MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR OPEN & DISTANCE EDUCATION

Data Structure

Bachelor of Computer Applications (BCA)
Semester - 3

SELF LEARNING MATERIAL

MATS UNIVERSITY
www.matsuniversity.ac.in

NAAC GRADE A+ ACCREDITED UNIVERSITY

Bachelor of Computer Applications

ODL BCA DSC-07

Data Structure

COURSE DEVELOPMENT EXPERT COMMITTEE

Prof. (Dr.) K. P. Yadav, Vice Chancellor, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Sanjay Kumar, Professor and Dean, Pt. Ravishankar Shukla University, Raipur, Chhattisgarh

Prof. (Dr.) Jatinder kumar R. Saini, Professor and Director, Symbiosis Institute of Computer Studies and Research, Pune

Dr. Ronak Panchal, Senior Data Scientist, Cognizant, Mumbai

Mr. Saurabh Chandrakar, Senior Software Engineer, Oracle Corporation, Hyderabad

COURSE COORDINATOR

Dr. Abhishek Guru, Associate Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

COURSE PREPARATION

Dr. Abhishek Guru, Associate Professor and Ms. Tanuja Sahu, Assistant Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

March, 2025

ISBN: 978-93-49916-13-5

Printed & Published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. Meghanadhudu Katabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

Data structures are the backbone of efficient algorithm design and software development. This course provides a comprehensive understanding of fundamental data structures such as arrays, stacks, linked lists, trees, and graphs. Students will gain both theoretical knowledge and practical skills in implementing and utilizing these structures to optimize data storage, retrieval, and processing.

Module 1: Introduction to Data Structure

Data structures are essential for organizing and managing data effectively in computer science. This Unit introduces the concept of data structures, their classification (linear and non-linear), and their significance in problem-solving and algorithm efficiency. Students will learn how to choose appropriate data structures based on computational requirements.

Module 2: Array

Arrays are one of the simplest yet most widely used data structures for storing elements sequentially. This Unit covers array types, operations (insertion, deletion, searching, and sorting), and their applications. Students will understand the advantages and limitations of arrays in comparison to other data structures.

Module 3: Stack

Stacks follow the Last In, First Out (LIFO) principle and are used in scenarios such as expression evaluation, function calls, and backtracking algorithms. This Unit explores stack implementation using arrays and linked lists, stack operations (push, pop, peek), and applications in recursion and memory management.

Module 4: Linked List

Linked lists provide a dynamic way to store and manage data, overcoming the limitations of arrays. This Unit covers singly, doubly, and circular linked lists, along with operations such as insertion, deletion, traversal, and searching. Students will learn the significance of linked lists

in dynamic memory management and real-world applications.

Module 5: Tree and Graph

Trees and graphs are advanced non-linear data structures used in hierarchical and network-based applications. This Unit introduces binary trees, binary search trees (BST), tree traversals, and graph representations (adjacency matrix and adjacency list). Students will explore algorithms for tree and graph traversal, including BFS (Breadth-First Search) and DFS (Depth-First Search).

MODULE 1
INTRODUCTION TO DATA STRUCTURE

LEARNING OUTCOMES

- Understand the definition and classification of data structures.
- Learn about different types of data structures such as arrays, linked lists, queues, stacks, trees, and graphs.
- Understand C++ memory management, including new and delete operators.
- Learn about performance analysis of data structures, including time complexity and space complexity.

Unit 1: Introduction to data structure

1.1 Introduction - Definition, Classification of Data Structure

Structures allow programmers to address intricate questions with neat solutions, whether it is through basic arrays or intricate graphs. Which data is stored can have a considerable effect on both the performance and functionality of software. The structure in Data structure is the core of computing today an organized way of storing and scalable software systems. by every application in software, database systems, operating systems, and algorithms to work efficiently. Data structures are essential to writing efficient algorithms, code performance optimization, and building academic subject; they have immense real-world applications. Well-designed data structure is needed Data structures are not just an strengths, limitations, and appropriate use cases for various data structures can help developers make informed decisions that will greatly affect the performance and correctness of their software solutions.

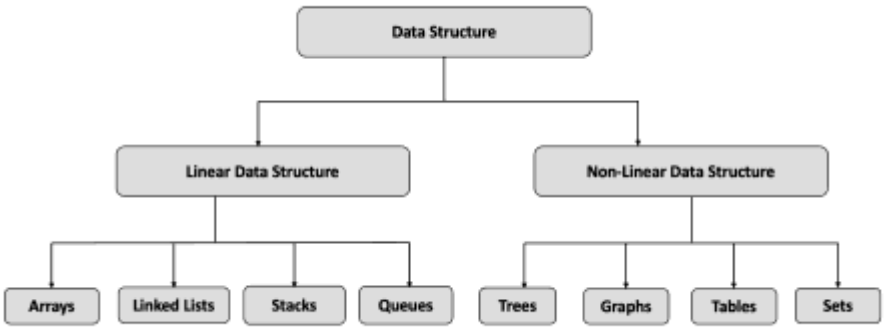

Figure 1.1: Data Structure Type
[Source: https://technologystrive.com/]

Definition of Data Structures

Structure of data values, how they relate to each other, and what kind of function or action can be used on the data. Because there is so much data available now, it needs to be organized, processed, retrieved, and stored quickly and easily. It's a data structure is an arrangement for data. or mathematical models followed by the organization based on which they can efficiently access and modify the data present in databases. It is precisely the logical etc,). The elements can be numbers, characters, or even What are Data

Structures at a high level, a data structure combines data elements based on the set of relationships between them (e.g. hierarchical, linear, In theory, that method works well. algorithm by making it easier to work with data, letting access patterns be faster, or lowering the amount of memory that is used. The opposite is also true: a bad choice of data format can slow down an algorithm and the data it works on. A good data format can make an algorithm much more effective. Data structures connect algorithms to the things that are being changed. a first-in, first-out relationship, and a tree structure show how parts are related in a hierarchical way. These connections show how to get data, add data, or remove data from the data. As an example, a queue data structure sets up Data structures are more than just places to store information; they also show the logical connections between the operations of deleting, traversing, searching, sorting, and merging. Also, list the actions that can be done on the data. Putting in (adding), Data structures explain what operations can be done on data and what they mean. Data structures, on the other hand, describe how these operations are carried out in memory, or how they are kept. of the first week talks about data structures and abstract data types (more on ADTs later), both in a basic and an advanced way. ADT stands for the second set of data numbers. Structure of Data and Different kinds of it file structures Categories help programmers choose the right kind of data format for the type of data they need for their problem or use case. Data structures are groups of the main categories. Let's look at it in various ways. Putting data types into these groups is important.

Structures for linear data

When you use a linear data format, the calculations are the facts are set up in a way that is either sequential or linear. Every node is connected to both node before it and the node next to it. These plans are easy to understand and use. To give some cases, another name for an array is basic data structure that holds a group of things that are all same type and size. A block of information that is all connected to each other. This makes it possible to reach the elements in constant time. One more important distinction to note between Lists and Arrays is that the latter are very efficient for scenarios in which the usage of retrieval operations on an array element occurs quite frequently, whereas Lists are not. But one important drawback is their

self-contained size, limiting their ability to resize dynamically. Creating a new larger array and copying all elements into it when an array reaches its capacity is an expensive operation in time and memory. Even though they have this problem, arrays are still widely used in many situations, including sorting algorithms, search methods, and matrix models. Arrays are often used when random access is needed because direct element access is easy and works well. A linked list, on the other hand, is a flexible way to store a group of things. It's called a linked list. It has data field &link to next structure in the chain. This is called a node. Because it doesn't have to rotate parts like an array does, it's made so that adding and removing items is quick. Simple changes to a few pointers are all it takes to add or remove a part. The problem with linked lists is that they need more memory for pointers, which means they need more space overall. Also, to get an item from a linked list, we have to start at the beginning of the structure, which takes a set amount of time. Linked lists, on the other hand, are a useful data structure that is often used in situations where parts need to be added and removed quickly, like dynamic memory allocation, graph representation, and building stacks and queues. To make stacks, which are type of data structure, you use the Last-In-First-Out (LIFO) rule. This means that first thing that was taken away is the last thing that was added. You can only do two main things with a stack: push and pop. Push adds an item to the top of the stack, and pop takes the item off the top. A stack is often used when the order of things is important, like when managing names for function calls, evaluating expressions, reversing algorithms, and so on. In this case, the system saves an implicit stack to which the state of the function execution is pushed whenever a recursion call is made. One way stacks are used is to make sure that operators and operands are processed in the right order in formulas. Because adding and removing items can only happen on one end of a stack, and accessing and changing items can only happen on one end as well, it is usually a fast and efficient structure. However, it can't be used for jobs that need random access and manipulation.

In computers, queues usually work with "First In, First Out" (FIFO) rule, which means that things are added at back & taken away from front. There are times when applications need to make sure that the oldest job is handled first, and this can help with that. As an example,

an operating system uses queues to schedule tasks. Web sites use them to handle requests. Printer job queues use queues for jobs to be printed. Different types of queues, such as circular queues, priority queues, dequeues (double-ended lines), and more, each have their own benefits. Applications. Dequeue queues can be implemented from both ends and circular queues can use previously occupied memory of removed elements, and priority queues provide additional operations for defueling based on priority rather than enquire order. In contrast, deques allow insertions and deletions at both ends, offering more versatility. Even though there are additional variations, the main FIFO principle remains vital in situations that needed execution of tasks in the exact order. All these data structures have their own significance in terms of usages in various functionalities based upon their strength and weaknesses. Do you know what their fundamental operations and performance characteristics are? Arrays are great for accessing elements with an index, linked lists are good at allowing you to change your size and insert elements or delete them quickly, stacks allow you to access elements in a controlled way based on Last In First Out (LIFO) principle, & queues enable you to keep your tasks in line and process them by the First In First Out (FIFO) method. Choosing the right data structure according to what they need can help the developers in writing more efficient and maintainable code. Linear data structures help in storing data in order in which they are supposed to be accessed or will have a sequential relationship between as well. They are a fact easier to implement and to understand, however it may not be the optimum solution in case you are going to be dealing with complex relationships or in case your access patterns will not be very straight forward.

Non-Linear& Data Structures

Non-linear data structures do& have a sequential arrangement of elements. Instead of each element having a target, it can connect too many other elements, creating complex relationship. These are more flexible but can also be slightly harder to implement and explore. Trees are one of the basic data structures in & computer science. A tree is a structure where you have a root node with child nodes cascading out of it. In a tree, each node may have multiple child nodes but only one parent node (the root node has no parent). Examples of hierarchical relationships where trees are commonly

used include file systems, organizational structures, and database indexes. The main advantages of trees are their better traversal, insertion and deletion. This guarantees that the leftmost child node will always hold a lesser value compared to the rightmost child node of any given tree node. It means that search operations are greatly optimized, the complexity of this is o (log n) in a balanced tree. Arguments: Trees are also used in computer graphics, ai (decision trees) and compilers (syntax trees) etc. Other significant variants include the balanced tree, like AVL and Red-Black trees. In addition to that, trees also forms the base of network routing algorithms, game development (through mini max trees) and even for encryption algorithms. Graphs & non-hierarchical data& structures made up of nodes (vertices) connected by edges. Trees are hierarchical structures. If the edges have directed edges, then these types of graphs are called directed graphs, otherwise undirected. Graphs are particularly popular for modelling relationships & networks of objects in real life, such as social networks, transportation networks, and linking structures of web pages. A graph can be represented as an adjacency matrix and adjacency list, where time and space complexity is different but have their advantages. Algorithms like DFS (Depth-First Search) & BFS (Breadth-First Search) efficiently search through graph structures to find the simplest paths, find cycles, and analyse networks. Dijkstra's algorithm and the Bellman-Ford algorithm are often used to find the quickest path in a weighted graph. Floyd-Wars hall’s algorithm and Prim's algorithm are more specialized and are used to make networks work better. Today's technologies, like search engines, recommendation systems, and artificial intelligence, are all based on new structures. But graph theory is at the heart of all of them, which is why it's the most important data structure to study. Graphs are also used to describe things that happen in the real world, like disease spread modelling in epidemiology, syntax parsing in linguistics, and molecular structure representation in chemistry. This gives them even more uses. For each node in a heap, value of its children is either greater than or equal to (a max heap) or less than or equal to (min heap). Heaps are made to work like trees while still being heaps. In some stacks, the parent nodes are the same size as or bigger than the children nodes. The parent node is always bigger than or the same size as its children in these heaps. In computer science, heaps, also called

heap structures, are a type of tree-based data structure. Structures that satisfy the heap property; that is, for a max heap, the key of a parent node is for a max-heap, the key is less than or equal to the keys of its children. For a min-heap, the key is greater than or equal to the keys of its children. That means that the actions to insert & delete take O(log n) time. Because it uses a binary heap data format, heap sort is a fast way to sort data. Another important use for heaps is in graph algorithms, especially Dijkstra's shortest path method, which uses them to efficiently extract the minimum distance vertex. Heaps are also utilized in real-time event-driven systems, as well as for scheduling in operating systems and in data compression algorithms, such as Huffman coding. Their methodical approach to prioritization renders them essential in situations where order and efficiency are paramount. Hash tables are also known as hash maps, which are data structures that allow you to access elements quickly via a key-value mapping mechanism. A hash is a function that computes an index in an array, that's where the value is stored. This allows for efficient search, insertion, & deletion operations, which usually have an average time complexity of O (1). Because hash tables can store and retrieve data very quickly, they are often used for database indexing, caching mechanisms, and associative arrays. This topic would refer to one of the key issues in the implementation of hash tables, which is collision resolution, which occurs when you have two the hash number is the same for all keys. Chaining (linked lists) and Linear probing, quadratic probing, double hashing, and other types of open addressing.) Are two methods used for collision resolution? Hash tables are the backbone of many modern needs, such as storing password (using functions like SHA-256), compiler symbol tables, and even network routing (through hash-based load balancing). This feature makes it the first choice for any high-performance computing application that requires key-based lookups in O (1) time. Furthermore, they are a critical component in cyber security, data deduplication, and large-sized distributed systems, where fast information retrieval is important for efficiency and scalability. To sum up, trees, graphs, heap, and hash tables are fundamental data structures upon which many algorithmic processes and applications are built. Trees are a practical tool used for dealing with hierarchical data or optimizing search operations, and they have become the very

backbone of databases, compilers, and artificial intelligence. Graphs and Their Applications Graphs are used to represent complex relationships in many fields, including networks, search engines, and social media. Heaps are vital in work schedules, graph Dakota and sorting methods due to their efficient prioritization! With their fast key-based access, hash tables enable everything from database indexing to caching to cyber security. Knowing these data structures & their practical applications allows you to optimize algorithms and build applications more efficiently. Their importance is not limited to theoretical computer science; they have had a profound impact on modern technology and problem-solving approaches. Listing non-linear data structures is better for portraying complex relationships and allowing more advanced operations. They form a foundational base for solving an array of problems, including hierarchies, networks, mappings, and priority-based access.

1.2 Description of Various Data Structures - Array, Linked List, Queue, Stack, Tree, Graph

Static Data Structures

Static data structures have fixed extent, their size allocated at compile time. You cannot change their size during runtime, which means that you need to know the maximum size and declare it in advance. Data structures form a vital part of computer science and programming, as they are the basis of how information is stored and accessed. The two main categories divide them into static data structures and dynamic data structures, both with distinct features, properties, pros, and cons. These classes have great significance in the construction of efficient algorithms and the optimization of memory. Static data structures are those with a fixed (invariant) size. This trait is exemplified in traditional arrays, such as in C and Java, which require you to specify the size when creating the array. What gives the static data structures an advantage over dynamic data structures is their memory allocation is predictable, leading to quick access times, followed by efficient indexing. Elements of static array are stored at contiguous memory addresses, which guarantee that accessing a specific index takes always constant time. This characteristic makes static arrays exceptionally valuable in situations where quick access to data is essential. But static data structures have their own significant disadvantages. Their static size can cause inefficiency in memory

usage. Memory wastage is when more space is allocated than the actual number of items. This means that if the amount of data you try to process as a list exceeds the memory for that list, the program is unable to maintain more elements which can lead to failures or other avoidance steps such as copying the data from one array to others. Another example of a static data structure is matrices which are fixed size multi-dimensional arrays. These structures are commonly used in scientific computing, image processing, and graph algorithms, where data must be organized in a tabular form. However, while matrices are useful, they also suffer from the same limitations as traditional arrays, which means that memory allocation and efficiency must still be carefully considered. Static structures have many shortcomings, which can be addressed by using dynamic data structures and allocating memory at runtime. Dynamic Data Structure vs. Static Data Structure: dynamic data & structure can grow and shrink as needed during game play, which can lead to better memory use, among other things. They are especially useful in applications where the data is dynamic and fills an unpredictable size of time window. Things like Array List in Java and vector in C++ (dynamic arrays) show this kind of flexibility. These structures are not like traditional arrays, their size is adjusted automatically when their capacity is full and new elements have arrived to be added to it. Once array is full, new array that is bigger is created, & current elements are copied to the new memory location. This resizing strategy enables dynamic arrays to provide a good compromise between efficiency and scalability at the expense of some overhead incurred during reallocation, which may be infrequent and is amortized over multiple insertions.

Another of the basic dynamic data structures is linked lists. Arrays are made up of a Sequence of data elements, while linked lists are made up of nodes. Each node is a data element that has a pointer (or reference) to the next node. This allows for fast insert and deletes operations as new elements can simply be inserted and removed from memory without increasing or decreasing the index values of other elements. In fact, one of the main reasons to use linked lists is to make up data & structure where number of elements is unknown ahead of time or should not stay the same and should be changed often. But their use of pointers comes with extra memory overhead and possible cache inefficiency as compared to arrays Dynamic data structures can

be further represented by more complex forms, such as trees and graphs. Popular tree structures include binary search trees (BSTs), heaps, and quad trees, which facilitate efficient searching, insertion, and deletion operations; indeed, they are crucial for tasks such as database indexing, search algorithms, and representing hierarchical data. Graphs (composed of nodes (vertices) connected by edges) are used extensively in social networks, route optimization, and network flow analysis. These structures are flexible and can expand as required by adding nodes and edges, without any bounds on how big they can grow in theory. Remember, your data structure is the fundamental backbone of your application, and how you decide to structure it really comes down to what you're doing. Static structures are predictable in memory usage and fast access but less flexible, while dynamic structures are flexible and utilize memory efficiently. With this knowledge, programmers can create an effective and efficient software system. Dynamic structures allow for flexibility and also possibly efficient use of memory; however, they incur overhead for allocation & deal location of memory, and can be more complex to implement.

Homogeneous vs. Heterogeneous Data Structures

Homogeneous data structure: If all the data & elements in data structure are of the same data types. Because of this uniformity, there is type safety, which implies that operations on these data structures are less likely to be erroneous. Moreover, the homogenous nature of the data structures can lead to more efficient usage of memory. An example of a homogeneous data structure is an array in strong typed programming languages like Java and C; it is an array which stores elements of only one type. Once an array is declared for integers for instance, it will not store floating-point numbers or characters. This limitation ensures guaranteed behaviour and reduces the chances of runtime errors. More examples of homogeneous data structures are vectors, which are dynamic arrays that pass elements of the same type. A vector is a dynamic array; allow resizing dynamically with same type. This makes them helpful in situations where the exact element count is not known when they are declared, but type uniformity is still important. Consistent data structures allow for better performance characteristics and can give better memory allocation patterns as they can keep elements stored close to each

other in memory resulting in better cache locality when accessing elements. Furthermore, they improve performance by eliminating overheads on type checks at runtime making them a great fit for systems programming and high-performance applications. Heterogeneous data structures allow elements of different data types to be stored in the same data structure, which gives more flexibility. This adaptability is especially helpful in cases where complex data entities need to be represented. The main disadvantage of heterogeneous data structures is that type checking must be performed at runtime, leading to higher memory and processing power overheads and increased memory fragmentation. Typical examples of heterogeneous data structures are structures and records, they are collections of different fields, each one with a possibly different type. For example, a C structure might include an integer field, a floating-point field, and a character array field, making it an excellent candidate for representing immutable entities in the real world, such as returned records from employee forms or database records. A more direct example of a heterogeneous data structure would be Objects in object-oriented programming (OOP). An object is collection of data members, each with their own data type. This provides more flexibility for modeling during software development. In Object Oriented Programming based systems, for instance, a "Person" object may have a name (string), age (integer) and height (floating-point point number) - heterogeneous data structures are extremely beneficial in representing real time objects. You can also have elements of different types in the same array in dynamically typed languages like JavaScript and Python. So this feature give you flexibility, but at the risk of matching any type and necessitating runtime checks that require type correctness.

This separation of data structures can also be observed in manners regarding primitive and non-primitive data structures. Primitive Data Structures are the basic data types which Take Take direct care of the machine's directions. They are types of data that only hold one value and are often built into computer languages. A whole number is called an integer. decimal number is called a floating-point number. A character is a single letter or symbol. A boolean value is either true or false. Because they are primitive, data structures are simple by definition. They have a set size and a specific way of storing values in

memory that works best at the hardware level. Primitive data structures are simple structures that are described in the same way. Non-primitive data structures, on the other hand, are more complicated and are made with primitive data types. Structures like data frames are made from data and help manage bigger or more complicated information. Groups, linked lists, stacks, queues, trees, and graphs are all types of non-primitive data structures. As we already said, arrays are a type of homogenous data structure that store many items of the same type in memory spots that are close to each other. Linked lists, on the other hand, are made up of nodes. Each node has data inside it and a link to the next node in the list. You can change how much memory is used, but the keys take up extra room. Structures for linear data: Ordered Groups: Queues and Stacks When you set up a linear data structure, the parts are put together in a certain way. Adding or taking away items from a stack starts at the same end. This is called a Last In, First Out (LIFO) movement. Stacks can also be used to keep track of things like computer language function calls. When a function ends, its calls are added to a stack and run backwards. This means that first thing that is added to the line will be the first thing that is served. This works great when they are used in operating systems to plan when to do things like send network packets or print jobs. Data structures like trees and graphs are more complicated than simple data structures. They allow you to arrange data in a tree or a network. We can see a tree here. Each node has a parent and maybe more than one kid node. Aside from that, trees are used to set up file systems, make choices in machine learning, plan syntax trees for programs, and do other things. Relational database stores information in tables, while a graph is made up of nodes (sometimes called vertices) that are linked together by edges. This makes it perfect for showing complicated relationships like those in social networks, road maps, and project management dependencies. However, homogeneous data structures speed up compilation more, but they aren't good for situations where you need more freedom. Different types of data files give you more options, but they may slow down your computer. Whether to use uniform or heterogeneous data structures depends on the needs of application, such as memory constraints, type safety, and how quickly data can be processed. With this information, developers can pick the best data structures for their

apps, making sure that the design of their software is both fast and durable in the long term.

They are made up of basic data types that are spread out and can be taken for granted. They are made up of primitive or non-primitive data structures and allow certain processes to work. There are many patterns we've already talked about, such as Linked Lists, Trees, Graphs, and more. There are times when non-primitive data structures are useful, & they have functions that are good for those times. They make the info and the things that can be done with it public. This group talks about how data is saved in a computer's memory, while the logical view talks about how data is thought of and seen.

Ways to organize real data

Logical data structures show us how we think about data, while physical data structures describe how data is stored in a computer brain. We only teach about how things are put together and how memory is organized. Arrays are basically groups of things that are linked to each other. Memory will be used to store things like the pieces in an array. They have an index that lets you view them at any time, but their size is fixed, which can make it hard to change it. On the other hand, connected lists are made up of memory blocks that are not next to each other but are linked by pointers. While this makes insertion quick and deletion without needing to resize, it leads to worse element access times from needing to traverse the nodes sequentially. Physical structures focus on how efficiently memory is used, and the performance consequences of memory access patterns. The physical data structure you need to use depends on memory, access time, and which kind of operations will be performed most often.

Logical Data& Structures

The logical data structures between the data and the programmer describe how the data is viewed by it and how the programmer can access that data as an abstraction of the physical implementations. They aim at telling how everything is organized and works conceptually, disregarding memory layout. Such as stacks, queues, trees, graphs, etc. Stacks & exhibit Last-In-First-Out (LIFO) behaviour, whether they are implemented by an array or a linked list. Queues provide First-In-First-Out (FIFO) access which is ideal for scheduling and buffering applications. Tree and graph data have a

hierarchical or relational structure and are popular in representing structured and relational data. Logical structures abstract away from the implementation details and think in terms of the conceptual model. There can be multiple physical implementations of the same logical structure, yielding different performance characteristics. A standard adenine queue would end be used with an array or linked list behind the scenes, with ironical trade-offs in cypher efficient and cipher speed.

Arrays

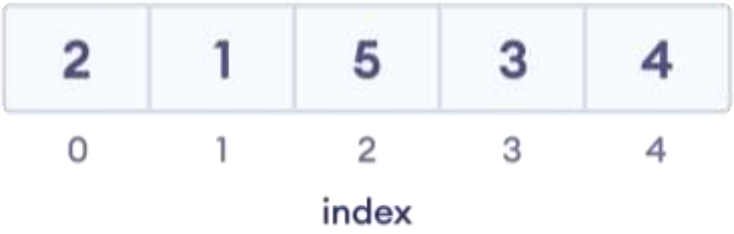

Figure 2: Arrays
[Source: https://www.programiz.com/]

Array is one of the simplest & most commonly used data structure. It is a collection of items which are identified by indices or keys. Arrays have a set size in traditional implementations, which result in static memory allocation. Features contiguously stored elements, index retrieval of information on the other hand, inserting or deleting elements can be expensive as it may require shifting elements to maintain order.

