1). On the other hand if the loop is like

Eg: *for (i=0; i< N; i++)*

The number of trips depends on N; the input size, so the performance is O(N).

Nested loops: The performance depends on the counters at each nested loop. For ex:

for (i=0; i< N; i++) {

for (j=0; j< N; i++) {

sequence of simple statements

} }

$$\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}1=\sum_{i=0}^{N-1}N=N\sum_{i=0}^{N-1}1=N^2$$

The outer loop count is N but the inner loop executes N times for each time. So the body of the inner loop will execute N*N and the entire performance will be $O(N^2)$.

```
for (i=1; i<=N; i++) {
        for (j=0; j< i; j++) {
                sequence of simple statement} }
```

$$\sum_{i=1}^{N}\sum_{j=0}^{i-1}1=\sum_{i=1}^{N}i=\frac{N(N+1)}{2}=\frac{N^2}{2}+\frac{N}{2}$$

In this case outer count trip is N, but the trip count of the inner loop depends not only N, but the value of the outer loop counter as well. If outer counter is 1, the inner loop has a trip count 1 and so on. If outer counter is N the inner loop trip count is N.
How many times the body will be executed?

1+2+3+.....(N-1)+N = N(N+1) / 2 = ((N_2) +N)/2

Therefore the performance is O(N_2).

Generalization: A structure with k nested counting loops where the counter is just incremented or decrement by one has performance O(N_k) if the trip counts depends on the problem size only.

Popular Notations in Complexity Analysis of Algorithms

1. Big-O Notation

We define an algorithm's **worst-case** time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires to consider all input values.

Popular Notations in Complexity Analysis of Algorithms

2. Omega Notation

It defines the **best case** of an algorithm's time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires to consider all input values.

3. Theta Notation

It defines the **average case** of an algorithm's time complexity, the Theta notation defines when the set of functions lies in both **O(expression)** and **Omega(expression)**, then Theta notation is used. This is how we define a time complexity average case for an algorithm.