In arrays operations such as reading elements can be done in constant (o(1)) time, searching can be done in linear (o(n)) or logarithmic (o(log n)) time as long as array is sorted, inserting or deleting elements takes o(n) time since the rest of the elements are to be shifted. The size of the input is known and won't change, or you want to get random access often. Arrays are a great way to start building more complicated data structures like matrices and multidimensional data. Dynamic arrays, like the Array List in Java, reallocate themselves automatically when they run out of space. Jagged arrays, where the sizes of the inner arrays vary, and parallel arrays, where the same data is stored in more than one array (one for each type of data), are some other types.

Lists with Links

Figure 3: Linked List
[Source: https://www.programiz.com/]

When you put together a linked list, you get a list of things, which are called "nodes." The list has a link for each thing that goes to the next one. Link lists let you change their size as needed, so you don't have to give them all the memory at once. An easy-to-use type of linked list is the single linked list. Another popular type is the double linked list. A linked list is made up of nodes, and each node only has one link to the next node Each node in a doubly linked list points to both the previous and next node, so you can move through the list in both ways. The last node in a circular linked list points to the first node in the list, making a loop. You can use this loop to do things with the nodes (a list of items) in a certain order. When a lot of items are added or removed, linked lists work well because they can be run in O(1) time if the links are set up properly. To get to the one you want in a linked list, you have to go through each node one by one. This makes the search take longer than when groups are used.

Stacks

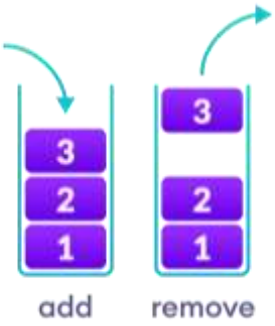

Figure 4: Stack][Source: https://www.programiz.com/]

The name "Stack" comes from the Last-In-First-Out (LIFO) method. You can add an item to a hash set, take away the top item, and then look at the top item. This is a type of data structure in computer science called LIFO. It is a stack. Ranges or linked groups can be used

to make it. A lot of people use them to do things like test expressions, handle function calls in computer languages, undo in text editors, and more. Line up and waiting "First-In, First-Out" (FIFO) is the idea behind a list, which is a simple way to organize data. On one end, things are added, and on the other end, things are taken away. Both enquire (add an item) and desuetude (remove an item) would work this way. Arrays, linked lists, and circular files can all be used to make queues. Lines come in many forms, including priority lines and double ended queues (deques). In priority lines, items are taken out of the queue based on how important they are, not their order. Graphs and trees tree is type of hierarchical logical data structure where each node is linked to one or more other nodes in a parent-child relationship. In binary trees, each node can have no more than two children. They are one of the most popular types of data structures. If you come across a special type, like binary search tree (BST), it will keep things in order. Other types, like an AVL tree or a Red-Black tree, will keep the heights equal. Graphs can be either directed or undirected, and they can be weighted or not weighted. They are used for many things, like routing in networks, social networks, and feedback systems. Depth-first search (DFS) and breadth-first search (BFS) are two popular ways to move through a graph. They are very important to computers because they control how data is saved, accessed, and changed. It's important to know that data structures come in two different types: virtual and real. These differences are very important for figuring out how to make programs work well and improve systems. linked list is a group of nodes that hold changing information & term that connects one list to the next. most important thing about a linked list is probably its dynamic size, as it can grow or shrink while executing and does not require a predefined size. This proves particularly helpful in situations where the data size is variable. Non-contiguous memory allocation is the other important feature. Arrays require memory elements to be stored at contiguous memory locations, while linked lists can have them at random places in memory. This gives the system a certain amount of flexibility, which minimizes memory fragmentation and enables the efficient utilization of available memory. Meanwhile, linked lists have sequential access, which allows traversal from head node to the desired element in a linear fashion. Because this structure does not

support direct indexing, access is slower than with an array, because an element is not specified through an index but must be traversed sequentially.

There are three main types of linked lists: single-linked lists, double-linked lists, and circular linked lists. Each node in a singly linked list has two fields: one that keeps the node's data and one that points to the next node. This building makes it easy to move forward, but not backward. Two-linked lists have data at each node, as well as a pointer to the next node and pointer to the node before it. To move forward or backward, this extra pointer can be used. This makes some jobs go faster. Finally, there is a type of linked list called a circular linked list. Its last node points to its first node, making it a loop that never ends. This trait of going around in a circle is useful in situations where you need to go around in a circle, like in scheduling systems. List with Links: Access, adding, deleting, and searching are some of the things that can be done with a linked list. Accessing an element in an array-based data structure takes O (1) time and doesn't require traversal because we can just use the element's index. However, in a linked list-based data structure, we have to start at the head node and work our way down until we reach the desired position, which takes O(n) time. It takes O (1) time to add and remove things once you are in a place. It takes O(n) time to find an element, though, since each one has to be checked one at a time. Linked lists are helpful when we need to add or remove items often because we can quickly change the links without having to move the items around like we do with arrays. They work well when the amount of data is unknown or changes a lot. This makes them a good choice for allocating memory on the fly. As well as this, this is also true for many generic data types that use linked lists, such as memory allocators, stacks, and queues. Stack is just an abstract data type that works with Last-In-First-Out (LIFO), which means that the last thing you put will be the first thing taken away. This is very helpful in some cases because it keeps things in a certain order. Adding and taking away things from only one end, called the top, is what makes a stack. The latest element is the only one that can be accessed, which means that actions can only be done on that element. You can put together a stack in two ways: with an array or a linked list. Stacks can be used to do four main things: push, pop, peek (or top), and isEmpty. To add an item to the bottom of the

stack, use the push method, which takes O (1) time. The pop action also takes O (1) time to get rid of the topmost element. With that peek action, you can see the most recent addition because it only shows the top element and doesn't get rid of it. The is Empty operation checks to see if there are any things on the stack that can be used and returns a Boolean value in constant time.

Stacks can be used for many things. This is one of the most popular uses LIFO data structures are often used in programming to handle function calls. In this case, function calls are pushed onto a call stack, and the most recently called function runs first. Stacks are utilized for the evaluation of mathematical expressions or the reading of computer language code structures. Applications, like word processors, use stacks to remember past states and undo actions when needed. Then you'll read data until you get to the first Flap object that pops off the stack. You'll then have to go back to the last state that was saved before that flap (which removes data from the stack until a state we need to return to is found). All stacks have three properties: they are reversible, they use last-in-first-out memory, and they are forever. A queue is another type of abstract data. It is made up of elements at the ends of the queue and works on the First-In-First-Out (FIFO) principle, which means that first element added is the first one removed. In situations where things need to be handled in the order they arrive, queues are important. When you add something to one end (the back), you take something away from the other end (the front). This is called a queue. Applications need to use either arrays or linked lists to perform Queue. One type of queue could be easy, while another could be circle, priority, or deques double-ended. The usual FIFO method is used by a simple queue; items are added at the back and taken out at the front. To make a circular line instead, you can use the circular buffer. The end of the queue will wrap around to the top of the queue. This saves room and keeps memory from going to waste. A priority queue is different from other data structures because it links elements' priorities, which means that elements with high priorities are dequeued before elements with low priorities. Lastly, a deque lets you append elements on both sides, so you can manipulate data how you want. Basic operations of queues are enqueue, desuetude, front, & is Empty. In O (1) time, the enqueue move adds a component to the back of the queue. When you use dequeue, an item

is taken out of the front of the queue, which also takes O (1) time. The front action lets us see the front element without taking it away. This is helpful when we want to see what the next element is that needs to be processed. The is Empty action is part of the Write Queue class. The action is Empty checks to see if the queue is empty. It takes no time at all returns a Boolean. In the real world, lines are useful in many ways. There are tools in operating systems that use queues all the time to schedule tasks. Processes are put in a queue by the operating system so that they can run in a fair way. It's interesting that most algorithms that move through the structure of trees (or graphs) are built so that they can handle some of the nested structure of the tree or graph. This means that the whole layer or depth of data is processed at once before moving on to the next layer or depth form, which is easy to do with queues. One important use for queues is as input/output buffers, which store data while it is being read and make the reading go more smoothly. Another place where queues are used a lot is in networking. Servers that handle requests on the web use queues, and networks that send and receive data use them to keep track of packets. The linked list, stack, and queue data structures all give us the best ways to solve different types of situations. Linked lists are great for applications that need to be flexible because they can dynamically allocate memory and add and remove items quickly. In LIFO, the first thing taken away is the last thing added. A stack is used to call functions, evaluate expressions, and undo things. You can do many things with queues, such as plan jobs, store data, do breadth-first searches, and more. The FIFO rule tells them how to work. There are different types of material that can be used to solve different types of problems.

Trees

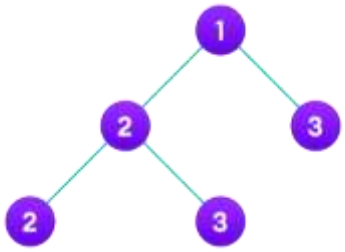

Figure 5: Trees [Source: https://www.programiz.com/]

Tree is a non-linear structured data structure made up of nodes that are linked together by edges. A node can work for one or more child nodes. There is a root node at the top of the hierarchy, & each node can have more than one kid but only one parent (other than the root).

Types:

1. In a binary tree, each node can only have two children.
2. The left child goes to the adult, and the right child goes to the left child.
3. The AVL Tree: BST that balances itself
4. Red-Black Tree: BST that evens itself out and has color traits
5. B-Tree and B+ Tree are balanced trees made for storage systems.
6. Heap: A full binary tree with a heap value (either max heap or min heap).
7. Trie: A tree for saving strings that start with the same letter

How it works:

- Adding: O(log n) for trees that are balanced, O(n) for trees that are not balanced
- Deletion takes O(log n) time for balanced trees and O(n) time for lopsided trees.
- The search time is O(log n) for balanced trees and O(n) for lopsided trees.
- Crossover: level order, in-order, pre-order, and post-order

Use Cases:

- Hierarchical data representation (file systems, organizations)
- Database indexing (B-Trees, B+ Trees)
- Priority queues (heaps)
- Expression parsing and evaluation
- Routing algorithms in networks
- Decision-making algorithms

Graphs

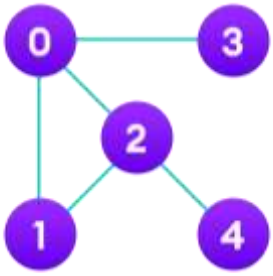

Figure 6: Graph

[Source: https://www.programiz.com/]

graph is a non-linear data structure consisting of vertices (nodes) & edges connecting these vertices. It represents relationships between pairs of objects.

Characteristics:

- Collection of vertices joined by edges
- Edges can point in one direction or not point in any direction (two-way).
- The weights or costs of edges can be changed.

Signs and symbols:

- 2D collection called an adjacency matrix shows the edge between points i and j as cell.
- Adjacency List: collection of linked lists that each contains edges that touch a vertex
- Edge List: This is a list of all the graph's edges.

Types:

1. Directed Graph (Digraph): Edges point in a certain way
2. In an undirected graph, the edges don't go in any particular way.
3. Weighed Graph: Every edge has a cost, also called a weight.
4. A cyclic graph has at least one circle.
5. An acyclic graph doesn't have any loops.
6. This type of graph has a path between every pair of points.
7. A disconnected graph has parts, or edges, that are not linked to each other.

How it works:

- Depending on the format, use O(1) to O(V+E) to add or remove a vertex.
- Change the model by adding or taking away an edge from O(1) to O(V).
- If you have an adjacency matrix, use O(1) to see if two points are next to each other. If you have an adjacency list, use O(8).
- Traversal: Breadth-First Search (BFS) & Depth-First Search (DFS)

Use Cases:

- Social networks (connections between people)
- Transportation networks (roads, flights)

- Web page linking structures
- Network routing algorithms
- Dependency resolution
- Recommendation systems
- Path finding algorithms (GPS navigation)

Hash Tables

Hash table (hash map) is a data structure that& implements an associative array abstract data type, mapping keys to values using a hash function.

Characteristics:

- Uses a hash function to compute an index into an array of buckets
- Ideally provides O(1) average time complexity for lookups
- Handles cell

1.3 C++ Memory Map, Memory Allocation Operators (new, delete)

Memory management is a fundamental aspect of programming, particularly in languages like C++, where developers have explicit control over memory allocation and deallocation. In the context of data structures, understanding how memory is allocated, utilized, and freed is crucial for optimizing performance and avoiding memory-related issues such as leaks and fragmentation. The C++ memory map provides insight into how memory is structured, and memory allocation operators like new and delete allow dynamic memory management, making it easier to work with complex data structures such as linked lists, trees, and graphs. A typical C++ memory map consists of several regions: the code segment, the stack, the heap, and the data segment. The code segment contains the executable instructions of the program, while the data segment holds global and static variables. The stack is used for managing function calls and local variables, following the Last In, First Out (LIFO) principle. Each time a function is called, a new stack frame is created, which contains function parameters, return addresses, and local variables. When the function completes execution, its stack frame is removed. The heap, on the other hand, is used for dynamic memory allocation, where memory is allocated and deallocated manually by the programmer. Unlike the stack, the heap does not follow a strict order for allocation and deallocation, making it more flexible but also more prone to issues such as fragmentation and memory leaks. In the context of data structures, dynamic memory allocation is particularly useful. When working with arrays, for instance, static arrays have a fixed size determined at compile time, limiting their flexibility. Dynamic arrays, however, can be allocated on the heap, allowing for more efficient memory usage and resizing during runtime. Linked lists, trees, and other dynamic data structures rely heavily on heap memory because their sizes are not known in advance, and they require the ability to grow and shrink dynamically.

C++ provides memory allocation operators to manage heap memory efficiently. The new operator is used to allocate memory dynamically, returning a pointer to the allocated space. It can be used to allocate memory for single variables as well as arrays. For example, int* ptr =

new int; allocates memory for a single integer on the heap, while int* arr = new int[10]; allocates an array of ten integers. When using new, it is essential to release the allocated memory once it is no longer needed to prevent memory leaks. This is done using the delete operator, which deallocates the memory and frees up the space for reuse. For a single variable, delete ptr; is used, and for an array, delete[] arr; ensures that all elements are properly deallocated. Memory management becomes more complex when working with objects. When an object is allocated dynamically using new, its constructor is called automatically, ensuring proper initialization. Similarly, when delete is used, the object's destructor is invoked, allowing for any necessary cleanup. This is particularly important when dealing with classes that manage dynamic resources, such as file handles or dynamically allocated arrays within objects. To handle such cases, C++ also provides smart pointers, such as std::unique_ptr and std::shared_ptr, which automate memory management and help prevent memory leaks by ensuring that allocated memory is released when no longer needed. Despite the advantages of dynamic memory allocation, improper usage can lead to several issues. One common problem is memory leaks, which occur when allocated memory is not properly deallocated. Over time, this can lead to increased memory consumption, slowing down the program and eventually causing it to crash. Dangling pointers are another issue, occurring when a pointer is used after the memory it points to has been deallocated. This can result in undefined behavior, leading to segmentation faults and program instability. Double deletion, where the delete operator is called on the same memory address more than once, can also cause unpredictable behavior and program crashes.

Efficient memory management is particularly important in data structures that involve frequent memory allocation and deallocation, such as linked lists and trees. In a linked list, each node is dynamically allocated, and failure to properly deallocate nodes when deleting elements can result in memory leaks. Similarly, in trees and graphs, managing memory efficiently is crucial to ensure optimal performance and avoid excessive memory consumption. Many advanced data structures use memory pooling techniques or custom memory allocators to optimize performance and minimize the overhead of frequent allocation and deallocation operations. Another consideration

in memory management is fragmentation, which occurs when free memory is broken into small, non-contiguous blocks, making it difficult to allocate large contiguous memory chunks. This is particularly problematic in long-running programs that frequently allocate and deallocate memory. To mitigate fragmentation, programmers can use strategies such as memory compaction or custom memory allocators that group similar allocations together. Understanding memory allocation and deallocation is also essential when working with multi-threaded applications. When multiple threads allocate and deallocate memory simultaneously, race conditions and synchronization issues can arise. Properly managing memory in multi-threaded programs requires techniques such as thread-local storage, mutexes, or lock-free data structures to ensure safe and efficient memory access.

In modern C++ programming, best practices encourage the use of smart pointers and the RAII (Resource Acquisition Is Initialization) principle to automate memory management and reduce the risks associated with manual allocation and deallocation. Smart pointers like std::unique_ptr automatically delete the allocated memory when they go out of scope, ensuring that memory leaks are prevented. std::shared_ptr allows multiple pointers to share ownership of a resource, automatically freeing the memory when the last reference is destroyed. Understanding the C++ memory map and memory allocation operators is essential for efficient programming, especially in the context of data structures. Proper memory management ensures that programs run efficiently, avoid crashes, and make optimal use of available system resources. While dynamic memory allocation provides flexibility, it also introduces complexities that require careful handling. By following best practices, such as using smart pointers and minimizing memory fragmentation, programmers can create robust and efficient applications that make the best use of available memory.

1.4 Performance Analysis & Management - Space Complexity, Time Complexity

In the study of data structures, performance analysis and management play a crucial role in determining the efficiency of algorithms and their practical applicability. Two fundamental aspects of performance analysis are space complexity and time complexity, both of which directly influence how well an algorithm performs under various conditions. Understanding these complexities is essential for optimizing computational resources, ensuring that operations are executed efficiently, and enabling effective data processing. As data structures form the foundation of computer science, the ability to analyze their performance helps in selecting the most suitable data structures for specific tasks and environments. Space complexity refers to the amount of memory an algorithm requires to execute, including both the input storage and auxiliary storage used during computation. This metric is essential because memory is a finite resource, and inefficient use can lead to system slowdowns or failures, especially in large-scale applications. Space complexity is typically expressed in terms of Big-O notation, which describes the upper bound of an algorithm's memory usage relative to input size. For example, an algorithm with O(n) space complexity grows linearly with input size, while one with O(1) remains constant regardless of input. Some data structures, such as arrays, require contiguous memory allocation and can be memory-intensive if large elements are stored. In contrast, linked lists utilize memory dynamically, allocating space as needed, which can be more efficient in certain scenarios. However, the trade-off lies in the additional storage required for pointers in linked lists, which increases overall memory usage. Trees, graphs, and hash tables also come with their own space requirements, with trees typically needing space proportional to the number of nodes and graphs depending on their representation, whether adjacency lists or adjacency matrices. Time complexity, on the other hand, refers to the amount of time an algorithm takes to complete as a function of input size. It is also expressed using Big-O notation to classify algorithms based on their growth rates. The time complexity of an operation varies depending on the data structure used. For instance,

searching in an unsorted array takes O(n) time in the worst case, whereas a well-balanced binary search tree achieves O(log n) search time. Sorting algorithms also have varying time complexities, with bubble sort operating at $O(n^2)$ in the worst case, while quicksort and mergesort can achieve O(n log n) performance under optimal conditions. Understanding time complexity helps in making informed choices about which data structure or algorithm to employ for a particular problem. For example, hash tables provide average-case O(1) time complexity for search, insertion, and deletion operations, making them ideal for applications requiring fast lookups, such as database indexing. However, in cases of hash collisions, the worst-case time complexity can degrade to O(n), making collision resolution techniques a critical aspect of hash table implementation.

The balance between time and space complexity is a crucial factor in data structure design. Often, reducing time complexity may come at the cost of increased space usage and vice versa. A classic example is the trade-off between recursion and iteration. Recursive algorithms, such as those used in tree traversals, may have a simple and intuitive implementation but consume additional stack space, leading to O(n) auxiliary space complexity in the case of deep recursive calls. Similarly, caching techniques like dynamic programming improve time complexity by storing intermediate results, but they require additional memory storage, leading to an increase in space complexity. Efficient performance management also involves optimizing data structures based on the problem requirements and constraints. Consider the case of priority queues, which can be implemented using different data structures such as arrays, linked lists, or heaps. A simple array implementation may provide O(1) insertion time but require O(n) time for extracting the highest-priority element. On the other hand, a binary heap enables both insertion and extraction in O(log n) time, making it a more balanced choice for handling priority-based tasks efficiently. Similarly, graph algorithms require careful consideration of space and time complexity. For instance, Dijkstra's algorithm for shortest path computation performs better with an adjacency list representation in sparse graphs but benefits from an adjacency matrix in dense graphs where quick lookups are necessary. In real-world applications, the principles of performance analysis guide the design of efficient software and

systems. Search engines, for example, rely on indexing techniques that balance time and space trade-offs to deliver fast query results. Social media platforms use graph data structures to manage user connections and efficiently recommend new friends or content. Database management systems optimize query performance by employing indexing, caching, and balanced tree structures like B-Trees. Furthermore, operating systems employ scheduling algorithms and memory management strategies that are deeply rooted in performance optimization principles.

The impact of poor performance analysis can lead to inefficiencies, increased operational costs, and scalability issues. An algorithm that performs well on small datasets may become impractical when scaled to millions of records. Thus, continuous assessment and refinement of data structures and algorithms are necessary to keep systems responsive and efficient. Developers often use profiling tools to analyze performance bottlenecks, and techniques like amortized analysis help in understanding long-term efficiency trends of data structures. Performance analysis and management in data structures are critical aspects of computer science that determine how effectively algorithms handle computational tasks. Space complexity ensures that memory usage is optimized, preventing resource wastage and system slowdowns. Time complexity, on the other hand, dictates how quickly an algorithm can process inputs and deliver results. By understanding these complexities, developers can make informed decisions about selecting the right data structures and algorithms for specific applications. The trade-offs between space and time complexity must be carefully managed to achieve optimal performance, ensuring that software solutions remain efficient, scalable, and capable of handling growing data demands. Whether designing search engines, databases, or network systems, the principles of performance analysis remain integral to developing high-performance computing solutions.

MCQs:

1. Which of the following is NOT a linear data& structure?
    a) Stack
    b) Line up
    c) Graph
    d) List

2. Which of the following data& types & allows LIFO (Last In, First Out) access?
    a) Queue
    b) Stack
    c) List with Links
    d) Graph
3. Which C++ operator is used for dynamic memory allocation?
    a) malloc
    b) delete
    c) new
    d) alloc
4. What does the complexity of time tell you?
    a) The amount of memory used by an algorithm
    b) How long a program takes to run based on the size of the input
    c) How fast the CPU is
    d) The amount of power an app uses
5. Which data structure follows FIFO (First In, First Out)?
    a) Stack
    b) Line up
    c) Graph
    d) Tree
6. Which of the following operations is NOT typically performed on an array?
    a) Insertion
    b) Deletion
    c) Traversal
    d) Random access
7. Which sorting algorithm has the best average-case time complexity?
    a) Bubble Sort
    b) Selection Sort
    c) Quick Sort
    d) Sort by Insertion
8. What is space complexity in data structures?
    a) The number of processors required
    b) The amount of memory required for an algorithm
    c) The execution time of an algorithm

d) The storage capacity of a hard disk

9. Which data structure is best suited for implementing recursion?
    a) Stack
    b) Queue
    c) Graph
    d) Linked List
10. Which of the following data structures can be used to implement a queue?
    a) Stack
    b) Linked List
    c) Array
    d) Both b and c

Short Questions:

1. Define data structure and explain its classification.
2. What are the advantages and disadvantages of an array?
3. Explain dynamic memory allocation using new and delete in C++.
4. What is time complexity, and why is it important?
5. Compare linear and non-linear data structures with examples.
6. Explain the concept of space complexity in data structure performance.
7. What is the difference between a stack and a queue?
8. List different types of data structures, and briefly describe their usage.
9. What is the difference between static and dynamic memory allocation?
10. Explain the importance of performance analysis in data structures.

Long Questions:

1. Explain the classification of data structures with examples.

2. Explain how the different types of data structures (list, stack, queue, tree, and graph) work.

3. Discuss C++ memory management and explain new and delete operators with examples.

4. What are time and space complexity, and how do they impact algorithm performance?

5. Compare array and linked list in terms of memory usage and operations.

6. Explain different types of data structures and their applications in real life.

7. How does performance analysis help in selecting a suitable data structure?

8. Write a program to demonstrate memory allocation and deallocation in C++.

9. Explain the difference between basic and non-primitive data structures.

10. Talk about how important it is for software developers to use fast data structures.

MODULE 2
ARRAY

LEARNING OUTCOMES

- Understand the concept of arrays and their usage in programming.
- Learn about one-dimensional and multi-dimensional arrays.
- Perform basic array operations such as insertion, deletion, traversal, and merging.
- Learn how to pass arrays to functions in C++.
- Understand the implementation of multi-dimensional arrays.

Unit 4: Introduction of Array

2.1 Introduction to Arrays: One-Dimensional Arrays, Initialization, Accessing, Implementation, Passing Arrays to Functions

Step by step. Data arranged in this way can be accessed and computer programming that allows you to store several values under single variable name. They create a heap in memory as a group of data points, generally with same data type, ordered array a core data structure in collections. They are used in structures and algorithms, so every coder needs to know about them. Files and data will be very much hard if the data is not properly structured and arranged into Arrays are the building blocks of many more sophisticated data

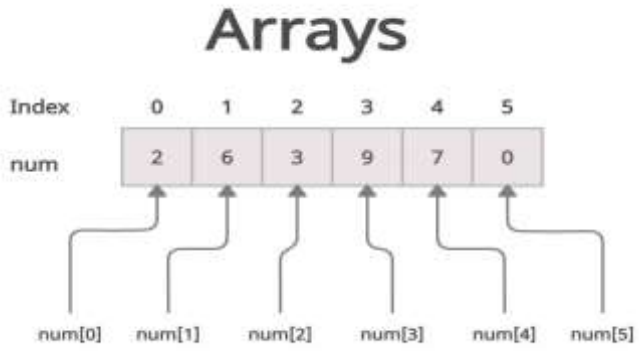

Figure 1.2: Arrays
[Source: https://usemynotes.com/]

One-Dimensional Arrays

They are stored next to each other in computer memory. As a linear structure, one-dimensional arrays are conceptually simple and practically useful for many programming are stored in consecutive memory locations. Imagine it as a series of boxes; each box holding a value, one dimensional array is linear data structure, where the elements can be accessed in constant time with just one index, which makes them ideal for many common tasks. Items put them in an array with only one dimension. With their simple structure, they can if you want to store sequences, such as a list of temperatures or student grades or inventory enough practice, it's like second nature! In most computer languages, first element is named 0 and the second element is named 1. For beginners, this zero-based indexing technique can be

hard to understand, but it's necessary to learn how arrays work, which is what indexing is. There is a basic idea behind the first element in an array**.** The time of announcement and can't be changed after the fact. In dynamic languages like Python, JavaScript, or Java (when using Array List) arrays can grow and shrink when you need, we refer to the size or length of the array, we typically mean how many elements it could contain. In languages such as C & C++, size is fixed When as: An array of five integers, for instance, could be depicted [23, 45, 12, 8, 95] 23, index 1 has 45 and index 4 has 95. Notice in here that at index 0 you have access to any of its elements in time complexity of O (1) if we know the index of our element, as the memory address can be calculated with the formula: occupied in the nearby memory locations). Because they are in a contiguous storage, we can One-dimensional arrays are allocated contiguously the elements are location of element at position i = base address + lot of performance when random access is needed.

Array Initialization

Languages have specific means of initializing arrays; the general ideas are the same. to its elements. While some programming Initialization is the creation of an array, where we may optionally assign the values

Static Initialization

Values are known in advance. For all array elements during declaration. This works fine if the Static initialization–it means providing values Initialization might look like: In C/C++, a static

50}; int numbers [5] = {10, 20, 30, 40,

In Java:

50}; int nums = {10, 20, 30, 40,

Are also referred to as arrays): Python uses lists instead of arrays (however, they

df (Data Numbers)` # < — 10 < 20 < 30< 40 < 50

Dynamic Initialization

then filling its elements, individualy, often inside loops or using user input. Dynamic initialization includes: creating the array first and

In C/C++:

```
int numbers[5];
for(int i = 0; i< 5; i++) {
numbers[i] = i * 10;
```

}

In Java:

```
int[] numbers = new int[5];
for(int i = 0; i< 5; i++) {
numbers[i] = i * 10;
}
```

In Python:

```
a list of 5 zeros  numbers = [0] * 5 # Creates
for i in range(5):
numbers[i] = i * 10
```

Default Initialization

Tend to give elements default values: If arrays are created without explicitly specifying values, programming languages

- In C/C++, local arrays have undefined values unless explicitly initialized, while global arrays are initialized to zero.
- In Java, array members are set to their default values by default (0 for numbers, false for booleans, and null for object references).
- In Python, you have to directly set up list items to be used. Initializing an array and figuring out its size
- Some languages allow the compiler or interpreter to decide the array size based on number of initializers:
- If you use C or C++, you can write: int numbers[] = {10, 20, 30, 40, 50}; int size = 5; If you use Java, you can set the size to 5 by calling int[] numbers = {10, 20, 30, 40, 50}; Partial Initialization
- You can only set up some parts of an array in some languages: In C/C++, the code looks like this: int numbers[5] = {10, 20}; // The other parts (numbers[2, 3], and 4]) are set to 0.
- Knowing about these initialization techniques can help you pick the best one for your programming job, taking into account things like how easy it is to read, maintain, and run. Getting to Array Elements
- Indexing is the main way to get to elements in an array. We use an integer value to tell the program where to find the element we want to change or recover. Access to Basic Elements

- Accessing elements in an array can be done in a few different ways depending on the computer language, but in most cases, you need to use array name followed by the index in square brackets:

When you use C/C++, Java, or JavaScript:

array_name[index]

In Python:

array_name[index]

For example, to access the third element (which is at index 2 due to zero-based indexing) of an array named "scores":

scores[2]

Modifying Array Elements

Arrays allow in-place modification of their elements, which is one of their key advantages:

scores[2] = 95; // Sets the value at index 2 to 95

Out-of-Bounds Access

When accessing array elements, it's crucial to ensure that the index is within the valid range. Accessing an array with an invalid index can lead to different behaviors depending on the language:

- In C/C++, accessing an out-of-bounds index is undefined behavior, which may cause program crashes, data corruption, or seemingly normal operation with unexpected results.
- In Java and Python, an exception is thrown (Array Index Out Of BoundsException in Java, Index Error in Python).
- Some languages like JavaScript will return undefined for out-of-bounds access.

Iterating Through Arrays

common operation is to process all elements of an array, typically done with loops:

Using a for loop (C-style):

```
for(int i = 0; i<array_length; i++) {
 // Process array[i]
}
```

In languages that support it, enhanced for loops (for-each loops) provide a cleaner syntax:

In Java:

```
for(int value : array) {
 // Process value
}
```

In Python:

for value in array:

Process value

Access at Random

The thing that makes arrays unique is that they can do constant-time random access. This means that it takes the same amount of time to get to any element by its number, no matter how big the array is or where the element is in it. Because arrays can allocate memory in chunks and calculate direct addresses, this feature is present.

Checking the Limits

To avoid runtime mistakes, it's good programming practice to use boundary checks when accessing array elements:

if (index >= 0 && index <array_length) { // It's safe to access array[index] else { // Take care of the error }

A lot of current programming languages and frameworks use automatic boundary checking to make code more reliable and stop security holes.

Several-step Access

To get to nested items in arrays of objects or arrays of arrays, you have to do more than one indexing operation:

students[2].grades[3] // Getting to the third student's fourth grade

Mastering array access is an important part of programming because it's the base for many algorithms and data manipulations.

Details on how to implement an array

Arrays are implemented in a way that depends on how the hardware and computer languages work with memory management and allocation. Knowing these details helps coders choose when and how to use arrays in the best way.

Layout of Memory

Arrays are typically implemented as contiguous & blocks of memory. Each element occupies a fixed amount of memory based on its data type. For example, in many systems:

- An integer array with 5 elements may occupy 20 bytes (assuming 4 bytes per integer)
- A character array with 10 elements may occupy 10 bytes (assuming 1 byte per character)

This contiguous layout enables the efficient calculation of element addresses and contributes to the fast access times arrays offer.

Figuring Out Memory Address

Formula to find the memory address of any element is: address of element at index i = base address + (i × size of element) This calculation is what makes it possible for any entry in an array to be accessed in O(1) time. Details about the Array Header There are many ways that arrays can store more than just the elements:

• How long or how big the array is
• Capacity (for groups that change)
• Type of data (in some languages)

This metadata helps control memory and check for bounds. Linear Arrays vs. Fluid Arrays

the size of a static array is set at build time & can't be changed. Traditional arrays in C and C++ are two examples: int numbers[100]; // Size is set at 100 items

Dynamic groups can change size while they're being used. Some examples are std::vector in C++, Array List in Java, and list in Python.

Array in Java Script

1. Giving out a new, bigger piece of memory when it's needed is a common way to use dynamic arrays.
2. Copying parts of the old block to the new one
3. Getting rid of the old memory block
4. Making changes to the base address pointer

This process is generally hidden from the programmer, but it slows things down, especially when the size of the object is changed a lot. Array Bounds Checking: Different languages have different ways of checking array bounds:

C and C++ don't usually do automatic bounds checking because they focus on speed. Java, Python, and many other modern languages do automatic bounds checking to stop buffer overflow vulnerabilities. Bounds checking slow things down a little, but it makes things much safer and more reliable. Memory Management: Arrays use a variety of memory management methods, such as:

- Arrays that are put on the stack, like local arrays in C/C++, are immediately freed up when they are no longer needed.
- Heap-allocated arrays, like those made with "new" in C++ or Java, need to be explicitly freed in languages that don't have trash collection.

- Languages that use garbage collection, like Java, Python, and JavaScript, instantly free up array memory when it's not being used.

Performance of Cache

Most of the time, arrays work well with current CPU cache systems because Predictable access patterns: Elements that are close to each other are likely to be accessed together and put into cache at the same time. Sequential navigation makes prefetching work well. This trait that is good for caches helps many array-based algorithms work faster.

Putting it into practice in different languages Arrays are used in different computer languages in different ways:

- C/C++: Basic arrays are simple memory blocks with minimal overhead, while STL containers like vector add functionality at the cost of slight overhead.
- Java: Arrays are objects with built-in length field and bounds checking.
- Python: Lists are dynamic arrays with extensive functionality, implemented as arrays of pointers to Python objects.
- JavaScript: Arrays are specialized objects where indices are converted to string keys.

Performance Characteristics

The implementation details lead to specific performance characteristics:

- Access: O(1) - Constant time for both read and write operations
- Search (unsorted array): O(n) - Linear time as each element must be checked
- Insert/Delete at the end (dynamic array): Amortized O(1) - Occasional resizing makes this amortized constant time
- Insert/Delete at arbitrary position: O(n) - Elements after the insertion/deletion point must be shifted

Understanding these implementation details helps programmers predict performance implications and choose appropriate data structures for their specific needs.

Passing Arrays to Functions

Passing arrays to functions is a common operation in programming, but the behavior varies significantly across languages due to

differences in how arrays are implemented and how function parameters work.

Pass by Reference vs. Pass by Value

When passing arrays to functions, most languages effectively pass them by reference, meaning function works with the original array rather than a copy. This behavior occurs because:

- In C/C++, an array name decays to pointer to its first element when passed to a function
- In Java, arrays are objects and objects are always passed by reference
- In Python, lists (Python's dynamic arrays) are passed by reference

This means that changes made to array inside function will affect original array outside the function.

Syntax for Array Parameters

The syntax for defining functions that accept arrays varies by language:

In C/C++:

```
void processArray(int arr[], int size); // or
void processArray(int* arr, int size);
```

In Java:

```
void processArray(int[] arr);
```

In Python:

```
def process_array(arr):
```

Note that in C and C++, you typically need to pass the array size as separate parameter since this information is not available from the array pointer itself.

Example: Modifying Arrays in Functions

This example demonstrates how modifications to an array inside a function affect the original array:

In C:

```
void doubleElements(int arr[], int size) {
for(int i = 0; i< size; i++) {
arr[i] *= 2;
 }
}
int main() {
 int numbers[5] = {1, 2, 3, 4, 5};
```

```
doubleElements(numbers, 5);
 // numbers is now {2, 4, 6, 8, 10}
 return 0;
}
```

In Java:

```
void doubleElements(int[] arr) {
for(int i = 0; i<arr.length; i++) {
arr[i] *= 2;
 }
}
// Usage
int[] numbers = {1, 2, 3, 4, 5};
doubleElements(numbers);
// numbers is now {2, 4, 6, 8, 10}
```

Passing Multidimensional Arrays

Passing multidimensional arrays requires special syntax in some languages:

In C/C++, the dimensions except the first must be specified:

```
void process2DArray(int arr[][10], int rows); // For a 2D array with 10 columns
```

In Java, multidimensional arrays are arrays of arrays, so only the first level needs to be specified:

```
void process2DArray(int[][] arr);
```

In Python, multidimensional arrays (nested lists) are passed like any other object:

```
def process_2d_array(arr):
```

Returning Arrays from Functions

Returning arrays from functions also varies by language:

In C, you typically return a pointer to the array, often using dynamically allocated memory:

```
int* createArray(int size) {
 int* arr = (int*)malloc(size * sizeof(int));
 return arr;
}
```

In Java, you can return the array reference directly:

```
int[] createArray(int size) {
int[] arr = new int[size];
 return arr;
```

```
}
```

In Python, you can return lists directly:

```
def create_array(size):
arr = [0] * size
 return arr
```

Using Array Parameters Safely

When working with array parameters, especially in languages without automatic bounds checking, it's crucial to implement safety measures:

1. Always pass and check the array size
2. Implement boundary checks before accessing elements
3. Document clearly how the function modifies the array
4. Consider using const/readonly qualifiers for arrays that shouldn't be modified

Performance Considerations

When passing large arrays, consider these performance aspects:

- In C/C++, passing arrays by pointer avoids copying, which is efficient
- In Java and Python, only the reference is copied, not the entire array
- For very large arrays, consider passing a reference/pointer to a subsection if only a portion needs processing

Arrays vs. Other Collection Types

When designing functions, consider whether arrays are the most appropriate parameter type:

- Arrays provide fast random access but limited functionality
- Some languages offer more feature-rich collection types like vectors, lists, or ArrayLists
- Collections with iterator support might provide more flexible function interfaces

Understanding these considerations helps in designing functions that are both safe and efficient when working with arrays.

Operations on Arrays &Common Algorithms

In these operations is crucial for good programming and algorithms. Being proficient Arrays have different operations and are building blocks of many.

Basic Operations

Insertion

an element at a given position in an array, this means you will have to shift all elements from that position: When you want to insert position, value, size): insert(arr,

if position size:

Invalid position return error //

elements to provide space Shift

position--: for i=size-1 to

arr[i+1] = arr[i]

// Insert the new element

arr[position] = value

return size + 1 // New size

O(n) time. In the worst case, this operation takes

Deletion

if one element is deleted, all subsequent ones must move: Likewise, a position of an arraysize of arrayfunctiondelete(arr, position, size): // deletes

if position

2.2 Operations on One-Dimensional Arrays: Insertion, Deletion, Traversal, and Merging Elements

Arrays are extremely popular and useful across many areas of programming for general use, as they are simple, and an efficient way to store data, and are basic data structures used when building other data structures found in computer programming, allowing multiple items of the same type to be grouped in a clean manner. Since these elements are consecutive in the memory location and it is easy to access here are one of the very common data the fixed size of many programming languages, and possibly expensive operations to add or remove elements in the data structure at places other than the tail. However it has few limitations such as O(1) time access operation. This trait of array is so useful for the scenarios which needs Arrays, are strong because they give indexing, a favourable in-line access to all their elements for a index, so they Arrays, but 1D arrays are a primer on what arrays do and how they work. Only one index exists for these arrays, which makes them intuitive and simple to use. For more complex and multi-dimensional data, there is the multidimensional row or column of elements. What is one-dimensional array and in what agents in this context, we the most basic type of arrays, One-dimensional arrays can be understood as just. and performance-related behaviour. All these operations are covered with their details of implementations, applications, traversal and deletion, merging etc. These operations are essential for all programming, especially in array manipulation, so good understanding of these are also vital for successful array manipulation. InsertionONE-Dimensional Insert Operations on 1D Array and performance parts challenges for each insertion scenario. This covers inserting at the head, middle or tail of an array. Complexity due to adding new element to array structure There are several methods to insert an element at a particular position in the array. Array Insertion at the end of insertion is easy: store the position right after the last element (typically by maintaining a length or size variable) and insert there. Array, this would be adding it to the first empty space after the last filled index. A simple and efficient operation that does that is adding an element to the array if you have

capacity to add. (Constant time insertion into an array: O(1)) arrays and they grow automatically, but that could mean allocating a new, larger array and copying elements over, and that will happen under the covers. A dynamic end needs a check whether there is enough space to add a new element in an array (Example: Python lists, Java Array List). But it should raise an error inserting in case it has already reached its capacity or the array should be resized, in case of fixed capacity array, inserting at application of amortized analysis so that this cost is charged over a sequence of operations, resulting in O(1) amortized time complexity finally for insertions to end. This will have O(1) space complexities unless we need to resize, in which case, it has O(n) complexities due to needing to copy the existing elements to the new memory location. Time complexity of insertion at last & of an array is O(1), thus if there is little space remaining we can efficiently make an insertion at end of array in most modern implementations. This storage is Hence, Beginning of Array Insert at must be contiguous that enables valid indexing of the element. This change the array storage However, it is more complicated to add an element to the beginning of an array than to the end, as it requires moving all existing elements.

This is ensuring that we are shifting all elements previous the last element till the first. At last, at the Time of starting the insertion: All the insertion steps: Existing elements needs to be shifted toward right from linked lists rather than arrays. arrays. In some use cases where frequent insertions to the front are required, maybe all or part even of the elements may be better candidates. So, it is slower to insert in starting than to insert at end, especially for large Insertion at first has O(n) time complexity, which is the take to make. Move in an Array Insert at Index means that every element from that position to the last on e should be shifted one position forward, creating Space for the new element to be inserted into the compartment. Inserting at an arbitrary position in the array combines both end for insertion and beginning insertion strategies. Inserting Element at particular PositionAs the name suggests, Inserting at any a new element is then placed at the array with the newly sized length, creating the new array. Next, it uses shifted right all the elements from the end to First of all, it checks if the position where they must be inserted (between the 0 and the input list, 1. The insertion may also be needed at

different levels that can be inserted at time. You're doing random insertions then in general that's (n/2) time which is still Big-O(n). On the contrary, insertion at specific positions can be quite expensive for huge arrays, considering the case for multiple insertions, however, if we are inserting at the front. Assuming you know about the insertion point.

Insertion in Sorted Arrays

If we are dealing with sorted arrays, we must fit the new element in the right place--which adds complexity to arrangements. This is about getting the right But such that: The specific steps in the case of sorted insertion typically appears Finding the correct position for the new element using binary search (for faster position determination) or linear searchShifting all elements greater than the new element one position to the rightInserting the new element at the identified positionSorted insertion ensures that the array remains ordered after each insertion, which can be beneficial for algorithms requiring sorted data. However, the operation still requires shifting elements, resulting in an O(n) time complexity in the worst case. This complexity comes from the potential need to shift all elements if the new element belongs at the beginning of the array.

Batch Insertion Strategies

For scenarios requiring multiple insertions, batch insertion strategies can be more efficient than performing individual insertions sequentially. These strategies minimize the number of shift operations by calculating the final positions of existing elements after all insertions are considered. One approach to batch insertion is to maintain a buffer zone in the array, allowing for multiple insertions without requiring immediate shifts. Another approach is to collect all insertion requests and process them together, optimizing the shifting operations. More sophisticated implementations might use techniques like sparse arrays or dynamic allocation to accommodate batch insertions efficiently. Batch insertion strategies can significantly improve performance for applications requiring frequent insertions. time complexity can be reduced from O(m*n) for m individual insertions to approximately O(n+m) for batch processing, where n is the number of existing elements and m is number of elements to insert.

Deletion Operations in One-Dimensional Arrays

Deletion, the process of removing elements from an array, is another fundamental operation in array manipulation. Like insertion, deletion strategies vary depending on the position of the element to be removed. Understanding these strategies is essential for maintaining array integrity and optimizing performance during element removal.

Deletion from the End of an Array

Removing an element from the end of an array is the simplest deletion operation. This operation involves reducing the array's logical size by one, effectively making the last element inaccessible through normal array operations. In many implementations, the element itself might remain in memory until overwritten, but it is no longer considered part of array from a logical perspective. When you delete an element from an array with specific size tracking, the size counter usually needs to be lowered. In programming languages that use dynamic arrays, this could also cause memory optimization processes to happen if the array gets significantly empty. In some implementations, the position of the removed element might be nullified to make trash collection easier or stop memory leaks. End delete takes O(1) time, which means it works very quickly no matter how big the array is. This operation that happens all the time is one reason why methods like stack implementation using arrays work so well; they mostly involve adding and removing items at the end.

Deletion from the Beginning of an Array

When you remove an element from beginning of an array, all other elements have to be moved to the left by one place to fill the empty space. This change is needed to keep the storage of elements next to each other and the ordering correct, so that array elements can still be accessed where they should be. Parts of the process are:

1. Getting rid of the part at index 0
2. Moving all parts that come after this one spot to the left
3. Taking away from the array's reasonable size

Like beginning insertion, beginning deletion takes O(n) time because all the remaining parts might need to be moved. Because of this linear complexity, beginning deletion is pretty pricey for big groups, especially if it is done a lot. Getting rid of something from a certain spot in an array Deleting from a certain point in an array is like deleting from the beginning or the end. Once the element at the given place is taken out, all elements that come after it must be moved one

position to the left to keep the array going. In most cases, the steps are as follows:

1. Make sure that the delete point is inside the array's boundaries.
2. Take out the part from the location given.
3. Move everything one space to the left after the deletion spot

Decrement the array's logical size

The time complexity for deletion from a specific position depends on the position itself. In worst case, when deleting from the beginning, complexity is O(n). On average, assuming random deletion positions, the complexity remains O(n/2), which simplifies to O(n) in big-O notation.

Deletion by Value Rather Than Position

In many practical scenarios, elements need to be deleted based on their value rather than position. This operation requires first searching for the element in the array and then performing the deletion once the element is found. If multiple instances of the value exist, additional logic is needed to determine whether to delete one or all instances.

The process typically involves:

It takes O (n) time to look for the item in the array, and it takes O(log n) time for sorted arrays to use binary search.

1. If found, deleting the item at the found place
2. Continuing the search to find and delete more cases if you want to

The search part takes the most time, so deletion by value takes O(n) time for unsorted arrays or O(log n + k) time for sorted arrays, where k is the number of places that need to be moved after deletion. During deletion, we make some elements look like they are removed, which gets rid of them without actually getting rid of them. This is done with flags or special values. There are two types of deletion: logical and physical. Logical deletion gets rid of things without deleting them. It takes more complicated traversal logic to skip over "deleted" elements. This is especially helpful when the space being removed requires a shift that takes a long time (O(1) time complexity). But it costs more in terms of space use and could when you use logical deletion, you can skip over empty space or long deletion periods, which speeds up the process. However, when you shift elements, the process takes O(n) time in worst case. Use: app can be used anywhere it wants to For example, physical deletion protects the integrity of an

array and saves space, but it comes with the cost of Batch Deletion Strategies, which are meant to cut down on shift operations by handling multiple deletions in one operation. If you do a lot of deletes, batch delete strategies can make your system run much faster than single deletes. They're If you delete m items, you can speed up apps that to O(n) from O(m*n). a first pass, and then in a second pass, clean up all the scoped parts at a high level. This will cut down on the number of shifts. One common method is to list the elements that need to be deleted so that deletion-heavy tasks run much faster without affecting the integrity of the array. When you do more than one delete action at the same time, the time complexity of each operation will still be taken into account. To reduce the time complexity, you can do a batch deletion operation. This could lead This means that even with improvements

Traversal Operations in One-Dimensional Arrays: Traversal operations help us work with arrays correctly. like looking, changing, collecting, and so on. Since we know different, we look at each item in a collection. A lot of other things depend on it, like Crossing: Most of the time, Sequential Traversal Cache is used for this. It is at the heart of many array functions. The contiguous memory layout of arrays is used by sequential traversal to make data access patterns that are easily kept in the CPU for each element of the array, from the first to the last. This way of knowing is easy, natural, and Sequential Traversal: We go to the last spot (size-1) with this method: It's usually implemented with a loop that has an index variable that goes up by one from the first position (which is usually 0). The standard deviation of the array is given by array; size = for i from 0 to size-1: process(array[i]); This uses memory access patterns that are good for caching, so going through them one by one is very fast in practice. taken to cross it can't be avoided. However, this constant factor is generally not very big because the array has n elements. There is a straight line between the size of the array and the time it takes to visit each part at least once. DILANOS: Sequential traversal takes O(n) time, while reverse traversal takes O(n) time.for operations requiring elements to process them in reverse order, or for constructing algorithms that function from the back of an array. last position to first. This method is especially helpful Reverse Traverse visits array

position in reverse order starting from decrementing instead of incrementing:

The implementation is almost identical to the sequential traversal, but with the loop index

integer, size:integer): function reverseTraversal(array:

for i from size-1 down to 0:

process(array[i])

Forward traversal in some hardware architectures. Is generally efficient because of good memory accesses. But it is less cache-efficient than As in the case of sequential traversal, reverse traversal has a time complexity of O(n)

Interval-Based Traversal

Algorithm that visits every element of the array at a particular interval instead of visiting every element in sequence. It can also be used to sample large arrays, to implement algorithms that operate on strides, or on elements Denver has two more games to lifecycle that gentle, and interval-based traversal is implementation uses a modified loop with step The > 1:

size, interval): def intervalTraversal(array,

0 to size-1 in steps of interval: for i =

process(array[i])

The interval is inappropriately selected focused on data distribution checked; this can greatly reduce the processing time for large arrays. But it risks losing important patterns or values when traversal (k = interval size) In cases where all elements need not be O(n/k) time complexity for interval-based

Conditional Traversal

Be visited, skipping portions which do not meet the required conditions. This is particularly effective for operations where you want to operate on only a subset of the array elements based on their values or their Utilizing conditional traversal; this way only allows certain elements to traversal with conditional checks: The implementation intersperses standard size, condition): def conditional Traversal(array, for i between 0 and size-1: If condition(array[i]) is true:

Process ([i] in array) elements.

Still is O(n), but now the processing time depends on how many elements match the condition. When only a small percentage of elements satisfy a particular condition, traversing only the list of target elements can greatly reduce the effective time complexity associated with a nested loop where the inner loop considers all Note that the time complexity for the traversal part of Multiple Arrays in Parallel Traversal visit corresponding elements in them in lockstep. This pat Parallel traversal means processing multiple arrays at the same time and multiple arrays, for example, vector addition, comparison, or merging. ern is critical for operations that hinge on combining elements from that iterates over all the arrays: We can implement them with one loop size, numArrays): def parallelTraversal(arrays, for i from 0 to size-1:

for j from 0 to numArrays-1:

process(arraysj)

As the bottom for many numerical computing algorithms. O(n*m) n= no of elements in each array m= no of arrays. Such parallel traversal is especially handy for element-wise operation and usually serves Time complexity -

Iterator-Based Traversal

iterators, these iterators implement a common interface for iterating over collections, as arrays, allowing cleaner and more readable code. of traversing an array. Ideally data bind is a powerful tool that can improve the usual code reference through Most modern languages feature iterator mechanisms that abstract away these specific details the implementation depends on the iterator mechanism for the language, but usually rely on built-in structures like “for-each” loops:

iterator Traversal (array) function

for element in array:

process (element)

Performance cost over raw index-based traversal for some languages, though. types to work on, all without compromising on time complexity. It may have minor Using iterators for traversing collections provides added advantages of simpler syntax, error handling, wider range of collection 1-D Arrays Combining Operations in Data integration processes, and set operations. There are several merging methods available, or patterns. This operation is essential for a lot of algorithms, including sorting algorithms (e.g

merging, on the other hand, pertains to merging elements from multiple arrays into one based on specific rules

Basic Merging of Two Arrays

Preserves the relative ordering of elements within each source array while not imposing any additional ordering across the arrays. It The simplest of the merging operations appends one array to another, yielding a new array with elements from both include:

Common implementation methods

1. Making a new array whose size is the same as the sum of the sizes of the source arrays

2. Moving all the items in the first array to the start of the second array

3. Moving all items in the second array to spots after the items in the first array (size2): define basic merging array1 with array2 of size 1 and result Size is size1 plus size2, which equals a new collection of size (result Size) results for each i from 0 to size1-1:

If i is between 0 and size2, the code will run as follows: result [size1 + i] = array2[i].

Return the answer

The is also O (n+m) the sizes of the two nput arrays are n and m. The amount of room needed for It takes O(n+m) time to do this basic merge, where Sets of A Brief Look at Arrays: Multi Dimensional goes one step further by letting data be kept in more than one dimension, like in tables, matrices, or even more complicated formats. Allow us to store more than one value of a certain type in a variable with the same name. There are times when one-dimensional arrays are useful, but multidimensional arrays are better in many situations. When you're computing, arrays are a useful type of data thatrepresent and work with complex relationships between data. in a structured manner makes them especially useful in fields such as mathematics, image processing, game development, scientific simulations, and data analysis.

2.3 Multi-Dimensional Arrays: Initialization, Accessing, and Implementation

Three, four or more dimensions, although it becomes quite difficult to visualize more than three dimensions. Two-dimensional arrays, which are akin to tables with rows and columns. However, arrays can be an "array of arrays." The most popular form is multi-dimensional

array is basically a collection of arrays, so you can think of it as access each one of its elements. Values and the process continues into what we call hyperspace. An array's dimensionality refers to the number of indices required to distinguish and mathematically speaking, a 2-dimensional array is matrix, 3-dimensional is cube of arrays initialization Multidimensional in a few common programming languages. multi-dimensional array initialization is a lot similar, just had either one or two syntax changes. In this article, we will go through the various types of initialization approaches with their implementation Whereas in languages, and Basic Initialization Instantiating arrays typically as the data to be stored and the number of dimensions. Declarable multi-dimensional the other hand, in C/C++, we can declare and initialize a 2D array like so.

```
// Declaration
```

Number of columns. First dimension is the number of rows, while the second dimension is the

```
// Initialization with values
int matrix3 = {
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12}
};
```

is somewhat similar in Java: The syntax

```
// Declaration
= new int3;  int matrix
// Initialization with values
int matrix = {
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12}
};
```

of writing the same: Python, which is designed to be simple, has a more compact way

Using nested lists

```
matrix = [
[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]]
```

Does the same thing as Python: Let matrix in JavaScript = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]; Dynamic Initialization or comprehensions on some input or computation. Loops can be used to make this happen. We might need to set up arrays based

In C++:

int rows = 3; int cols = 4; = new int[rows]; int matrix i = 0; i For(int | Array. for (let j = 0; j < {cols; j++));

Not uniform or sharp Arrays don't always have the same length. the exact same size in all ways. “Jagged” arrays are ones whose sub-arrays It is not necessary for all multidimensional arrays to have

In Java:

= new int3; int jaggedArrayjaggedArray[0] = new int[3];
jaggedArray[1] = new int[5];
jaggedArray[2] = new int[2];
// Initialize with values

5}. You should also be able to invoke jaggedArray[2] to return {9, 10}. You will want to ask jaggedArray[0] to return {1, 2} in the first iteration, jaggedArray[1] to return {3, 4,

In Python:

```
jagged_array = [
[1, 2, 3],
[4, 5, 6, 7, 8],
[9, 10]
]
```

Higher-Dimensional Arrays

as you extend beyond two dimensions. This sort of nesting continues array in C Three-dimensional

```
int cube2[4] = {
{
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12}
},
{
{13, 14, 15, 16},
{17, 18, 19, 20},
{21, 22, 23, 24}
}
```

};

In Python:

```
cube = [
[
[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]
 [13, 14, 15, 16],
[17, 18, 19, 20],
[21, 22, 23, 24]
```

Libraries Multi-Dimensional Arrays Specialized

and efficient multi-dimensional array implementations for scientific computing and data analysis. Specialized libraries provide more powerful

provides the ndarray object in Python: NumPy

```
import numpy as np
# Create a 3x4 array
matrix = np.array([
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
# Or create an array of zeros and fill it
matrix = np.zeros((3, 4), dtype=int)
for i in range(3):
 for j in range(4):
matrix[i, j] = i * 4 + j + 1
```

similar capabilities: math. libraries of JavaScript, for example js provide In

to install 'npm install mathjs' math - need

12]]); const matrix = math. matrix([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11,

from the Multi-Dimensional Array How to Access the Elements

indexing). creating it, a multi-dimensional array can only be accessed by specifying the index for each dimension. Most programming languages use 0-based indexing (the exception being MATLAB and R, which are based on 1-based After

Basic Access

In C/C++:

// Retrieve element at 2nd row (index 1) and 3rd column (index 2) (value: 7) int value = matrix1;

In Python:

as in C/C++ value = matrix1 # Access same way

another powerful idea called comma notation: In Python, NumPy offers

Equivalent of matrix1 value = matrix[1, 2] #

Through Multi-Dimensional Arrays Iterating

in 2D arrays to iterate through all their elements. We usually use nested loops

In C/C++:

```
for(int i = 0; i< 3; i++) {
for(int j = 0; j < 4; j++) {
printf("%d ", matrixi);
}
printf("\n");
}
```

In Python:

```
for i in range(len(matrix)):
len(matrix[i])): for j in range(0,
print(matrixi, end=" ")
print()
```

NumPy, you can take more compact approach: With

```
for row in matrix:
for element in row:
print(element, end=" ")
print()
```

Or even more concise

```
for row in matrix:
print(" ". join(map(str, row)))
```

Slicing and Dicing

letting you take sub-arrays. Certain languages and libraries allow you to slice multi-dimensional arrays,

In Python with NumPy:

rows and columns 1-3 Select first two

```
sub_matrix = matrix[0:2, 1:4]
```

or columns Extracting full rows

```
= matrix[1, :] # 2nd row row_2
```

matrix[:, 2] Third column column_3 =
Boundary Checking
behavior, or cause your program to crash. Also, protect yourself against access outside of bounds with arrays, which can return a runtime error, undefined
in C/C++, so if there is out of bounds access, we encounter segmentation fault: There is no boundary checking
checking dangerous: no bounds
access that you can't catch at compile time (in C/C++) int value = matrix5; // Out of bound
is the reason, that in Java when we access an out of bound index it throws an ArrayIndexOutOfBoundsException: This
try {
an exception int value = matrix5; // Will throw
e) { } catch (ArrayIndexOutOfBoundsException
of bounds!"); System. out. printIndex() // effectively println("Index out
}
out-of-range access, Python raises an IndexError: Even for
try:
raise at IndexError value = matrix5 # Will
except IndexError:
print("Index out of bounds!")
multi-dimensional arrays Changing elements of
indices and set a new value. pattern applies to modifying elements. You provide the The same
In C/C++:
// Sets the 2nd row, 3rd column value to 42 matrix1 = 42;
In Python:
 # The same way matrix1 = 42
Using NumPy:
= 42 # Using parentheses notation matrix(1, 2)
Modifying Slices
than one elements at once. Some languages and libraries provide slicing capability to change more
In Python with NumPy:
with 0 Fill in the first row
matrix[0, :] = 0

specific value to a 2x2 block Assign a
matrix[1:3, 1:3] = 99
Arrays Operations on Multi-Dimensional
built-in functions for common ARRAY things. Other languages and libraries with
In Python with NumPy:
Sum of all elements
total = np.sum(matrix)
Sum along rows or columns
The sum of every row row_sums = np. matrix.sum(axis=1)
of each column col_sums = np. matrix.sum(axis=0) # Sum
Mean, min, max
mean_value = np.mean(matrix)
min_value = np.min(matrix)
max_value = np.max(matrix)
Transpose (swap columns and rows) #
transposed = np. transpose(matrix)
or
transposed = matrix.T
In MATLAB-like languages:
% Sum of all elements
total = sum(sum(matrix));
% Sum along rows or columns
row_sum row_sums = sum(matrix, 2); % Each
2], size(mat,2)); % Treating as two by two groups of 1, 1, 1, 2, 2, 2,...
do. from_matrix = mat2cell(mat, [1 1
% Mean, min, max
mean(mean(matrix)); mean_value =
min_value = min(min(matrix));
max_value = max(max(matrix));
% Transposition
transposed = matrix';
Array 2.2 Alternate Implementation — Multi Dimensional
understand memory usage and optimizing performance when working with big datasets. Knowing how to work with multidimensional data is really important to better
Memory Layout

programming languages: multi-dimensional arrays are stored in either row-major or column-major order in most

1. **Row-Major Order** (used by C, C++, Python): Elements of each row are stored contiguously in memory.
2. **Column-Major Order** (used by FORTRAN, MATLAB, R): Elements of each column are stored contiguously.

with a shape of 3×4 in row-major order, the arrangement in memory would looks like this: For a matrix

7 8 9 10 11 12] [1 2 3 4 5 6

In column-major order:

2, 6, 10, 3, 7, 11, 4, 8, 12] [1, 5, 9,

Addressing Formula

j is the jth column of the matrix, and c is the total number of columns in the matrix. Here r denotes a dimension of the array, 0 is the ith row of the matrix,

2D array: The row-major form of a

can compute the cell address as follows: Finally, we

array in column-major order: In the case of a 2D

of the specific element within the 2D array. You can use the following formula to calculate the address extend similarly in higher dimensions. These formulas cache efficiency Maximizing example, thanks to how CPU caches are built, allocating arrays and looping over them in the order they show in memory (in line with the memory layout) improves performance due to cache locality. For

In C/C++ (row-major):

ordering) — Good cache performance (row-major

0; i= 0 for(int i = && row = 0 && col

MCQs:

1. **What is an array in programming?**
   a) A collection of elements of different data types
   b) A collection of elements stored at contiguous memory locations
   c) A special type of loop
   d) A pointer variable
2. **Which of the following correctly declares an array in C++?**
   a) int arr;
   b) int arr[5];
   c) arr[5] int;
   d) array int[5];

3. **What is the index of the first element in a C++ array?**
   a) -1
   b) 0
   c) 1
   d) 2
4. **What is the time complexity for accessing an element in an array?**
   a) O(1)
   b) O(n)
   c) O(log n)
   d) $O(n^2)$
5. **Which of the following operations can be performed on a one-dimensional array?**
   a) Insertion
   b) Deletion
   c) Traversal
   d) All of the above
6. **What is the disadvantage of arrays?**
   a) Fixed size
   b) Sequential memory allocation
   c) Slow data access
   d) Cannot store multiple elements
7. **What is the best method for searching an element in a sorted array?**
   a) Linear Search
   b) Binary Search
   c) Jump Search
   d) Fibonacci Search
8. **How are multi-dimensional arrays stored in memory?**
   a) Row-wise (Row-major order)
   b) Column-wise (Column-major order)
   c) Both A and B depending on compiler settings
   d) Randomly
9. **Which of the following is true about passing an array to a function?**
   a) The entire array is copied to the function
   b) The function receives a pointer to the first element of the array

c) The array cannot be passed to a function

d) Arrays are passed by reference only

10. **How can an array be initialized in C++?**

a) int arr[3] = {1, 2, 3};

b) int arr = {1, 2, 3};

c) arr[3] = {1, 2, 3};

d) int arr(3) = {1, 2, 3};

Short Questions:

1. What is an array, and why is it used?
2. How are one-dimensional arrays declared and initialized?
3. Explain the concept of indexing in arrays.
4. What are the advantages and disadvantages of arrays?
5. How do you pass an array to a function in C++?
6. Write a C++ program to insert an element into an array.
7. What is the difference between traversing and merging an array?
8. How does binary search work on an array?
9. Explain the concept of multi-dimensional arrays with an example.
10. Write a C++ program to delete an element from an array.

Long Questions:

1. Explain the concept of arrays and their types in detail.
2. Discuss the different operations on one-dimensional arrays with examples.
3. How does insertion and deletion work in an array? Provide algorithms.
4. Compare linear search and binary search for array searching.
5. Explain multi-dimensional arrays, their initialization, and applications.
6. How can arrays be passed to functions? Write a C++ program demonstrating this.
7. Discuss the advantages and limitations of arrays in data structure applications.
8. Explain the concept of merging two arrays with an example.
9. Describe the difference between a stack and an array.

10. Write a C++ program to perform all basic operations on an array (insert, delete, traverse, merge).

Module 3
STACK

LEARNING OUTCOMES

- Understand the concept of stacks and their operations.
- Learn about stack implementation using arrays and linked lists.
- Explore applications of stacks such as reversing a string and evaluating expressions.
- Understand stack-based algorithms for infix to postfix conversion.
- Learn about queue operations and their implementation using arrays.

Unit 6: Introduction to Stack

3.1 Introduction to Stacks and Operations on Stacks

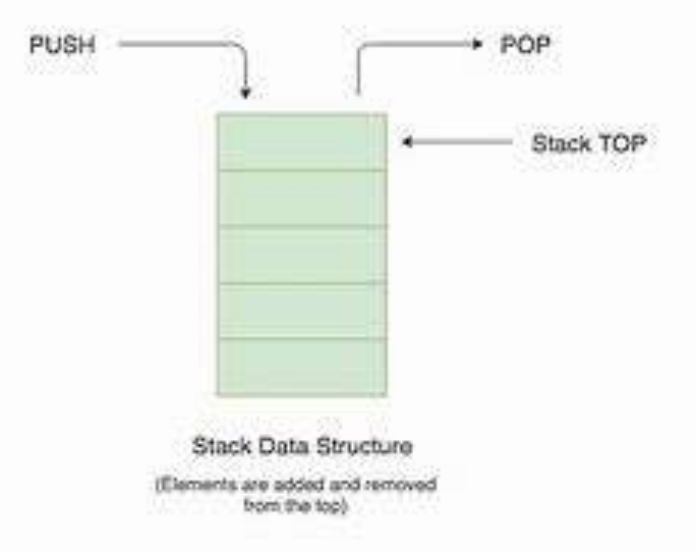

Figure 7.1: Stack Data Structure
[Source: https://th.bing.com/]

This characteristic of stacks forms the basis for their use in many algorithmic processes, such stack. In a stack, the most recently added element is removed first a bigger scope and as we handle data. One of the most visually simple yet most useful structures is the Efficient computing is built on data structures that help us make the best use of space and time complexity, at both functionality of stacks is their limited access pattern, which makes many complicated problems simple(since it enforces an orderly way of operations). functions are supplemented with additional operations such as peek (to inspect the top element of the stack without removing it), and is Empty (to determine whether the stack still contains elements). Purer top of the stack) give a limited way to access data. These core are simple but powerful. The core operations push (to place an item on top of the stack) and pop (to retrieve an item from the Operations of the stack browser history. Management and function calls. Even in regular applications, stacks enable the undo feature in text editors and keep structures. Stacks are used by operating systems for process between different notations and also to evaluate arithmetic expressions. Stacks are used in compiler design to check the syntax and manage nested science. In expression evaluation, they are mainly used to convert Stacks are used across various aspects of computer for a given use

case. list implementations can behave with a dynamic size, at the cost of an additional memory overhead. Understanding these tradeoffs is critical to choosing the right implementation can implement stacks using two methods one is array based and other is linked list based, both of them have their own advantages and disadvantages. While array-based implementations can offer constant-time access, they may be limited in size, whereas linked Implementing Stacks programmers a beginner exploring data structures for the first time, or an experienced programmer searching for a more in-depth understanding of stacks, this article provides a complete exploration of this foundational data structure. data structures and algorithms and their usage in solving complex computational problems. So, whether you are will cover various properties, operations, implementations and applications of stacks. Additionally, we will demonstrate how stacks can work hand in hand with other To begin with, the next few rounds in the world of stacks

Definition and Core Concepts

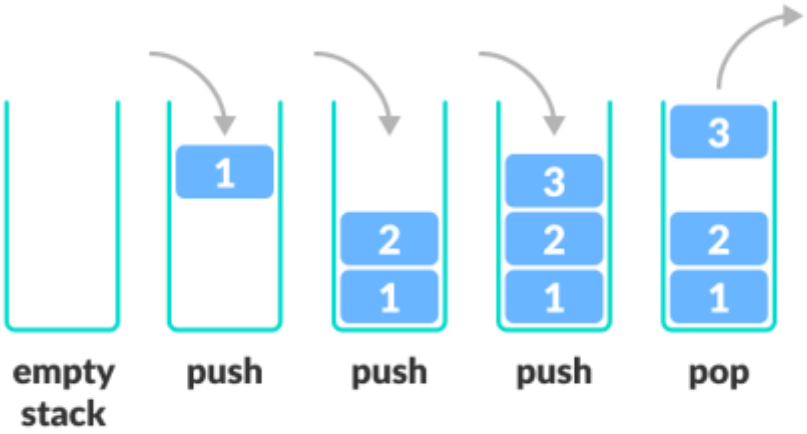

Figure 8: Stack Push and Pop Operations
[Source: https://cdn.programiz.com/]

Use makes them a vital data structure to know, as they can be useful for solving specific types of problems. like a pile of books: you can only add or remove books from the top of the pile. You can imagine a stack as being stack is a linear data structure based on Last-In-First-Out (LIFO) organization. Stack follows Last In First Out A the bottom of the stack has the first element added and will be the last to

be removed (when the stack is emptied fully). entry of a stack is the only one which all operations are performed with respect to, thereby enforcing the LIFO behaviour. In contrast, top, which refers to the latest item added to it. The top A stack has a contribution of this level of restriction is a clearer and more predictable state, preventing errors early with complex operations. to make more algorithms simpler by preserving a certain order of processing. Another allows access to only the top element. At first, the restriction may feel limiting, but it helps While arrays or linked list provide position access to an element, stack the operations performed on it. of the current stack; when we pop an element, we take it from the top. By this visual, we can visualize the LIFO behaviour of stack and as a vertical structure where the elements are arranged one on top of the other. When we push a new element, we insert it on top We can visualise a stack call another, the state of the current function is pushed onto the stack; when the called function returns, the state of the calling function is popped from the stack, execution resumes. is a pattern of access and manipulation seen in other areas of computer science. The call stack in programming languages, for example, uses the stack principle: when one function is more than four words; it is a data structure. It the stack usage in algorithms and system implementation. In order to appreciate how it works and why it's preferred in some cases.

Basic Stack Operations

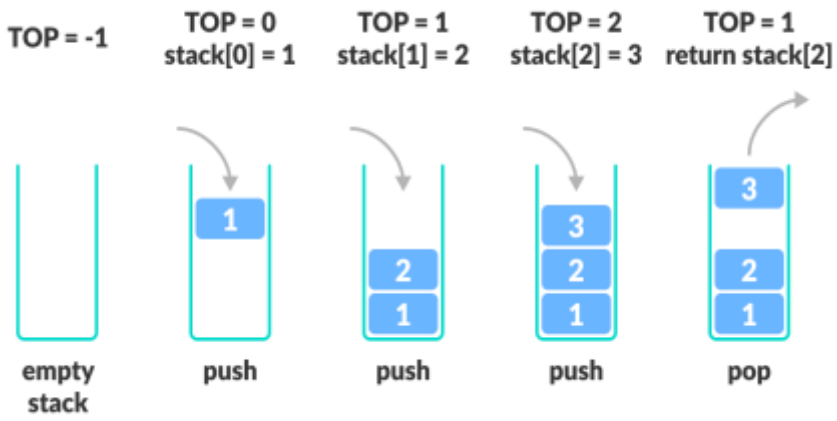

Figure 9:Working od Stack Data Structure
[Source: https://cdn.programiz.com/]

Discover and manipulate the stack data structure. of basic operations that can manipulate the elements of the stack by following the LIFO principle. These operations provide an interfacing through which programs can The concept of a stack is simply defined through a series

Push Operation

That it runs in constant time irrespective of stack size. The push operation generally has a time complexity of O(1), indicating we use the push operation It then increases the size of the stack by one and updates the top reference to point to the To add element to the top of stack, the push operation may reallocate memory if the implementation exceeds the storage. e.g., getting stack overflow in case of fixed size implementation. In dynamic implementations, Push Operation in Stack: When we perform a push operation, we may come across certain edge cases,

Pop Operation

Operation, like push, usually has time complexity of O (1). Top reference to refer to the next element in the stack. Pop to remove the top element of the stack and can return it to the caller. This operation reduces the size of the stack by one and makes the pop operation is used handling or exception throwing mechanism is required. The edge case of stack underflow, which means trying to pop an element from an empty stack. In case of such scenario, proper error similarly, during a pop operation, we also need to consider

Peek (or Top) Operation

The time complexity of the peek top of the stack without removing It does not modify the size or structure of the stack, only that it returns the the peek operation (known as top in some implementations) retrieves the element located at the must take care of the case of an empty stack. As with pop, peek Similar to Pop, Peek operation, is used to see the top element, before pop or any other is Empty Operation provides for checking to avoid underflow errors that are by checking before performing pop or peek operations. Returns Boolean: true if the stack is empty else false. This is Empty operation inspects whether the stack has any elements. Command which the is Full Operation a push operation to prevent overflow errors. This operation also helps abandon full Returns true if and only if this stack

In a stack with a fixed size stack implementation, is Full checks if the stack is

Size or Count Operation

Do base on where you are in the stack. This operation is helpful for tracking how the stack is growing, so you can decide what to the size operation returns how many elements are in the stack at any

Clear Operation

Simply when we are done with the current set of elements. Elements in stack leaving it in clean state. This is useful in scenarios where we want to reuse a stack for a different use case or Clear operation which clears all a number of scenarios. Various ways, they allow us to create complex algorithms and solve virtually any problem. The restrictive access pattern enabled by these operations (you can only interact with the top element of a stack) is what grants stacks their unique behaviour and applicability in basis for stacks.

Implementation Approaches

Implement stacks, each offering benefits and trade-offs. There are two widely used implementations, array-based and linked Different underlying data structures can be used to

Array-based Implementation

Index of the top element, and push and pop operations respectively increase or decrease this index. Array to save elements is called an array-based stack. A variable record the A stack implemented using an Implementation are: Pros of array-based

- Simplicity and ease of implementation
- Efficient memory usage with minimal overhead
- Excellent locality of reference, leading to better cache performance
- Constant-time access to the top element

Limitations of array-based implementation include:

- Fixed size in languages with static arrays, requiring size estimation beforehand
- Potential for overflow if the stack grows beyond the array's capacity
- Need for resizing strategies in dynamic implementations, which can occasionally lead to O(n) time complexity for push operations

Here's a pseudo code example of an array-based stack implementation**:**

```
class Array Stack:
initialize(capacity):
stack  = new array of sizecapacity
top = -1
function push(element):
if top == capacity - 1:
throw StackOverflowError
top = top + 1
stack[top] = element
function pop():
if isEmpty():
throw StackUnderflowError
element = stack[top]
top = top - 1
return element
function peek():
if isEmpty():
throw StackUnderflowError
return stack[top]
function isEmpty():
return top == -1
function size():
return top + 1
```

If we were implementing a real dynamic array, we would have checked before pushing an item and resized the array to double its size if the array was full. This generally means allocating a new array that can hold more items and then copying all items into the old array.

Linked List-based Implementation

Stack implementation can be done through the use of Linked List as well. The head of the linked list is which the top of the stack is where push and pop operations have been added respectively.

Advantages of linked list-based implementation include:

- Dynamic size that grows and shrinks as needed
- No concept of overflow (limited only by available memory)
- Consistent O(1) time complexity for all operations, without occasional spikes for resizing

Limitations of linked list-based implementation include:

- Higher memory overhead due to storage of pointers/references
- Potentially worse cache performance due to non-contiguous memory allocation
- Slightly more complex implementation compared to arrays

Here's a pseudo code example of a linked list-based stack implementation**:**

```
class Node:
def  initialize(data, next=None):
this.data = data
this.next = next
class LinkedListStack:
function initialize():
top = null
count = 0
function push(element):
newNode =  Node(element, top)
top = newNode
count = count + 1
function pop():
if isEmpty():
throw StackUnderflowError
element = top.data
top = top.next
count = count - 1
return element
function peek():
if isEmpty():
throw StackUnderflowError
return top.data
function isEmpty():
return top == null
function size():
return count
```

This means selecting the appropriate way of implementing your application. The selection of array vs, linked list based implementation comes up based on:

So a conclusion between the two will depend on your case if memory factor is the point you can go for an array based implementation, and if you wish to perform your list by the end linked list will help, Because a stack is a Last In, First Out (LIFO) data structure, it needs to support operations like push, pop, and peek that can be done in constant time complexity, O(1). But the underlying structure of data you choose greatly affects the efficiency, scalability and memory use. If the stack's maximum size is known beforehand and memory efficiency is of utmost importance, you could implement it using an array. In contrast to lists, where the overall bots are allocated, allow to use predefined memory and thus to use large amounts of memory and thus to generate memory locality. But, because it is an indexed data type, anyone can get the speed O(1) when getting the element at a certain position. Moreover, the push and pop operations take constant time in an array-based stack implementation, whereas these operations involve pointer manipulation, leading to additional overhead, in a linked list-based stack implementation. Array-based stacks are very memory efficient when memory is theory constrained and allocation behaviour is well defined. Think of a situation where a stack can be utilized for a compiler that does function calls. The maximum recursion depth is statically known, and an array-based stack can handle return addresses with no additional pointer overhead. However, an array-based stack can only hold a set amount of data because they have a static size. On the other hand, if the maximum size of the stack is left underestimated, the program can run out of space and encounter a stack overflow error. On the flip side, if the stack is underestimated, capacity is wasted, as a large part of the allocated space might go unused. Although dynamic resizing strategies (e.g., increase the size of the array when the array gets full) can work around this limitation, such operations incur an additional overhead. The time complexity for resizing the array involves allocating a new chunk of memory and copying the contents of the old array into it: O (n). In applications with tight performance requirements where predictability is key, this may be an issue.

For particular use cases where the stack size may be highly variable or unpredictable a stack implementation can be even better as a linked list. Unlike arrays, linked lists provide dynamic memory allocation, meaning that the stack can grow and shrink as necessary without

reallocating and having to worry about that. The first of each node in linked list-based stack consists of data and the second of it consists a pointer to next node. Their dynamic nature makes them particularly useful in situations in which the number of elements in the stack changes often. Take the web browser back button for example. Example would be whenever a user opens the page and navigates to the different page then the current visited page will be pushed onto the stack. When the user taps the back button, the last page was popped of the stack. A user may browse from 10 - 200 pages at a time, and as a result using a stack based on a linked list would be a better choice as it allocates the required space as needed. DFS in graph traversal is another such example, where the recursion depth changes depending on the depth of your graph. With a linked list implementation, the stack expands as needed, preventing stack overflow errors that are common when using a fixed-size array. However, a stack based on a linked list also has some disadvantages. The biggest issue is memory overhead. Linked list: Each node requires extra space to hold the pointer to the next node. This can be crucial in environments with limited memory, in which every bite of memory must be put to efficient use. Moreover, in a linked list, pointer dereferencing adds extra computation overhead when retrieving an element as opposed to a direct array indexing, which might lead to certainly slower stack operations. In other words, access patterns become important to consider, as do the potential overheads of resizing operations in an array-based implementation. Certain applications require real-time performance, and any unwanted latency caused by memory being reallocated may prove fatal. Deterministic execution time is important in real time embedded systems like avionics software or automotive control systems. In these situations, an array-based stack may be better because it can ensure O(1) time complexity for both push and pop without needing to dynamically allocate and deal locate memory.

But performance wise cache locality matters. Arrays give better cache performance because their elements are stored in contiguous memory locations. Linked list nodes, on the other hand, can be located anywhere in memory, resulting in a potential increase in cache misses and greater latency for accessing data. This becomes quite relevant for use-cases involving HPC applications where cache optimizations are key to achieving peak efficiency. In memory-limited environments,

the additional memory overhead of the linked list nodes might be a strong argument to avoid that implementation. In resource-constrained systems such as embedded systems, mobile devices, and Internet of Things (IoT) applications, memory efficiency cannot be ignored. The downside is that this increases the memory taken up by the individual nodes: the advantage of a linked list comes at the expense of memory (in other cases, such as random access). If, as an example, a pointer requires an additional 8 bytes per node, and your stack has thousands of elements, the memory overhead quickly adds up. Additionally, for devices with strict power requirements (battery-operated devices), more memory operations to manage a linked list pointer will also increase energy consumption. On the other hand, an array-based stack has less overhead, which makes it better in these cases. Each implementation has its own pros and cons, and the selection of data structure and its implementation depends on the use-case. So if memory efficiency, predictable size and cache performance are the dominating factors, array-based implementation is the one you want. If the focus is on dynamic size, resizing even during runtime, and preventing stack overflow, use implementation based on linked list. However, many modern programming languages and libraries have built-in stack implementations that abstract away these details, allowing developers to use the stack without needing to implement it themselves. But knowing about these underlying implementation approaches can help when choosing the right stack type and improving performance in performance-intensive applications.

Time and Space Complexity Analysis

Knowing the time: space complexity of stack operations lets you analyze the efficiency of algorithms that leverage stacks, as well as determine which implementation to use for a given need.

Time Complexity

You are focused on your level, and the sentences you get are often trivial. Based on the implementation, the efficiency of push/pop operations will be different based on the implementation of stack through array or linked list. In both cases, however, the time complexity for most of the basic stack operations is O (1), which makes them very suitable for real time apps. For example, the push operation, which adds an element to the top of the stack, takes constant time O (1) for both stack implementations (array & linked

list). But, in a dynamic array implementation, some resizing operations take O (n) time because all elements must be copied into a new larger array. This resizing gives it an amortized O (1) time complexity, which gives the effect of a series of push operations costing O(1) across multiple operations even though one resize operation might require O(n) operations. In a similar way, the pop operation, which removes the top element from the stack, also has an O(1), constant time, performance in both array-based and linked list-based implementations. Also since the top element is directly accessible due to its nature in both the structure, removal of the top element is an easy case and does not require shifting of elements unlike the other data structures where accessing the last element is not direct as in queues or arrays which do not allow direct access to last element.

Another O (1) operation is the peek operation, which returns the top element from the stack without removing it. This is because accessing the most recently inserted element does not require any iteration or traversal. Its accessibility is as simple as going to the last index in an array-based implementation, or simply pointing to the top node's data field in case of a linked list implementation. The operations is Empty and is Full also run in constant time, O(1). To validate whether a stack has any data, we need to check, if the head pointer is denial or, in the case of an array-based stack, the value at index head is null. On a fixed-size array implementation, one has to check if the stack is full by comparing the top index to the array capacity. Since these operations do not depend on the number of elements in the stack, they are very efficient. Similarly, the size of the stack can be obtained in O (1) time, if an additional variable is maintained to keep track of number of elements. Since you do not have to check how many elements are in the stack every time, this is a very efficient way to do it. In a linked list-based stack, unless count is maintained explicitly, determining the size would require traversing all nodes, thus, O(n) operation. Clearing a stack has O (1) or O (n) time complexity depending on the implementation. For an array-based stack, the clear operation can be done using o(1) time by simply resetting its top index to -1 or 0. In a linked list implementation, on the other hand, to clear the stack properly, we must find (the head), and deal locate each node individually to avoid memory leaks which is an O (n) operation.

Stack operations are efficient, with O (1) time complexity for most stack operations, which is one of the key reasons for their extensive use in applications such as function call management, expression evaluation, and backtracking algorithms. Their predictablebecause of their high performance, which is important in real-time systems that need fast access to recently used data. Time complexity is not the only aspect to analyse the stack efficiency; we also have to look at the space complexity. The space complexity of a stack is when implemented using an array or linked list. For array-based implementations, the space complexity will be O(n), where n is the initial capacity of the stack. It indicates that the stack does not need to be at its full capacity to reserve all of the memory, which can in the event that not all of that memory is used, induce minor inefficiencies. There is dynamic memory allocation in a dynamic array-based stack, however this leads to extra overhead as the cost of using a resize strategy, where the size of an array is typically increased by a factor, like 1.5 or 2 to allow for enough space for upcoming pieces. In contrast, a linked list-based stack has a space complexity of O(n), with n being the number of elements in the stack. But, unlike arrays, linked lists have higher space overhead due to pointers used for storing the linked list pointers in addition to the actual data. They will consume a little more space than array-based stacks, though they allow you to avoid resize operations and dynamically allocate memory as needed. The two stack implementations differ significantly in memory usage, but both have their own advantages. If the maximum number of elements is known at the beginning and memory is a concern, fixed-size array-based stacks are useful. They offer O(1) access and no extra memory overhead for pointers. On the other hand, they have the inconvenience of a fixed size, which prevents them from being as adaptable to unpredictable growth patterns. Dynamic array-based stacks overcome this constraint by increasing the size of the array only when necessary, allowing for dynamic sizing while still maintaining an average time complexity of O(1) for the push operation. But there is an overhead with resizing arrays, and that overhead increases the more you have to resize.

In contrast, stacks made from linked lists provide true dynamic memory allocation; space is allocated only when needed. They also don't need resizing, making them more suitable for unknown or

highly variable stack-size applications. Because the code that involves dynamic memory management has a little more overhead than fixed size structures and every pointer takes some memory space as well pair of temporary values used in the process. But in real work situations, the choice of which one to use, whether array-based or linked list based will depend on the case use. A fixed-size array implementation may be more appropriate if memory efficiency and predictability are major concerns. In fact, if flexibility and dynamic growth is required, then a linked list implementation would be more suitable as it will incur extra memory overhead. Both versions yield stack operations achieving O(1) time complexity for common methods including push, pop, peek, isEmpty, isFull and size retrieval, making stacks one of the most efficient and frequently used data structures in computation.

Performance Considerations

In addition to the theoretical complexity, several elements influence the practical performance of the stack implementations:

This means just like stack implementations; cache locality is on the critical performance path of many data structures. An array-based implementation of a stack, to ensure that the elements are stored in contiguous memory locations, ensures that, owing to the spatial locality principle, all the sequential accesses have better locality. This is important to know, cache line, as a result, if an access to a stack operation is made, so are accesses to the nearby elements that would also be pushed into the cache, thus reducing the cache misses and increasing the performance of the code. This is even more pronounced in scenarios where two large stacks are constantly interacted with, as good cache locality reduces the necessity for costly fetches from RAM. On the other hand, with a linked list-based stack, there is a very bad cache locality, as each node is allocated separately and spread in memory. This excessive use of pointers dereferences leads to more cache misses, thus degrading performance. Thus, an array-based implementation is generally preferred for applications that necessitate high-performance stack operations. Another important consideration that can favour stack implementations over heap allocators is the memory allocation overhead. For a stack implemented as a linked list, every call to push allocates a new element on the heap and every call to pop frees up that memory.

Since dynamic memory allocation is inherently less efficient than allocating memory on the stack, these operations lead to additional computational overhead. Moreover, High frequency of alloc and dealloc causes memory fragmentation that make performance degrade over time. This overhead is more meaningful for applications which have stack operations with higher frequency, like recursive algorithms or real-time systems. In contrast, a stack using an array does not cost this overhead as all elements are allocated in one round in a direct segment of memory. That is, push and pop, which can be performed just via index modification, are much faster than their linked list variants. However, this efficiency comes with a trade-off required by growth and would require resizing operations if the stack exceeds its originally allocated space.

Stacks based on dynamic arrays are challenged by the cost of resizing. If the underlying array is full, we must allocate a larger array using malloc, copy the elements of the current array to the newly allocated array with memcpy, and then free the memory previously consumed by the old array. This carries quite a performance penalty, especially when you will repeatedly slap an Item at a Frame that, for instance, just changed in size. Appropriate growth factor can help reduce impact of resizing. One approach could be doubling the size of the array when the current array has run out of space, this guarantees that the amortized cost of resizing will remain low. Even so, certain use cases call for such constant amortized costs to be unsatisfactory, as in real-time applications needing predictability. In those cases, it might be appropriate to reallocate enough memory or use other data structures. Although stacks implemented using linked lists do not need to be resized, this benefit is usually outweighed by the memory overhead and cache inefficiencies associated with these types of stacks. Stack operations are not the only phase whose memory usage patterns affect overall system performance. Which data structure you use to implement a stack impacts how the memory is allocated and accessed both in terms of efficiency as well as locality of reference, fragmentations etc. Another consideration is the performance of cache; an array-based stack is very cache-efficient but may have wasted memory if the allocated stack size is greater than what is actually used. This is especially important within embedded systems or memory-constrained environments where every byte

counts. Alternatively, a linked list-based stack dynamically allocates memory as it needs it, cutting down on waste but introducing memory allocation overhead and fragmentation. The compromise of each of these should be weighed against the context of the application. The actual performance of different programming environments may therefore also be influenced by modern memory management techniques like garbage collection or custom allocators.

There is no guarantee that the performance of stack operations in practice is as good as you might expect based on the details of the stack's implementation. Efficiency beasts differ from language to language, even at low levels, and much less at high levels, based on language-specific optimizations, compiler behavior and hardware characteristics. For example, languages like C and C++ implement manual memory management that enables tight optimizations, whereas garbage-collected languages such as Java and Python incur extra runtime costs for automatic memory management. Performance can also be affected by compiler optimizations, such as loop unrolling and instruction pipelining that reduce redundant operations. Furthermore, it also depends on computer hardware with many types of CPU cache hierarchies and branch prediction mechanisms. Processors equipped with sophisticated caching mechanisms might counteract some of the cache locality drawbacks associated with linked lists, whereas designs with constrained caching resources could amplify them. Therefore, performance testing on target hardware is the only way to make proper stack choices. The decision to pick either an array-based or linked list-based stack implementation comes down to various trade-offs. Since arrays are continuous in memory, they have the best cache locality, resulting in fewer cache misses and better overall performance. Linked list-based stacks, however, incur more memory allocation overhead due to the need for dynamic memory allocation for each element, making them less suitable for performance-critical applications. Inappropriate growth strategies often lead to costly resizes in dynamic arrays. In memory-constrained environments, the efficiency of stack operations is totally reliant on the behavior of memory allocation patterns and fundamental mechanisms to avoid memory fragmentation and overhead. Finally, real-world performance is further affected by language-specific implementation details, compiler optimizations, and the capabilities of

the underlying hardware. With a deep knowledge of these elements, developers can choose or build stack implementations in a way that best serves their applications. Knowing about these complexity aspects will give you a better insight to what stack implementation to choose based on your use cases. From the theoretical perspective, either implementation would also be suitable for the majority of use cases, as the time complexity for basic stack operations is constant time. The decision often boils down to practical aspects such as memory usage, how easy it is to implement and integrate with existing code, etc.

3.2 Applications of Stacks: Stack Frames, Reversing a String, Postfix Expression Calculation

In computer science, stacks are used primarily for handling expressions and syntax validation, making them an important component of any programming language. Stacks are primarily used in various applications like converting arithmetic expressions from one form to another. Infix notation (such as A + B * C), where the operators come between the operands, is the standard notation used in human-readable expressions, but computers generate and process expressions more efficiently in postfix (Reverse Polish Notation) or prefix (Polish Notation) forms. For instance, the expression A + B * C is expressed in postfix as A B C * + and in prefix as + A * B C. Converting from infix to postfix or prefix notation uses a stack-based algorithm that pushes operators onto the stack and places operands directly into the output sequence. This approach avoids the complexity of solving precursor rule and implies that there is no need for parentheses when computing expressions because machines follow this single method of evaluating the expressions. Stacks also play a fundamental role in the evaluation of expressions. Postfix and prefix expressions are easier to evaluate than infix expressions as they do not require rules for operator precedence. When using postfix evaluation, when an operand is processed, it goes on the top of the stack, and when an operator is processed, the appropriate number of operands are popped off the top of the stack and the operation is processed and the result pushed back onto the stack. For example, for the postfix expression 5 3 + 8 *, we first push 5 and then 3 onto the stack, then apply the + operator which results in an output of 8 that we push back. Then, we push 8 onto the stack, and apply the * operator to the two

topmost values on the stack, producing 64. This makes very efficient computations with a low memory overhead, and this is a widely used approach for interpreters and calculators.

Stacks are widely used in syntax checking (e.g., to check the matching of delimiters like parentheses/brackets/braces). Stack-based algorithms are employed by compilers and interpreters to check if syntax is valid. During source code parsing, every opening delimiter is pushed onto the stack, and every closing delimiter matches against the one visible at the top of the stack. If they do, the top element is removed; if they do not, a syntax error is raised. This is a very important technique which helps you in avoiding the demon known as Mismatched parentheses statement which cans sometimes even stop compilation or cause wrong runtime results. This same concept can be used with other constructs, as well, like making sure that an HTML tag is properly nested or an expression is balanced in a math computation. Stacks also have an important role in compiler design, especially in parsing methods. Parsing analyzes a sequence of tokens to give it the syntactic structure. A top-down method is recursive descent parsing, which uses function calls that resemble language grammar rules. Because function calls use the call stack, stack-based execution is well tailored for recursive descent parsers. In contrast, shift-reduce parsing is a bottom-up technique which uses a stack to hold symbols and reduce them according to the rules of the grammar for the language. It's a common approach in compiler implementation for context-free grammars, original in Yacc allows fast syntax analysis and code generation. Stacks play an essential role in these computing tasks, in addition to expression handling and syntax validation, such as backtracking algorithms, memory management, and function calls execution. Stacks are a last in, first out (LIFO) data structure, making them suitable for working with scenarios where data that was most recently accessed needs to be processed first. Stacks are widely used in computer science for tasks like arithmetic calculations, syntax checking, and parsing due to their efficiency and reliability in managing structured data.

Function Call Management

The call stack is possibly the most basic usage of the stack concept in computing:

There should always be a part of computer programming that is like Functions depend on a stack. The stack is used when a function has been called as it'll push the return address and local variables. That way, once the function has finished running, the program can pick up where it left off by popping these values off the stack. The stack of function calls is essential for keeping an organized and expected flow of execution in a program. The stack mechanism eliminates the complexity of tracking nested function calls, which would otherwise be highly chaotic and difficult to implement, resulting in unpredictable program behaviour. Recursion is a basic principle in programming, and this is handled by the function call stack. When dealing with recursion you have to keep adding stack frames and pop them when the function calls itself until you get to the base case. Now the recursion begins unwinding, and frames are popped from the top in reverse order. So far we have structured recursion onclick to ensure that we keep track of what we are trying to compute, and also to allow the repeated computation. Recursion is extensively deployed in mathematical computations, searching algorithms, and dynamic programming techniques, thereby, making the management of the stack an indispensable part of programming. Stacks are also employed by operating systems for context switching, enabling the multiple processes or threads working to switch between themselves seamlessly. Since process has their assignment task, when the operating system switches from one process to another, it saves the execution context, which contains the instruction pointer and register states onto the stack. When the process continues, these values are restored, allowing execution to resume without losing data nor causing inconsistent states. Context switching helps keep resources idle, so even if only one process or thread is using all the CPU resources in the environment, it utilizes them very efficiently in multitasking and multi-threaded environments, given that you have enough memory available in the system.

Stacks are widely used in the implementation of algorithms, especially for maintaining state or tracking progress. Depth-First Search (DFS) is one example of this type of algorithm; the stack is

used as a structure to keep track of the vertices to visit. This can be done using a stack data structure explicitly, or using recursion implicitly. DFS is building block to graph traversal, solving maze problems and network connectivity analysis. In the same way, backtracking algorithms, like solving the N-Queens problem or generating permutations, utilize stacks to keep a log of choices made. The last visited node is popped off from the stack when the path leads to no solution, then alternative nodes are traversed by repeating this step. Another important algorithmic technique is called a topological sorting, which uses stacks to iteratively traverse directed acyclic graphs (DAGs). The vertices are processed iteratively and pushed onto a stack in their completion order. This helps in resolving dependencies in the respective order, most commonly as tasks to be scheduled, based on dependencies, among others like build automation. Stack also is important in tree traversal algorithms. Tree processing can be done even from an iterative perspective because stack can be used to simulate the function call stack in the case of in order, preorder and post order tree traversal non-recursive implementations. Aside from algorithm implementation, stacks have a range of features in user interfaces and software applications. For example, in text editors and graphic design software, the undo mechanism is implemented using stacks with each edit operation added to a stack. When a user presses undo, the last operation is removed from the stack and the previous state is restored. It enables users to quickly undo accidental changes. Web browsers use a stack to keep track of the history of the navigation. This is because the back button retrieves the previous page using the stack: every visited page is placed on the stack. The feature improves user experience by allowing for seamless navigation to go back to previous pages. With mobile apps, a similar approach is taken; the activities representing each screen maintain a stack of previous states so that users can go through screens seamlessly. Stacks are integral to memory management in computer systems. You are the stack memory, which is used to store local variables, information about function calls, and temporary data needed while the program is running. Stack memory is managed automatically and is less error-prone than heap memory, since it does not require explicit allocation and deal location. So, when you call a function, it allocates a new stack frame, and when

the function ends, the stack frame is automatically deal located. This optimal usage of memory provides better performance while avoiding unnecessary consumption of memory. Improper stack usage, however, could lead to stack overflow error whenever there are too many recursive calls or deep function nesting that exceed the allocated size for the stack.

Stack data structure is also used in the exception handling mechanism of programming languages. In the event of an exception, a runtime environment unwinds the stack, looking for an appropriate handler. It checks from the most recent function call frame to the oldest until it finds a suitable handler. If none is found, it aborts with an error message. The Error Handler and Debugging Section: This structured approach provides robust error handling and debugging capabilities that make it easier for developers to trace issues and take corrective action. Within compiler design, stacks are extensively used for expression evaluation and parsing the expression. Stacks are also used by compilers to ensure that operators have the correct priority when being solved by the system, such as used in the conversion of infix notation to postfix notation or during parsing of a string. Postfix (reverse Polish) and prefix (Polish) notations are stack-based and can parse expressions with no need for parentheses. It is most notably used in calculators and expression evaluation engines to allow the efficient evaluation of mathematical expressions. Execution contexts are managed using stacks in virtual machines and interpreters as well. Byte code interpreters like the Java Virtual Machine (JVM) and Python interpreter rely on stacks to handle method invocations, local variables and operand evaluation. Such an approach draws a clear pattern of how the code will be executed, enabling simple interpretation and execution of the code. This approach allows to keep the stack-based execution model, thus reducing the complexity of managing the state of the memory and speeding up to inject faster code for a runtime (in case the interpreter has a slot, e.g.)

In the long run, stacks absolutely are critical data structures in programming, operating systems, algorithms, user interfaces, memory management, exception handling, and compiler design. Thus, their ability to handle function calls, recursion, context-switching, algorithmic state, and user interaction make them one of the fundamental building blocks of modern computing. This can allow for

efficient program execution, robust software design, and enhanced computational performance, so it is important to understand and leverage stacks properly with a stack, and how to evaluate postfix expressions using a stack. useful for situations where you need to handle things in reverse order relative to how they appear, or where you need to manage a sequence of operations and undo in a specific way. This section will cover some more real theoretical applications of stacks including how stack frames are used in executing a program, how to reverse strings computing.

Stack Frames

Runtime will manage the stack using a stack-based memory organization. a program, must be able to keep track of function calls, local variables, parameters and return addresses. The important modern applications of stack data structures are the concept of stack frames, or activation records. The computer, when executing one of the most often included call.

1. Parameters passed to the function
2. Local variables declared within the function
3. The return address (indicating where program execution should continue after the function completes)
4. The previous frame pointer (linking to the calling function's stack frame)

For each function when it finishes executing. function C is done, its stack frame pops off the stack and control returns to function B at just the right point in code. And this continues calls function B and function B calls function C, so the stack frames are pushed to the stack one after the other. Once nested function calls are considered. For example, function A This shows how natural using a stack is for this reason, especially when straightforward recursive function to calculate the factorial of a number: For example, consider a function factorial(n): if n <= 1:

return 1 else:

return n * factorial(n-1)

We would have something like this in the stack frames: So if we call factorial(4),

1. Call factorial(4): Push stack frame for factorial(4)
2. Inside factorial(4), call factorial(3): Push stack frame for factorial(3)

3. Inside factorial(3), call factorial(2): Push stack frame for factorial(2)
4. Inside factorial(2), call factorial(1): Push stack frame for factorial(1)
5. Inside factorial(1), return 1 (base case): Pop stack frame for factorial(1)
6. Resume factorial(2), compute 2 * 1 = 2, return 2: Pop stack frame for factorial(2)
7. Resume factorial(3), compute 3 * 2 = 6, return 6: Pop stack frame for factorial(3)
8. Resume factorial(4), compute 4 * 6 = 24, return 24: Pop stack frame for factorial(4)

Stack frames, it would be practically impossible to manage the complex flow of execution of modern programs. that each function returns to its correct caller, with local variables and context intact to the exact degree required. Without This stack-based strategy ensures function properly. Function's stack frame contains its own set of local variables, avoiding variable conflicts and allowing proper scoping rules. Such isolation is basic to modular programming, and enables techniques like recursion to important separation between function calls. Each Stack frames also give operating systems reserve a fixed amount of memory for this stack, which is why very deep recursion results in a "stack overflow" when we run out of the allocated space. known as the call stack or execution stack. Most programming languages and Stack frame memory management is usually done via a memory region

Essentially popping stack frames until a handler is found. In dealing with exceptions, when an exception is thrown, the executable must traverse backwards in the chain of function calls to find a proper exception handler — Knowing about stack frames explains concepts such as unwinding a stack (in an exception handler, for Reversing a String property of Last in First out, that is well-structured for reversing a stream of elements, like characters in a string. Application of stack data structures is string reversal. So, a stack is a data structure with the One more classic stack is simple: Anime: using stack to reverse a string:

Create an empty stack

1. Push each character of the input string onto the stack, starting from the first character
2. Pop each character from the stack and append it to a new string until the stack is empty
3. The new string now contains the characters of the original string in reverse order

Let's illustrate this process with an example. Consider the string "HELLO":

1. Initialize an empty stack: []
2. Push 'H': ['H']
3. Push 'E': ['H', 'E']
4. Push 'L': ['H', 'E', 'L']
5. Push 'L': ['H', 'E', 'L', 'L']
6. Push 'O': ['H', 'E', 'L', 'L', 'O']
7. Now pop and build the reversed string:
    - Pop 'O': Reversed string = "O"
    - Pop 'L': Reversed string = "OL"
    - Pop 'L': Reversed string = "OLL"
    - Pop 'E': Reversed string = "OLLE"
    - Pop 'H': Reversed string = "OLLEH"

Reversed version of our string initially "HELLO". In the end "OLLEH" is the in-pseudo code as below: This algorithm can be implemented easily

```
function reverseString(str):
stack = new Stack()
the stack  { [stack pop] for each char in str } // push all chars onto
for each character c in str:
stack.push(c)
reversedStr = ""
add to result  // Pop all chars,
while not stack. isEmpty():
stack pop()  reversedStr +=
return reversedStr
```

Stack based manner. For example, we can reverse a array or a linked list in a similar This reversal of strings can also be applied to other runs in O(n) as we have to store all characters in the stack. each character, we have one push operation, followed by one pop operation for each character. The space complexity also string. Here, for So,

time complexity of this string reversal algorithm is O(n), where n is length of through the elements in the reverse order of appearance. Order-reversal. We have seen how this would apply to a slightly more complex problem where we need to iterate Though there are built-in functions in most programming languages to reverse strings, this stack approach helps you understand the fundamental relationship between stacks and expression Evaluate postfix optimal for computers. Mathematical expressions, specifically postfix expressions (or Reverse Polish Notation or RPN).

1. Initialize an empty stack
2. Scan the postfix expression from left to right
3. If the current token is an operand (a number), push it onto the stack
4. If the current token is an operator, pop the required number of operands from the stack, apply the operator, and push the result back onto the stack
5. After scanning the entire expression, the stack should contain only one value, which is the final result

Let's evaluate the postfix expression 3 4 + 5 * step by step:

1. Initialize empty stack: []
2. Scan '3': It's an operand, push to stack: [3]
3. Scan '4': It's an operand, push to stack: [3, 4]
4. Scan '+': It's an operator:
    - Pop two operands: 4 and 3
    - Calculate 3 + 4 = 7
    - Push result to stack: [7]
5. Scan '5': It's an operand, push to stack: [7, 5]
6. Scan '*': It's an operator:
    - Pop two operands: 5 and 7
    - Calculate 7 * 5 = 35
    - Push result to stack: [35]
7. End of expression, result = 35

Here's the algorithm in pseudocode:

```
function evaluatePostfix(expression):
 stack = new Stack()
for each token in expression:
if token is an operand:
stack.push(token)
```

else if token is an operator:
// For binary operators
operand2 = stack. pop()
operand1 = stack. pop()
a) result = operate(token, b,
stack.push(result)
return stack. pop() // The final result
operand2): def applyOperator(operator, operand1,
if operator is '+':
return operand1 + operand2
if operator == '-': else
return operand1 - operand2
operator == '*': else if
return operand1 * operand2
== '/': else if operator
return operand1 / operand2
accordingly // implement rest_operator

It tends to be lower since operators pop operands off the stack. Complexity is O(n) since we go through each token exactly once, where n is the size of the expression. The worst-case space complexity is O(n) but postfix expressions support arithmetic binary operators only. The time usually, some of the benefits of postfix notation: The postfix evaluation algorithm shows

1. No need for parentheses or operator precedence rules
2. Expressions can be evaluated in a single pass from left to right
3. The algorithm is simple and efficient
4. No need for a separate parsing step

Constantly evaluate mathematical expressions. This property gives postfix notation the advantage to be especially useful in calculators, compilers and other systems that have postfix type language or similar forms before the generation of code. in an efficient way to get calculations done required fewer strokes after familiarization with the notation. This approach is useful in compiler design, as many parsing algorithms output infix expressions into a HP calculator famously had RPN (Reverse Polish Notation) because working with it pop the correct amount of values. off the stack. For operators that require more than two operands, we allow us to parse postfix notation. For

unary operators (operators that take a single operand, like negation), you simply pop one value, rather than two,

2 + 4 * + 3 -

1. Initialize empty stack: []
2. Process '5': Push to stack: [5]
3. Process '1': Push to stack: [5, 1]
4. Process '2': Push to stack: [5, 1, 2]
5. Process '+': Pop 2 and 1, calculate 1+2=3, push result: [5, 3]
6. Process '4': Push to stack: [5, 3, 4]
7. Process '*: *Pop 4 and 3, calculate 3*4=12, push result: [5, 12]
8. Process '+': Pop 12 and 5, calculate 5+12=17, push result: [17]
9. Process '3': Push to stack: [17, 3]
10. Process '-': Pop 3 and 17, calculate 17-3=14, push result: [14]
11. End of expression, result = 14

Equivalent to the infix expression 5 + ((1 + 2) * 4) - 3. The following expression is heavily. Power of stacks is clearly seen in postfix evaluation as multiple nested operations can be simplified and handled in a linear fashion. Many computer science algorithms, especially those related to compiler design and expression evaluation, rely on this technique The

3.3 Algorithm for Infix to Postfix Conversion and Postfix Evaluation

An algorithm that converts infix expressions into postfix form prior to evaluation. The conversion and evaluation illustrated the utility of stack data structures in terms of their power and capability to solve more complex computational makes the evaluation of expressions simple, but human beings are most used to infix notation with the operators occurring between operands (e.g., 3 + 4). To avoid such issues, we rely on Postfix notation

Infix to Postfix Conversion

The essential feature is the postfix notation, which gives the order of operand operations implicitly through the order of operators. Precedence rules and parentheses are not needed. to convert the infix expression into postfix notation, we have to take care of priority and parentheses. Output: In order Postfix algorithm uses a stack to hold operators temporarily and includes the following steps:

1. Initialize an empty stack and an empty result string
2. Scan the infix expression from left to right
3. If the current token is an operand, add it directly to the result string
4. If the current token is an opening parenthesis '(', push it onto the stack
5. If the current token is a closing parenthesis ')', pop operators from the stack and add them to the result string until an opening parenthesis is encountered (which is discarded)
6. If the current token is an operator: a. Pop operators from the stack and add them to the result string as long as they have higher or equal precedence compared to the current operator b. Push the current operator onto the stack
7. After processing all tokens, pop any remaining operators from the stack and add them to the result string

Precedence than addition and subtraction, and parentheses can be used to alter the default precedence. Multiplication and division typically have a higher by setting up a natural hierarchy of operations, you are able to manage E. For example, consider the infix expression A + B * C - D /

1. = "" # this will be our result stack = [] # to use as a stack result
2. "A" Scan 'A': Operand, append to result:
3. so push to stack: Stack: ['+'] Read '+': This is an operator, stack is empty
4. to output: “A B” Scan 'B': It’s an operand, add
5. ['+', "] Scan ": operator: higher precedence than '+', push to stack:
6. C” Scan ‘C’: It’s an operand, append to result: “A B

Compare with '+' (equal precedence), pop '+' and add to result, pop '-' to stack: Getting to "--' (push to stack as '-' has lower precedence than ") Scan '-': pop " and add to result, then ++

o Result: "A B C * +"

o Stack: ['-']

1. its an operand, append it to result: "A B C * + D" Scan 'D':
2. Push: ['-', '/'] Scan '/': Higher precedence than '-':
3. add to result: “A B C * + D E” Scan ‘E’: it’s an operand,

pinosta ja yhdisteleM! Last expression, vielämoppaajääneetoperaattorit

B C * + D E /” ● Pop ‘/’ and add to result: “A

append to result: “A B C * + D E / -” o Pop '-' and

valid postfix representation of the above infix exrpression. Thus the final postfix expression is A B C * + D E / - which is a

algorithm, in pseudocode: But here’s the

is a string with something like two operands and one operator, so function infix To Postfix(expression): expression

```
stack = new Stack()
result = ""
for each token in expression:
if token is an operand:
result = result + token + " "
else if token is '(':
stack.push(token)
else if token is ')':
while not stack. isEmpty() and stack. peek()! = '(':
result = result + stack. pop() + " "
stack. pop() // Discard the '('
else if token is an operator:
```

is Empty() and precedence(token) while not stack. This is because <=
precedence (stack. peek()):
result = result + stack. pop() + " "
stack.push(token)
while not stack. isEmpty():
result = result + stack. pop() + " "
return result
precedence(operator): def
operator == '+' or operator == '-': if
return 1
"*" or operator == "/": else if operator ==
return 2
else:
Popped. Only when it meets ')' return 0 // For '(' cannot be expressions well, it can even be extended to more complex cases like: This algorithm manages simple arithmetic

1. operators (i.e., '^', '**') 2] Exponentiation
2. minus or logical NOT) Unary operators (e.g., unary
3. cos(), sqrt()) Function calls (e.g. sin(),
4. of variables) multi-character tokens (e.g., multi-digit numbers, names
5. highest priority to exponentiation

3.4 Introduction to Queues and Operations on Queues

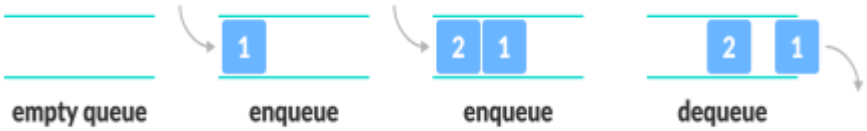

Figure 10:FIFO Representation of Queue
[Source: https://cdn.programiz.com/]

Queue is a type of data& structure in computer science that lets jobs be done in the order they were &, which is also called "First-In, First-Out" (FIFO). This means that the first thing that was added would be deleted first, just like people standing in line in real life. In everyday life, the queue data structure is used by the operating system to run tasks, by a web server to handle requests, by network packets to buffer data, and so on. Unlike stacks, which are Last-In, First-Out data structures, queues let parts be served in the same order they came in. A queue can be thought of as a list of things in order, with new items added at back and old ones taken away at the front. In other words, the oldest item in the queue would be dealt with first. But there are different types of lines, like the simple queue, the circular queue, the double-ended queue (deque), and priority queue. Dequeue takes something out of the queue, and Enqueue adds something to it. Peek gets the first item in line without taking it., and is Empty checks to see if the queue is empty. But some versions also offer the is Full operation, which can be used to see if the queue is full. Operations on queues: A system's performance depends a lot on how& well its queue processes work. This is especially true for time-sensitive tasks like CPU scheduling, network packet handling, and real-time processing systems. A queue can be made with both arrays and linked lists. When parts are taken out of the queue, the open hash table implementation wastes space because its size stays the same. In this case, circular queues are better because they make better use of the room. The problem with an array, on the other hand, is that it has a set size. With linked list, on the other hand, we could use its dynamic memory allocation to get around that problem. Managing memory and pointers is more work with linked lists, though. It's important to know different

types of queues so that you can handle data and make computations run faster. A regular queue uses the first-in, first-out (FIFO) model, while a circular queue lets the rear pointer circle back to the beginning of the array when it hits the & of its range. This stops array slots from being wasted. The double-ended queue (deque) gives you more options because you can add to or remove from both sides. Last but not least, a priority queue sorts items by importance instead of the order in which they were added. This makes it useful for many things, like task scheduling and pathfinding algorithms.

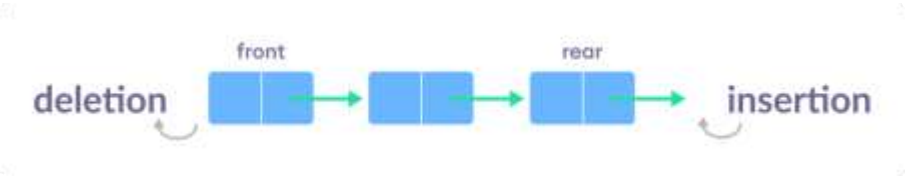

Figure 12:Simple Queue

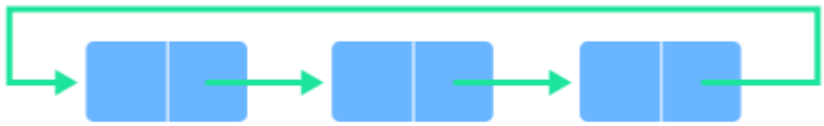

Figure 11:Circular Queue

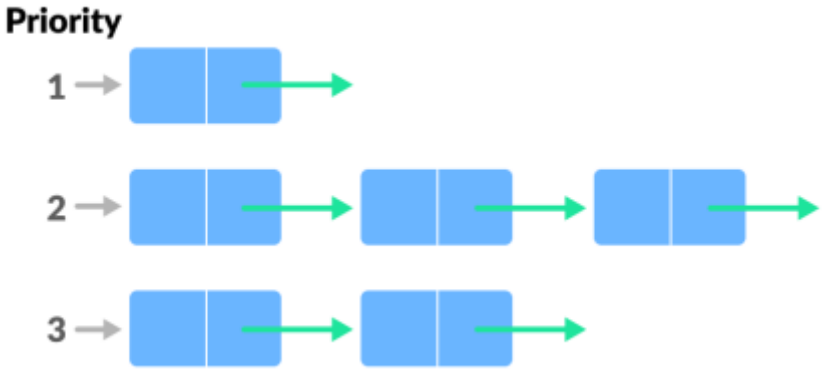

Figure 13: Priority Queue

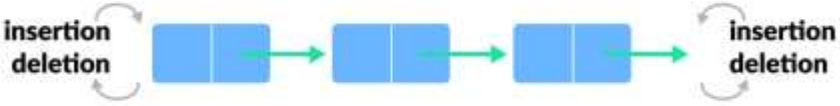

Figure 13: Deque Representation

3.5 An algorithm for adding and removing items from queue using arrays

In an array-based queue, first two actions (enqueue and dequeue) must be done while keeping the FIFO order in mind. So far, we've seen that because the array has a set size, we need to be extra careful not to let it overflow or underflow. Before adding a new element, a program should check to see if the queue is already full. An overflow mistake will happen if the information needs to be added to the queue but queue is already full. If not, rear pointer moves one space back, and the new element is saved at that new rear place. In a normal line, the space that was popped from the front is forgotten. This means that whenever something leaves the front of the queue, empty space is made that can't be used again, which is wasteful. The dequeue action takes something out of front of the queue. The non-empty queue needs to be reprimanded before it can be withdrawn. It can't remove if queue is empty, which is called an underflow error. If not, the front pointer is moved forward, which is the same thing as removing the first item from the list. But in an array-based queue, this leaves empty room at the front because it moves the index values around. We can use a circular queue to solve this problem by wrapping the numbers around. The following methods show how to add and remove items from an array-based queue:

As the first line says, a queue is a basic data structure in computer science. It works with the idea of "First In, First Out," or FIFO, which says that items are added to one & (the back) and taken away from other 7 (the front). Adding something to queue is called "enqueue," & taking something out of a queue is called "dequeue." These two actions are necessary for a queue to work as it should. You need to understand these algorithms in order to use queues in the real world. For example, you would use them to schedule a process in an operating system, handle requests in a web server, or handle jobs in a real-time system. The first thing we do in the enqueue process is

check to see if the queue is FULL. In an array-based linear queue, this condition is checked by comparing the back pointer to the queue's largest size. If the back pointer gets to the end of the array, it means the queue is full. Then you send back an overflow message because the element could not be added and the other elements in the queue have not changed. If the queue isn't already full, the rear pointer is moved forward to make room for a new element. After that, the new element is added at the number set by the back pointer. If deque was empty (meaning it didn't have any items in it before this action), front pointer is also set to the first index. process ends, keeping the integrity of the queue, and the insertion is done. This action takes something out of the queue. First, it checks to see if the queue is empty. Either the front pointer & rear pointer are equal, or the front pointer number is -1. This means that queue is empty. At this point, an underflow message is shown, process ends, and nothing changes. item at the back of the front pointer is returned and taken away when the queue is not empty. The front pointer is then moved to next item in the queue after item is removed. As you delete, if the list is empty, set both the front and back to -1 to show that there are no more items. Also, this makes sure that any future additions stay in the right place, which keeps the list correct. In computer science and the real world, queues are often used because they are good at handling sequential data. These can be done with arrays or linked lists; each has pros and cons. It's possible for fixed-size array-based queues to waste memory, but dynamic memory allocation, especially with linked list-based queues, makes memory allocation more efficient. Different types of advanced variations, like circular queues and priority queues, make queue processes even better for certain uses. Adding and removing items from a queue are the most important actions in this type of data structure. By following their algorithms, you can learn how to make efficient systems for processing data in a wide range of computing settings. These designs come in handy for many things, from keeping track of computer networks to keeping things in order during processes.

These actions make sure that the queue is in the right order and stop memory leaks. But, as we already said, when parts are taken out of a simple array-based queue, space is wasted. To get around this issue, circular queues wrap around the indices to make good use of room. To

make the best use of the space in the buffer, the rear pointer goes all the way back to the beginning of the array in this version. Operating systems, network traffic management, and models are all examples of places where queue management needs to work well. For managing tasks and resources, queues are used to solve a lot of problems in the real world. You can always make the queue more efficient by using circular queues or linked lists instead of a fixed-length array, since the linear structure of the queue has clear limits on the maximum size that can be set.

MCQ

1. Which of the following follows the "Last In, First Out" (LIFO) principle?
   a) Stack
   b) Queue
   c) Array
   d) Linked List
2. Which action does NOT occur in a stack?
   a) Push
   b) Pop
   c) Peek
   d) Queue
3. What is the time complexity of the "push" operation in a stack?
   a) O(1)
   b) O(n)
   c) O(log n)
   d) $O(n^2)$
4. Which of the following is NOT a common use of a stack?
   a) Function call management in recursion
   b) Reversing a string
   c) Converting postfix expressions to infix
   d) CPU scheduling
5. What happens when a stack exceeds its maximum capacity?
   a) The stack is automatically cleared
   b) An overflow error occurs
   c) The program continues running normally
   d) Elements shift downward
6. Which data structure is used to evaluate mathematical expressions?

a) Stack
b) Queue
c) Linked List
d) Graph

7. Which algorithm is used to convert an infix expression to a postfix expression?
   a) Kruskal's Algorithm
   b) Dijkstra's Algorithm
   c) Stack-based Conversion Algorithm
   d) Floyd-Warshall Algorithm
8. Which stack operation retrieves the top element without removing it?
   a) Pop
   b) Push
   c) Peek
   d) Fill
9. Which of the following correctly represents a stack operation sequence?
   a) Push(10) → Push(20) → Pop() → Push(30) → Pop()
   b) Push(10) → Pop() → Push(20) → Push(30) → Pop()
   c) Both A and B
   d) None of the above
10. What is the key difference between a stack and a queue?
    a) A queue follows LIFO, while a stack follows FIFO
    b) A stack follows LIFO, while a queue follows FIFO
    c) Both follow FIFO
    d) Both follow LIFO

A Few Questions:

1. Explain what a stack is and how it works.
2. What does it mean to "push" something in a stack?
3. What is the pop process, and when do you use it?
4. Where do stacks come in handy in real life?
5. Explain what stack overflow and underflow mean.
6. How can arrays and linked lists be used to make a stack?
7. Seventh, explain how to use a stack to reverse a string.
8. Describe the steps needed to change an infix expression to a postfix expression.
9. How does evaluating a postfix statement with a stack work?

10. What's the difference between a queue and a stack?

A Lot of Questions:

1. Give an example to show how the LIFO principle works in stacks.
2. Talk about algorithms for the different stack actions (push, pop, and peek).
3. Use an example program to show how to build a stack using an array.
4. Use a linked list to make a stack in your C++ program.
5. What does a stack do in recursion? Give an example to help you.
6. Explain how to use a stack to change infix expressions to postfix expressions.
7. Describe how stacks are used to analyze postfix expressions.
8. Compare how stacks and queues work and what they are used for.
9. Use an array to write a method for adding and removing items from a queue.
10. Talk about how stacks are used in real life in computer science

.

MODULE 4
LINKED LIST

LEARNING OUTCOMES

- Understand the concept and structure of linked lists.
- Learn about different types of linked lists (Single, Double, Circular).
- Perform basic operations on linked lists, such as insertion, deletion, and traversal.
- Understand sorting techniques such as Bubble Sort, Selection Sort, Insertion Sort, and Quick Sort.
- Learn different searching algorithms for linked lists and arrays.

Unit 9 Introduction and Basic Operation of Link List

4.1 Introduction to Linked List and Its Types (Single, Double, Circular)

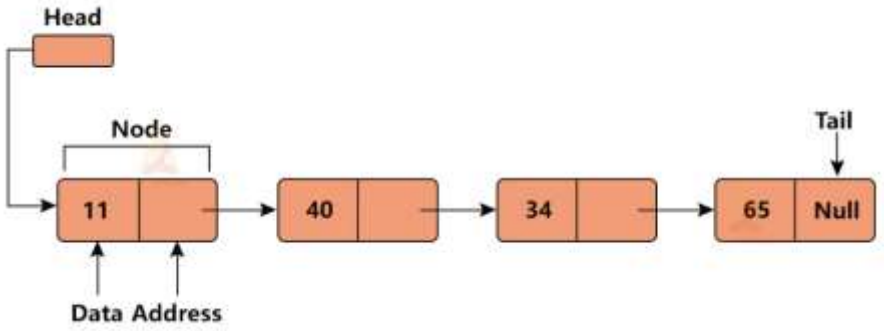

Figure 14.1: Linked List
[Source: https://techvidvan.com/]

One of most basic & important data structures in computer science is linked list. Arrays are the first data structure that most programmers learn, but linked lists have their own perks and features that make them useful in more than& one situation. Arrays take in blocks of memory that are all connected to each other. Linked lists, on the other hand, are made up of nodes that can be created at different locations in memory and are connected by a reference or pointer. This gives you more control over how memory is used and how big the data structure is during gameplay than static arrays do. It's important to keep in mind that link lists aren't just an idea that is studied in computer science classes; they are also used in real life to make some applications work and even in more complex data structures. Linked lists can be used to create any kind of structure, from simple ones like stacks and queues to more complicated ones like graphs and hash tables. Modern programming languages may have built-in collections and libraries that hide the details of how linked lists work. However, it is still important for programmers to understand how these work in order to write code that runs quickly and choose the right data structure for the job. We will talk about everything you need to know about linked lists in this article, from what they are to their different types, how they are implemented, how well they work, and how to use them in real life. We will look at singly linked lists because they have

easy logic, doubly linked lists because they can know both ways, and circular linked lists because they are unique. But we still have a long way to go. As an example of a conventional approach, The Basic Idea Behind Linked Lists By definition, a linked& list is linear data structure made up of series of nodes, or elements, that are not saved in memory locations that are next to each other. Each node in chain has a data field and a link to the node that comes next in the chain. Arrays and linked lists are different in this basic way, which makes them easier to use and apply but also more difficult. How a Node Works Any linked list is made up of basic building blocks known as nodes. Most nodes are made up of two parts:

- **Information:** In most situations, this can be either simple type of value, like int, or a well-structured object type of value.
- **Reference (or Link):** These points to the next node in list, creating the "link" that goes to the data structure's name.

A simple way to describe a node in a computer language like C is shown below: struct Node {~ int data; struct Node* next}; Chains of nodes can easily be made to grow and shrink while the program is running with such a simple structure. In a Linked List, the next (or link) component is what lets you move from one node to the next. It is at the heart of all processes that are done on linked lists. Dogs with Heads & Tails Most versions of linked lists keep special pointers to make operations easier:

• **Head:** first node in list, which is used as the starting place for traversal.

• **Tail:** This is reference to last node in the list, which can make actions like adding new items to list much faster.

The node doesn't have any other links going in. It is important to remember that tail pointer is not required, but it makes some actions faster (like adding elements to the end of The head pointer is all that matters—if we lose track of the head pointer, we lose the whole list, because in a simple linked list, Null Termination is used to get to nodes that don't exist. is like a watchtower that lets you know the series is over and will end. When you're traversing a linked list, this null reference means you've hit the end of the list and don't need to try again. The list is shown by a null reference in the "next" pointer of the last node. This value is null The end of Dynamic Memory Allocation feature in other standard linked list types lets linked lists grow or

shrink at runtime without having to follow size limits or perform performance-heavy reallocations. Since items in a linked list don't have to be directly next to each other in memory, linked lists can use memory locations even if the items are spread out in memory. The most interesting thing about linked lists is that memory can be allocated on the fly. Most of the time, they work better than groups. Setting aside the right amount of memory for each part and rearranging the structure as you add or remove parts, which is something that static data structures can't do. or how well memory works is the most important thing. When linked lists are used, they only Many situations call for linked lists, especially when it's hard to say what the largest store size can be. Linked List & Its Operations structure of a singly to understand and use, while taking advantage of the basic benefits that these kinds of data structures can offer. building. In the end, they are easy because they are simple. Singly linked lists are the most basic type of linked list. In these lists, each node only points to the next node in the list, making a one-way path through the list. The "next" property of the Structure and Implementation reference is set to null to show that the list is over. the first node. The last node in the chain only has a reference to the next node and no data. A head pointer to the list is usually used to get to it. In Python, a single linked list is one where each list node: Here is a simple example of how to use the single linked class Node:

elf, data; def init(self):

data self.data = data self. class next = None SinglyLinkedList: def start(self):

self. Everything that happens to a single linked list is based on the head = None simple data& structure. This is true whether you are traversing the nodes, adding nodes to the structure, or removing nodes from it. single linked list is made up of nodes that hold data and a pointer—this Accessing individual nodes based on where they are or what they contain is called traversal operation. Each time, move on to the next node until you hit the null reference, which means the list is over. This action is very important for many other operations on linked lists because it provides the way for You can visit a singly linked list by beginning at the top and working your way down. Here's an example of a popular traversal algorithm:

This is fake code for the function traverse (head):

Current = head as long as current is not zero:

Process the current. Data current = right now. The next number is the number of nodes in the list. You can only visit each node once. Complexity of traversal time: It takes O(n) time to go through both the linked list of words and the list of permutations.

1. The time complexity of process is O(1). to the new node with the link. But this Head Insertion doesn't need to go through the container, so Make a new node and move the head so that its next pointer points to it.

2. instead of tail pointer, which takes O(1) time because the new node can be connected directly to the tail before the tail is changed. node to the new one. If you keep an Append (at the end): Without a tail pointer, this action takes O(n) time, where n is the number of nodes in the list. This is because it has to find the end of the list and set the next pointer of the last node.

3. take O(1) time in the best case and O(n) time in the worst case, depending on how far away the target node is. by going through the list until you get to the node you want to change, and then moving the new node next to it. How long it takes can It is possible to add something after a certain node.

Delete operations are also done in different places, like entry operations: It can take O(1) time to get rid of things from lists that are only linked to one other list. Get the first and second nodes free. How long this takes Take away from the start: It takes O(n) times as long to go through the whole structure as many times as there are nodes before the tail to change the head pointer to point to that node. We have to do this until we get to the second-to-last node, which we need to set to NULL. How to Get to the End Delete: We need to do a worst-case time complexity of O(n) as we go through the data. node. Who is in it: Anderson A certain part was taken away. We see that both the node and the one that came before it have been taken away. We then change the next node to skip revenge for the predecessor node.

Search Operation should be looked at. Searching in a singly linked list is like going through a list from the beginning to the end or to the value we want to find. Like many actions on linked lists, this one takes O(n) time in worst case because every node could To make a

simple search program work, you could: One example of a target: def search(head, current):
Current = head if current is not null
if up to date. When data equals goal,
return current = current. Next // not found; return null
Lists What Are the Pros and Cons of Singly Linked? A list with only one link • Dynamic size change without reallocation
Adding and removing items quickly at the start (O(1))

- Simple implementation compared to more complex list types

However, they also come with limitations:

- Linear time complexity (O(n)) for accessing arbitrary elements
- No backward traversal capability
- Deletion from the end requires finding the second-last node, which is inefficient

A doubly linked list is an advanced variant of a linked list that enhances performance by allowing bidirectional access to nodes. Unlike a singly linked list, where each node contains only a reference to the next node, a doubly linked list maintains two references—one pointing to the next node and another pointing to the previous node. This structure enables efficient traversal in both forward and backward directions, making operations like insertion, deletion, and searching more flexible. However, the improved functionality comes at the cost of increased memory usage and greater implementation complexity, as each node requires additional storage for the backward reference. The structure of a node in a doubly linked list consists of three parts:
1. Data: thing or value that node stores
2. Next pointer: A link to the next item in the list
3. Previous pointer: A link to the node in the series that comes before

Figure 15: Doubly Linked List
[Source: https://www.programiz.com/]

Here is a possible implementation,

```
class Node {
int data;
Node next;
Node prev;
public Node(int data) {
this.data = data;
this.next = null;
this.prev = null;
}
}
class DoublyLinkedList {
Node head;
Node tail;
public DoublyLinkedList() {
this.head = null;
this.tail = null;
}
}
```

Linked list is transformed into a bidirectional linked list, allowing traversal both forward & backward. In practice, implementations of doubly linked list include both head and tail pointers with the addition of the previous pointer, the unidirectional

Bidirectional Traversal

Backward traversal starts in the tail and moves to the front with lists? Forward traversal é similar aosencadeamentos simples, percorreosponteiros next So what differentiates doubly linked lists from other types of linked Forward traversal pseudocode:

```
forwardTraversal(head):  function
current = head
while current is not null:
process current.data
current = current. next
for traversing backwards:  Pseudocode
*/  for tail, traverseBackward, * /...
current = tail
while current is not null:
process current.data
```

current = current. prev

For applications that require reverse access to the data or backtrack to the previously visited nodes. The bidirectional nature of the data structure allows for going back in a more efficient way and makes it incredibly useful Faster Insertion and deletion operations are

1. easier & more efficient than their singly linked equivalents: The bidirectional accessibility of doubly linked lists makes a few operations
2. Insertion operations (assuming you know the target position). pointers need to be updated to create the new bidirectional links when inserting a node between two nodes. This does require more pointer massaging, but it doesn't increase the time complexity, which is still O(1) for Insertion: The next and previous
3. is especially beneficial when deleting from tail end of the list, where the tail pointer allows a constant time operation. predecessor, so deletion of the node becomes an O(1) operation at any point, given we have the pointer to the node. This Operations for deletion greatly benefit from the bidirectional structure. Access to previous pointer of a node in the linked list helps us avoid traversing the list to find its Deletion:
4. is an advantage that makes it especially suitable for applications in which pointers to individual nodes will be kept. that node in constant time by moving it 7 users up in the chain nodes list element. This Inefficiency of List Operations: If we already have a reference to particular node in linked list, we can insert And Disadvantages of Doubly Linked Lists Benefits Over Singly Linked List.

Advantage of Doubly Linked List

- Bidirectional traversal capability
- Efficient deletion operations at any position (O(1) once the node is located)
- Simplified implementation of certain algorithms that require backward movement
- O(1) operations at both ends of the list when using head and tail pointers

However, these benefits come with trade-offs:

- Increased memory overhead due to the additional previous pointer in each node
- Greater implementation complexity, particularly in pointer manipulation during insertions and deletions
- Slightly more complex maintenance as both sets of pointers must be kept consistent

Whether to implement singly or doubly linked list based on the needs of the application, especially as related to traversing the data and the number of each kind of operation. Practical use of linked lists often involves the consideration of Number of Links Linked lists in a circle: A circle linked list can be made with the Loop node pointing back to first node. This is different from linked list where null marks the ends of the linked structure. By Oneself On the other hand, the last and some cases. This circular arrangement can be used with doubly linked lists, which have some unique features that may make them better for Lists: Building Blocks and Types Circular Linked lists are made up of the following:

Figure 16: Circular Linked List
[Source: https://www.programiz.com/]

There are two main types of Can be reached by following next pointers from any node. In the last node, there is no null. Instead, it goes back to the beginning of the list. In this structure, there is now only one loop that all nodes can be a part of. It has data and a next link. One thing that makes it different is that the next pointer of the circular singly linked list points to the head, and the previous pointer of the head points to the last node. This makes a full linked ring that can be traversed in either direction. three things: data, a next pointer, and an earlier pointer. List: This is both a doubly linked list and a circle linked list put together by making next pointer of List. Each point is linked in circle twice.

Simple implementation of a circular singly linked list in C++:
Here is a

```
class Node {
public:
int data;
Node* next;
Node(int data) {
this->data = data;
this->next = nullptr;
}
};
class CircularLinkedList {
private:
Node* head;
public:
CircularLinkedList() {
head = nullptr;
}
etc.  insert, delete,
};
```

Implementation with regular linked lists is in way pointers are manipulated to maintain the circular effect. The only difference with

Traversal in Circular Lists

Use an alternative stopping criterion, generally checking if they're at (or have returned to) the starting point of the traversal, since there is no null termination. care as it can lead to infinite loops. Here, traversal algorithms have to unlike the regular linked list, traversal in circular linked lists needs extra Traverse algorithm about circular linked list:

A simple Function traverses (head):
If head is null:
return
current = head
do:
process current.data
current = current. next
while current is not head

For traversing forward and backward in the case of circular doubly linked lists. & each of the nodes is visited once, even in the head of the list circular. same algorithms are used It helps to guarantee Circular Lists Special Operations on idea suggests: More for Circular Linked Lists Circular linked lists allow for certain operations that are exactly as efficient or natural as the scheduling or any context where the data centre has to be processed in a cyclic way. Time pointer adjustment. It is helpful especially for those functions of application like round-robin Rotation: It makes operations such as rotating the list (changing which node is the head) a constant suited for implementing applications that have repetitive or cyclical access patterns. The circular structure allows for continuous processing of elements without needing to return to the beginning when reaching the end. Circular lists are thus best Circular Structure: Josephus problem and similar elimination algorithms, where elements are systematically eliminated according to a counting method. Josephus Problem Circular lists allow to solve the

Implementation Considerations

When implementing circular linked lists, there are a few things you need to be careful with:

Pointer of the node points at itself. In the case of a single element list, next List: We need to take care of empty lists or lists with just one element. The head is null in Handling Special Case of Empty

1. Linked list, next pointer of the new node must point to the head, & if it's head that is being removed, the next pointer of the tail must be updated to point to the new head. Deletion: These operations need to keep circular structure. For example, when appending to a circular singly Insertion and
2. Did not return to the previous root that is, there should not be an infinite loop. Termination for Traversal: Similar to the traversal, for some mutations, it must make sure that the traversal

Circular Linked Lists Pros and Use Cases for advantages of circular linked lists:

- Elimination of null checking during traversal (after ensuring the list is non-empty)
- Natural representation of cyclical relationships or processes

- Efficient implementation of round-robin algorithms
- Simplified access to both ends of the list (the last node can be accessed in O(1)
- time even in singly linked circular list by following head->next if a tail pointer is maintained)

Common applications include:

- Process scheduling in operating systems
- Music playlist implementation (playing songs in a loop)
- Implementing circular buffers
- Game implementations where players take turns in a cyclical manner
- Any application involving repetitive access patterns

4.2 Operations on Linked List: Insertion, Deletion, Traversal

That need to be continuously processed in a cyclic this is essentially a design decision between adopting either standard or circular linked lists, but cyclical relationships make more sense when there are parts or relationships that go in a circle. Operations That Cross Linked Lists Adding, removing, and connecting items in a linked list gives it a special structure that makes it useful for certain tasks. Keep in. When compared to arrays, this flexible storage system has both pros and cons. Arrays store their elements in blocks of memory that are next to each other. Linked lists, on the other hand, store their elements all over memory and connect them using references, or "links Linked lists are one of the most basic types of data structures in computer science. They have a set price, which makes them adjustable. This makes them best for data systems that need to add and remove things a lot. Turn the whole building around. In link lists, the next node can't be the size of the data structure, though. Linked lists can grow or shrink while they're going without having to be programmed because this basic structure lets memory be given out on the fly. Each item in a linked list is called a "node." It holds both information and a pointer that you need to learn and understand in order to use them. to make more complicated data structures and methods, like stacks, queues, and some kinds of trees. A programmer should know a lot about linked lists and how to use them in basic ways. In order to work with linked lists, you need these things. These methods can be used by developers to Adding to, removing from, and navigating linked lists to change them That link in the chain has been hit. Most of the time, the

last node goes to NULL. In other words, the very end of the list has a data structure where data elements, or nodes, are not saved next to each other in memory. Each node stores a data field and a link to the next node. You must understand how linked lists work in order to get this. Any list with links is a You must know what the program needs in order to understand how things work. Hey. to the first node instead of NULL). The pros and cons of each are shown below. Each link only leads to the next one. Circle linked lists (where the last node points to itself) and doubly linked lists (where each node points to both the next and old nodes) are more complicated. One more type of linked list is a single linked list. If you want to write a node in code, this is how you do it: This node has data (int data) and a link to the next node. Right now, there is nothing in Node* next; // ^;. Some types also keep the tip of the tail. It points to the last node in the list and maps to the first node. When there are no links in our list, we add NULL here. The linked list starts at the head and ends at the end. This lets us quickly get to the last node without having to build up the whole list, but it takes more space to store the references and we can't just pick out parts like arrays do. Not needed any longer. However Memory can be quickly given to a new node while the program is running and taken away when the node is no longer needed. This means that linked lists can be used in different ways.

How to Get Through Lists Linked next pointers until we reach the end, which is shown by a NULL reference. A linked list lets you find, count, and show its items. It's one thing at time from top of list to the bottom. That same idea is also used in a lot of other things, like easiest thing to do in a linked list is navigation. It only means going through the list one item at a time. One issue with linked lists is that you can't just jump to certain items; you have to go through all the nodes before you reach the one you want. There is a set order to traversal, which shows one of the Each node can be moved through and dealt with in a different way. For example, the data can be printed, compared to a key, or the list's beginning or end can be changed. It is possible to drop the time complexity to O(1) by keeping track of extra pointers. There is no way to avoid this linear time complexity since the loop has to go through each item in the list. When you need to get to certain places, on the other hand, like is O(n), where n is number of linked list nodes. How long it takes for a

simple traversal method for a list with only one link to be: This is the code: Node* head and void travel If current is not NULL, do something while current is not NULL. { (like print data) // Now work on the current node coutdata; } }, which means the first node in the list is taken care of first. It will keep going until it hits the NULL address, which means the end of the list. By following the next links, you can easily walk through the list as you go through the nodes. This is because each node is added as it is moved through and then dealt with. The It's sometimes easier to read code when it starts with a current pointer at head. This is especially true for processes that do a lot of work at each node. In general, traversal uses more memory than repeated traversal, and when lists are very long, they can cause the stack to overflow. This method, on the other hand, uses the call stack to go through each node one at a time.

Master lists that can be interconnected, similar to how you search for a specific part. The idea of traversal is important in data structures with vague meanings, like linked lists. You need to know about it in order to do many things with linked lists, you need to traverse them. These tasks can be easy, like printing parts, or hard, like using Linked List. Adding to a linked list is faster than adding to an array because you only need to change pointers instead of moving a lot of items each time. This is especially true for large data sets. This means that insertions could be Putting in: To add a new node to a linked list is something that is done a lot. The links between the nodes are changed to do this. The main idea is the same. Two different ways to add something to a list are to put it at the beginning (propend), at the end (append), or at the beginning (prepend). You can make all of these changes with the Start (Prepend) Insertions, which let you move through the list. and putting the new node at the very top. This move takes O(1) time since it doesn't need to be at the top of the list. That word is also used for this. Add a new node and move it to the top of the list. It's possible to change the head to point to the new node by setting the node's next to head reference to *head Ref. Please set *head Ref = new Node; lists. The head is at the top of the list, and the old head is next. This way works very quickly and makes it easy to change the head pointer in stacks that use linked lists. This brand-new node is now A double pointer (**head Ref) is used in these changes so that the method can add at the end pointer. This means that it takes

O(1) time. We don't have a tail address yet, so we need to find the last node in the single linked list. It takes O(n) time to do this because we have This is how you add a new node to the end of the number. Empty spot for new Data "Make a new node" is what Node** head Ref means at the end. You can see if the list is empty by adding a new node and setting the next value of its last node to NULL. The new node should be the head. Return if *header is not NULL; if it is, go to the end // Node* last = *head Ref; while (last->next! = NULL) { last = last->next; } node and set next to the new node. Finally, Add the last node to the end (last->next = new Node;} time action that doesn't change). put it next to the new node. (Not going through the list can be skipped if you keep the version of the linked list that has a tail address. After that, this node would be at the top of the list. It goes through the list again if not. The first part of this answer sets the place for when the input linked list is empty, which doesn't happen very often. To add some things close to the start, put them in at a certain reach O(1). Change the links in the list to add the new node. This is how we find the ith place in a linked list. It takes O(n) time at most to add something to the end of a list, but it can be done faster. The void insertAfter(Node* prevNode, int new_data) method adds at the ith Check to see if the node before this one is empty.

if (prevNode is not NULL) {

Let coutdata = newData.

The next node next to the last node

If prevNode->next = newNode, then

The next link from the node before it leads to a new node.

NextNode of prevNode = newNode;

}

In this case, we add a new node after node that was given (prev Node). To begin, it checks to see if the given node is empty. It adds the given data to a new node if it is. Then, the next pointer of the new node is changed to point to first node's next, and next pointer of the first node is changed to point to the new node. new node is now where it should be. An average case is adding an item at a specific index point from the beginning of the chart:

Void insert At Position(Node** head Ref, int position, int new Data)

{

// Add at the beginning if position is 0

If position is 0, do this:

The insertAtBeginning() code is made, which will add the new node to the beginning of the linked list.

come back;

}

// Make a fresh node

The new Node, newNode, is a Node*.

data for newNode = newData;

– Move to the node right before the place

Current in the node is *headRef;

If int i = 0 and inext

}

To see if the position is correct, call

if current is not NULL;

{ "cout" : "Position out of range" }

come back;

}

// Add the new node

This is current654->next = newNode;

this->next = newNode;

}

This version takes care of inserting at the given position if that spot is indexed at 0. This first checks to see if the value is 0, and if it is, it sends the call to insert At Beginning. If not, it goes through the list until it finds the node before the requested position and adds the new node after it. It gives an error if the point is out of range. Taking out items from linked lists is also a basic action that lets you take out items from different places in the list. In addition, deletion only involves changing links and not moving any elements, so it could be very fast if the data set is big. This is different from deletion in an array. In a linked list, there are three common deletion scenarios, That is, get rid of the first node, the last node, and a node by its position or value. All these three cases require different pointer manipulations, but they follow the same logic of changing the links to completely skip the node to be deleted.

Deletion of the First Node

Removing from the beginning (It's easy to get rid of the first node in a linked list. This way moves the head pointer to the second node

instead of the first one, but it doesn't delete anything from the list. This action takes O(1) time because it doesn't involve going through the list. To delete the first node, we can do the following.

```
// Check if the list is empty
if (*headRef == NULL) {
return; // Nothing to delete
}
// Store the current head
Node* temp = *headRef;
Update head to next node
*headRef = temp->next;
⋯ //Free memory of the deleted node.
delete temp;
}
```

The first step the function takes in this implementation is to assess whether the list is empty. Otherwise, it saves the current head node in a reference so it has access to the node we are going to get rid of, then it sets the current head to the next node, and deletes the node we removed from next node. This operation has the best efficiency, which is why it is the most common for implementing queues in linked list.

Taking away the last node

In a single linked list, there is no such structure, so getting to the second-to-last node from the beginning to delete the last node is more difficult. This action will take O(n) time because it needs to go through the queue.

```
throw new void(Node* * head) {
// See if the list is empty
If *headRef is not NULL, do this:
return; // Nothing to get rid of
}
// If there is just one node
If (*headRef)->next is not NULL,
Get rid of *headRef;
*headRef = '';
come back;
}
// Go to the next node and find the one where the next node is not null.
```

Current in the node is *headRef;
while (next! = NULL for current, next, next) {
right now = right now->next;
}
// Get rid of the last node
Get rid of current->next;
this->next = null;
}

This solution handles two types of edge cases: a list with no items and a list with one item. It goes through the list with more than one node until it finds the last node, deletes the second-to-last node, and sets the second-to-last node's next pointer to NULL. If you use a linked list with two links or a linked list with a tail pointer, this process might run faster. Getting rid of a node at a certain position or value Also, to If we want to delete a node in middle of the link list, we have to go through the list elements until we get to the node we want to get rid of. Then we have to link the next node we want to get rid of to the node that came before it. You have to look through the list and find a node with that number in order to delete it. In the worst case, both tasks take O(n) time to finish.

void deleteNodeAtPosition(Node** headRef, int position) deletes a node at a certain point.

// See if the list is empty

If *headRef is not NULL, do this:

return; // Nothing to get rid of

}

// Keep track of the head

Node*temp = *headRef;

The head will be moved away from the data point.

If position is 0, do this:

*headRef = temp->next;

Get rid of temp;

come back;

}

for (int i = 0; temp! = NULL &&i next;)

}

// check to see if position is greater than the number of nodes

if (temp == NULL || temp->next == NULL)

return; // Don't do anything, that's not allowed.

}

node to get rid of? >

Node* nodeToDelete = temp->next;

Take the node off the list and connect the node before it to the node after it that was removed.

temp->next = nodeToDelete->next;

Free up the memory of the node that was removed.

Delete nodeToDelete;

}

First, this version checks to see if the list is empty or if head (position

0) needs to be taken off. When the position is not the same as the head, the pointer/reference pointer makes a new pointer that points to the head and moves forward until it hits node before the node that needs to be deleted. It then frees up the memory.

It's kind of the same for getting rid of a node with a certain value:

Int value, Node** headRef; void deleteNodeWithValue(headRef, value);

// See if the list is empty

If *headRef is not NULL, do this:

return; // Nothing to get rid of

}
// Keep track of the head

Node*temp = *headRef;

Node* prev = null;
The first thing we should do is see if the value we want to delete is in the head node.
if (temp! is null and temp->data is equal to value) {

*headRef = temp->next;
Get rid of temp;
come back;

}

// Get rid of the value at the node that has it.
If temp! is NULL and temp->data! is value, do this:
prev = temp;
temp = temp->next;

}

// If the value couldn't be found
If temp is not NULL, do this:
come back;
}

remove(): Take the node out of the list.
prev->next = temp->next;
// Free up the memory that the removed node used
Get rid of temp;
}

This implementation checks to see if the value that needs to be removed is in the head node. If it doesn't find it, it goes through the list again and again until it finds the node with the number that was given, while keeping track of the previous node. If we find that node, we'll change the pointers to skip it and then free up its memory.

Advanced Uses of Linked Lists

Linked lists have many useful features besides the basic processes of adding, removing, and iterating. These features make them appropriate for many real-life situations. Not only do these processes search and reverse, but they also merge & find cycles, to name a few. Understanding these more advanced functions is important if you want to use linked lists in complex algorithms and data structures to their fullest.

4.3 Sorting Techniques: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort

Bubble Sort

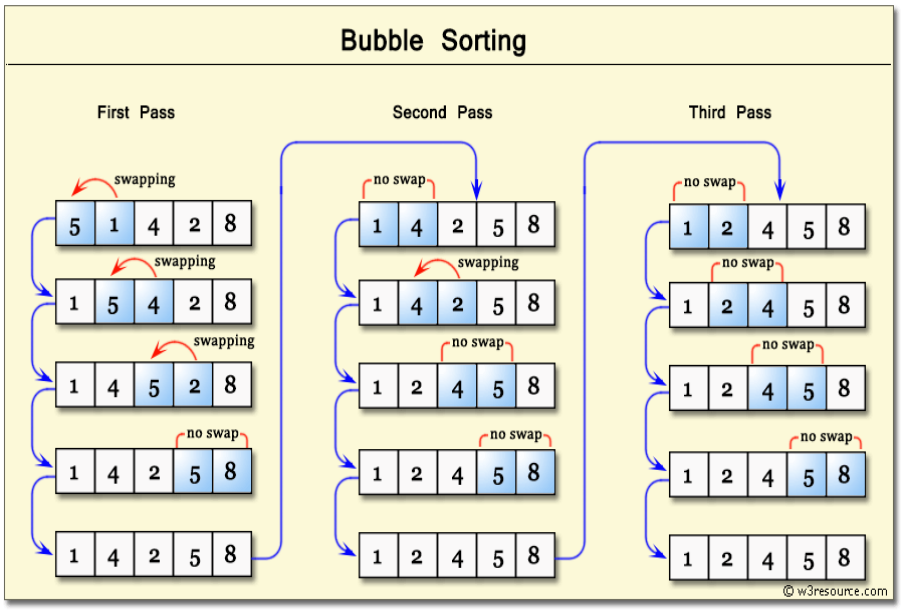

Figure 17: Bubble Sort
[Source: https://miro.medium.com/]

Some sorting algorithms are more complicated than others. Bubble sort is the simplest because it just compares neighboring items in the list and changes the order of them if they're not in the right place. This goes on until there are no more swaps to do and the list is in order. Random access isn't possible with queue-based bubble sort, so it's not good for linked lists either. This is because the data are saved in running memory. The only way to swap two nodes in linked list is to swap the data as we go through it. This is easier than moving pointers. Once this method groups the linked list, it doesn't change the way it looks. To avoid this, the bubble sort should not be used for bigger files because it might take too long. Bubble Sort is great because it is simple and easy to use. It's stable and doesn't take up a lot of extra space. In other words, items with the same value appear in the same order in both the unsorted array and the sorted result. We saw that it takes quadratic time, which means that a very long linked list can't be sorted with it. GET OUT this is a junk-sort sort algorithm called

bubble sort. It's slow, but it's a good way to learn about them or to handle very small linked lists, where even simple sort algorithms would take too much work. The linked list needs to be sorted by bubble sort more than once. Each time through the tree, the values of nodes that are next to each other are checked to see if they are in the right order. If they aren't, they are switched. This is done again and again until the linked list is fully traversed without any swaps. The linked list is now in the right order, as shown. Our method is different from array-based bubble sort because we have to start at the beginning of the linked list every time. We can't just go to any entry. Bubble sort isn't as easy to use when you only have one list that is connected to another list. We can only go forward in a list with one link. It's easy to compare this node to the next one, but we have to go back to the beginning of the list to compare it to the ones that came before it. Because of this, bubble sort takes even longer, and single linked groups are now the worst in terms of time complexity when compared to other data structures. Because we can move around the list in both ways with doubly linked lists, the implementation makes more sense. This inverted navigation feature makes it easier to compare and swap, but it still takes $O(n^2)$ time total. You can still use bubble sort on a doubly linked list, but it's not designed to work on big doubly linked lists, even if you use the above slight edge. One way to make things better is to use Bubble sort on a linked list so that items come back faster. To do this, a flag is kept to show if any changes were done in a single pass through the list. But if there were no changes, the list would already be in order, so the algorithm could end early. If the list is already somewhat sorted, this optimization can make things run much faster without changing how long it takes in the worst case. A bidirectional bubble sort, which is also known as a cocktail sort or shaker sort, is one example. It goes through the list twice, once going forward and once going backward. On the forward pass, it moves the biggest items to the top, and on the backward pass, it moves the smallest items to the bottom. While this method can cut down on the number of times list has to be sorted in some situations, it still has a worst-case time complexity of $O(n^2)$.

Using bubble sort on linked lists will be a great way to learn about how sorting algorithms work and how to use linked lists, even though it is quite complicated. This is because it shows how hard it is to

change linked data structures and teaches basic ideas that can be used to understand more complicated sorting methods. The method is easy to understand and follow because it is written in steps. This makes it a useful teaching tool.

In short, bubble sort works on a linked list by switching nodes that are next to each other if they are not in the right order. Even though it takes $O(n^2)$ time and doesn't work for bigger sets of data, its ease of use and helpful descriptions make it worth it.

Make it worth your time to learn. To sort big linked lists, on the other hand, other sorting algorithms work better and should be used.

Pick Choose Sort

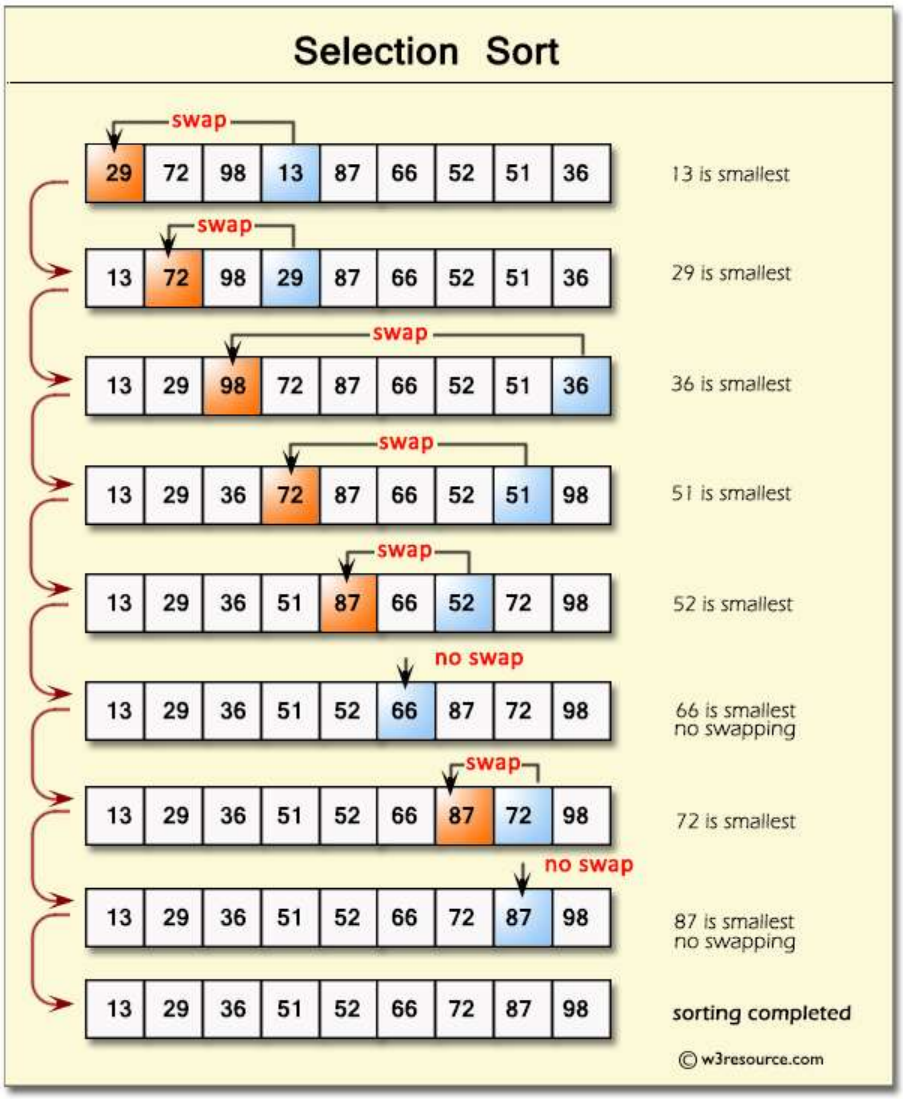

Figure 18: Selection Sort
[Source: https://www.w3resource.com/]

A simple way to sort that can be used with linked lists is selection sort. It takes the list you give it and splits it into three sublists: two that are organized and one that is not. When you sort a sublist, everything is at the beginning, but when you unsort it, it's empty. In other words, the method moves the item at the end of the sorted sublist that is the smallest (or biggest) in the unsorted sublist. It keeps doing this until the sublist that isn't in order is empty and the list is in order. As long as we change how we do it, we can use selection sort on a linked list. Linked lists are already in order. There are nodes in a connected list. You have to start at the top (or present) node and work your way down to get to a certain node. This is helpful because it changes how the swapping process works and how the smallest part is found. in sort by choosing. Most of the time, selection sort takes O(n2) time with a single linked list because it takes O(n) time to find the smallest selection in the part of the list that isn't sorted. As soon as we find the smallest item on the list, we trade its value for the first item that is not in any one order. We won't have to do an expensive O(n) time list rebuilding if we do this. When you swap, the line between the sorted and unsorted areas moves one node to the right. The time it takes to pick sort on a linked list is always $O(n^2)$. It doesn't matter what order the things are put in. It takes O(n) time to find the smallest item in the unsorted part of the list each time. Sort by choice: This type of time complexity is quadratic, not bubble sort: This type of time complexity is quadratic, just like bubble sort. However, it is better than bubble sort because the number of changes costs O(n), while in bubble sort they cost $O(n^2)$. This approach works best with linked lists and cases where it costs more to swap parts than to compare them. Choice sort might be faster than bubble sort when there are a lot of changes and a lot of data elements. This is because it doesn't change much. In addition to the size of the list it is given, selection sort also uses a set amount of memory. This is called a "in-place" algorithm. When you use selection sort on a linked list, it can go off track sometimes. This means that parts that are the same may appear in a different order than what was typed in. On top of that, it can be hard to keep the relative order of parts that are the same. This sorting method doesn't use lists that are already in order, and it always compares the same number of items, so it doesn't matter how many items are in order.

We don't flip values, but instead use a pointer-based method that works best with linked lists. It takes a long time to value-swap between linked list nodes when we need to swap large amounts of data, like big structs. We can restructure the linked list by moving the links around. This way is harder, though, and you have to be very careful with how the lists are organized, especially if they are only tied to each other. When we implement selection sort, it can be faster if we make doubly linked lists because we can move through them both forward and backward. The bidirectional feature might help us find the minimum element more quickly, but it doesn't change the fact that it takes $O(n^2)$ time. Even though that might not be necessary to build the list, it can make it easy to move the pointers around while sorting their values. There is a simple comparison-based sorting algorithm called selection sort. It may not be as fast as other sorting algorithms because it takes O(n^2) time, but it can still be useful in some situations. For example, when working with linked lists, selection sort does the fewest number of swaps compared to bubble sort and other algorithms. Selection sort works the same way no matter what data is put in, which can be helpful in some situations. However, the algorithm's quadratic time complexity means it can't be used with big data sets. Sorting items by selection on linked lists is also a great way to learn how to make algorithms and change data structures, especially in a classroom setting. It shows how hard sequential access data structures can be and how different ways of sorting can be both expensive and useful. This change to the selection sort algorithm will help students understand it better and how it can be used with linked lists. For short, selection sort is a simple method for sorting linked lists that takes $O(n^2)$ time to run. To be fair, this isn't the fastest way to sort large databases, but it's easy to use, doesn't take up any extra space, and only swaps during sorting, which makes it a good choice in some situations, like when memory usage is an issue or when swap transactions are too expensive.

Sort by Insertion

Method: Insertion Sort: Insertion sort is a sorting method that & each item to the sorted collection one at a time. It saves more time than bubble sort and selection sort on small data sets, for example, and it adapts to the input by being faster when it is partially sorted. When working with linked lists, insertion sort uses the fact that the parts are

linked to make it easier to put them where they belong in the sorted list.

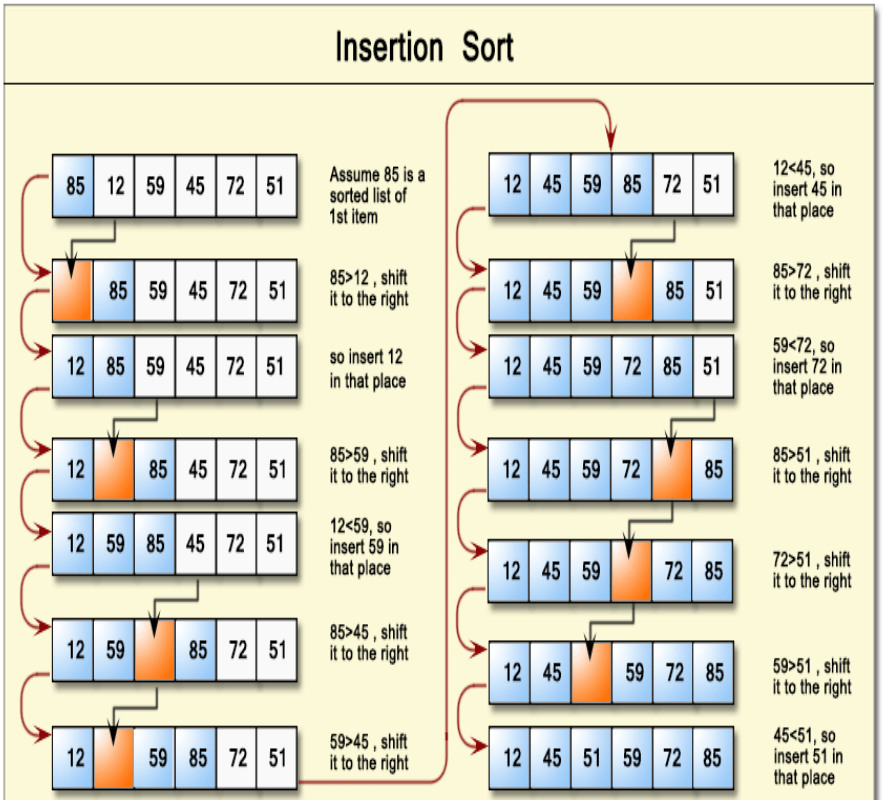

Figure 19: Insertion Sort
[Source: https://www.w3resource.com/]

Insertion Sort Idea: The main idea behind insertion sort is to split the list into two parts, one that is sorted and one that is not. 1) At this point, only the first element has been sorted. The rest of the elements are still not sorted. It picks each item from the list that isn't in any particular order and puts it in the list that is. This is done again and again until all the items on the list have been handled and the list is in order.

In the singly linked list form of insertion sort, every item in the first sorted list will be an integer. This means that the unsorted list will hold these as well while this part is being built. We can begin the unsorted list with all the numbers from the original list as given. Things keep getting moved around in the sorted list as we remove them from the unsorted list. With this method, we don't have to move the things around, which is something we can't do with a single linked list because we'd have to start over. Adding items to a linked list takes $O(n^2)$ time in the worst case. This is what happens when the list of

data is put in the wrong order. In the best case, when the input is already almost sorted, it only takes O(n2) time, but in the worst case, it takes O(n) time. It works better for partially sorted lists. Because it can handle different situations, insertion sort is a good choice when the data is almost certainly sorted. One great thing about insertion sort is that it is stable. This means that in the result, elements that are the same stay where they are. When the order of parts that are the same is important, this trait comes in handy. It's also a live program, which means it can sort a list as new items are added. It works better with streaming info because of this. In insertion sort, working with linked lists takes up about O(1) extra memory. This is true no matter what size the input is. It's an in-place method because of this. When memory is weak, this function comes in handy. limited or where the data set is very big and might be too big to copy in memory. Insertion sort on a list with two links, so you can go through it both ways. This makes adding a new node a lot easier because we don't have to go through the list from the beginning and can go straight to the node before it. This could be an advantage that makes it better than insertion sort on lists with only one link. Because it is flexible, insertion sort can work better for lists with few items or that are only partly sorted than bubble sort and selection sort. In the worst case, though, more complex algorithms like merge sort work much better with long lists that are sorted randomly. Insertion sort is still useful in some situations, especially when dealing with small datasets or partially sorted arrays, where it can work as well as more advanced sorting algorithms. Binary insertion sort is a cool type of insertion sort that can be used when the input is a linked list. It uses a binary search to find the right place to enter the item. One of these optimizations doesn't work at all with linked lists because they don't allow random entry. There are, however, ways to make insertion sort on linked lists better, based on the situation and requirements. For example, you could add sentinel nodes or keep extra pointers. It is true that insertion sort on linked lists is often used as part of more complicated sorting algorithms, but. It can be used as the base case in recursive divide-and-conquer sorting methods like merge sort and quick sort when the sublist's size drops below a certain level, for example. In this way, the hybrid method takes both into account by using insertion sort for small lists and more powerful algorithms for

asymptotic performance on larger lists. There are many sorting algorithms out there, but insertion sort is one of the best for working with a linked list data structure, especially if it's small or only partly sorted. For instance, it can be used in a lot of different situations because it is flexible, the sorting methods are stable, and it can sort things in place. In the worst case, it takes $O(n^2)$ time, but in reality, it works better than most quadratic sorting algorithms, especially for linked lists where adding items costs less than in arrays.

Sort Quickly

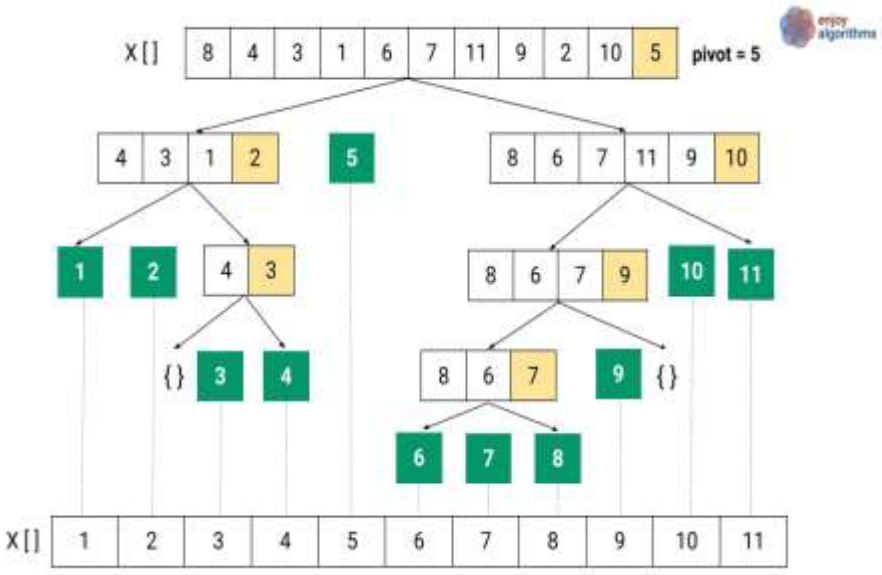

Figure 20: Insertion Sort
[Source: https://miro.medium.com/]

One good way to do quick sort is to use divide-and-conquer sorting method. This kind of sorting can be faster than simpler ones like bubble sort, selection sort, and insertion sort. Since linked lists are sequential-access data structures, it can be hard to use quick sort on them, but it also presents a chance. Unlike arrays, linked lists don't let you access them at random, so we need to change the way the quick sort method works. Quick sort uses a method called "partitioning," which involves choosing an element as "pivot" and dividing the other elements in the array into groups based on whether they are less than or greater than the pivot. This process is repeated for each of the new groups that are created. It goes back to the two most recent sub-arrays it made. There is an average time complexity of O(n log n) for this method. It is one of the fastest sort algorithms. Quick sort needs to

split, which means moving items around the pivot7, but this is harder to do on linked lists. When you move elements around the center in an array, it's easy to switch them around. But when you want to move things around in a linked list, you need to be careful not to change the pointers. To do this, you can make three different lists: one with elements less than the pivot, one with elements equal to the pivot, and one with elements greater than the pivot. The three lists are then joined together after the lesser than and greater than lists are sorted over and over again. When we have a list with only one link, we usually choose the first or last node as the center node. After this shift, the list is split into two parts, and recursive calls are made on both of the resulting subarrays. In this last step, these ordered sublists are just linked together to make the full ordered list. To do this, you need to pay close attention to edge cases like lists with only one item or lists that are empty. Quick Sort on linked lists takes O(n log n) time on average, which is a lot less time than easier algorithms that take $O(n^2)$ time. Still, the worst-case complexity is still $O(n^2)$ if the pivot pick makes each stage's partitions not balanced. You can choose as the first or last item on the list if it is already sorted or almost sorted.

Choosing the right turning points can lower the risk of bad performance in the worst case. Some of these are picking a random element as the center, finding the middle point between the first, middle, and last elements, or using more complex sampling methods. These methods can be used to make balanced partitions, which is necessary to keep the program logarithmic so that the data is organized well. Quick sort in linked groups also saves a lot of space, which is a big plus. The recursive calls take up O(h) stack space, where h is the height of the recursive tree. In the best case, the height is O(log n), but in the worst case, it can reach O(n). Sorting can be done while the lists are being changed, so no new lists need to be made. This skips more steps than making new lists. terms of memory in such cases. This feature makes quick sort a more efficient algorithm in sorting algorithms like bubble sort, selection sort and insertion sort. In fact, for large ordered lists its performance of quick sort is better than simpler elements better performance since it has a worst-case time complexity of O(n log n). But merge sort usually guarantees a stable sort, which means it can rearrange equal elements. Quick sort on the other hand is not any order. One key drawback of

quick sort is that it is not stable: when simplifying, equal value elements may be output in original ordering among equal elements is necessary. This may be problematic in cases where maintaining the strategy for pivot selection. In addition, the efficiency of quick sort can greatly degrade for particular input patterns, especially when insufficient implementation of a hybrid approach where insertion sort is used on small sublists. A very neat optimization of quick sort applied to linked lists is the the algorithm's performance. Since insertion sort has lower overhead and performs well on small lists, this hybrid strategy can reduce the constant factors in more quick sort divided integers Generally, when the number of elements in a sublist approaches a certain number (sometimes around 10-20 elements) insertion sort is used rather than recursion and. for doubly linked lists because of this double pointer (traversing in both directions). Quick sort has an improved implementation ability makes partitioning much easier and can allow for better pointer manipulations. This bidirectional it are essentially the same as for singly linked lists. Nevertheless, the fundamental algorithm and the complexity characteristics of (for example, in the eighties or nineties quick sort for linked lists was used often) in cases, when the average and worst cases are acceptable. Quick sort on lists is usually used in real applications sets. Since the input is assumed to come random, it works best for large data performance is paramount, or stability are needed alternative algorithms such as merge sort is more suitable. In applications where constant to linked lists. In brief, quick sort is a highly powerful algorithm and can be efficiently applied average time complexity of O(n log n) makes it more efficient than naive quadratic algorithms for large data sets.

Unit 11: Searching Algorithm

4.4 Searching Techniques: Linear Search, Binary Search

Linear search is also called sequential search. It is the simplest search algorithm, which can be implemented over a linked list, and this method, we start from the head and go through the list node by node until we find the element or reach the end of the list. In on a linked list. In a linked list, where elements are not stored in contiguous memory locations, this approach becomes quite intuitive, as linked lists have to be traversed sequentially, making linear search a primitive operation search on linked list is quite straightforward. The implementation of linear value in each of the node with the target value. Let us start from the head node and compare found, the search exits successfully, and returns the node or its location. If a match is list and there is no match, it terminates unsuccessfully; that is, the target element is not found. If the search process goes through the whole linked lists, meaning each call to get() has O(n) complexity. As a result, it has a worst-case time complexity of O(n) due to the sequential nature of that the target element is nonexistent or is the last node in the linked list, it needs to traverse through the entire list. In the case In the best case

MCQs:

1. Which of the following is a key advantage of a linked list over an array?
   a) Faster access time
   b) Dynamic memory allocation
   c) More memory efficient
   d) Requires less programming effort
2. Which type of linked list allows traversal in both directions?
   a) Single Linked List
   b) Circular Linked List
   c) Doubly Linked List
   d) None of the above
3. Which operation is the most time-consuming in a singly linked list?
   a) Insertion at the head
   b) Deletion at the head

c) Searching for an element
d) Insertion at the tail

4. How does a circular linked list differ from a singly linked list?
   a) It has no nodes
   b) The last node points back to the first node
   c) It uses an array for storage
   d) It does not allow insertion
5. Which sorting algorithm has the worst-case time complexity of $O(n^2)$?
   a) Quick Sort
   b) Merge Sort
   c) Bubble Sort
   d) Heap Sort
6. What is the time complexity of binary search in a sorted array?
   a) (n)
   b) O(log n)
   c) $O(n^2)$
   d) O(1)
7. Which of the following is NOT a searching technique?
   a) Binary Search
   b) Linear Search
   c) Bubble Sort
   d) Hashing
8. What is the best case time complexity of Quick Sort?
   a) (n)
   b) $O(n^2)$
   c) O(n log n)
   d) O(1)
9. Which data structure is best suited for implementing dynamic memory allocation?
   a) Array
   b) Stack
   c) Linked List
   d) Queue
10. Which of the following sorting techniques works best with small data sets?
    a) Merge Sort
    b) Quick Sort

c) Bubble Sort
d) Heap Sort

Short Questions:

1. What is a linked list, and how does it differ from an array?
2. Explain the difference between a singly linked list and a doubly linked list.
3. What are the advantages of using a circular linked list?
4. Describe the steps to insert a node at the beginning of a linked list.
5. How do you delete a node from a linked list?
6. Compare Bubble Sort and Quick Sort in terms of time complexity.
7. What is the difference between Linear Search and Binary Search?
8. Write a simple C++ program to traverse a linked list.
9. How does the merge operation work in sorting algorithms?
10. What is the importance of sorting algorithms in data structures?

Long Questions:

1. Explain the different types of linked lists with examples.
2. Discuss the operations (insertion, deletion, traversal) on a singly linked list with algorithms.
3. Write a C++ program to implement a doubly linked list with insertion and deletion.
4. How does a circular linked list work, and where is it used?
5. Compare and contrast Bubble Sort, Selection Sort, and Quick Sort with examples.
6. Explain Linear Search and Binary Search algorithms with step-by-step examples.
7. Describe the advantages and disadvantages of linked lists compared to arrays.
8. Write a C++ program to perform searching operations on an array.
9. Discuss the importance of sorting in real-world applications.
10. Explain how linked lists can be used to implement stacks and queues.

MODULE 5
TREE AND GRAPH

LEARNING OUTCOMES

- Understand the concept of trees and graphs in data structures.
- Learn about different types of binary trees and their properties.
- Explore graph representations and their real-world applications.
- Understand graph traversal techniques such as Breadth-First Search (BFS) and Depth-First Search (DFS).

Unit 12: Introduction to Tree and Graph

5.1 Introduction to Tree and Graph

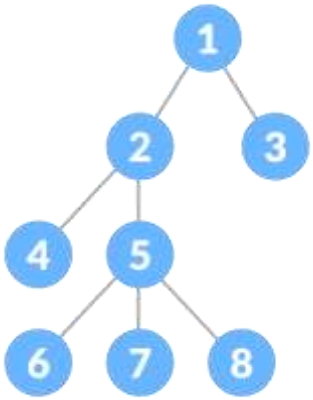

Figure 21: A Tree
[Source: https://www.programiz.com/]

A huge range of algorithms and programs, from databases and file systems to social networks and sites that make suggestions. They back up It is very important to know how to use trees and graphs in computer science to show how things are related to each other. Manipulation of data faster enough to solve complex problems. While graphs allow a wider representation of relationships& through interconnected nodes and edges. This skills help storing, retrieval and Trees allow to represent hierarchical arrangements since each of its elements (also called nodes) has a defined parent-child relationship among them, we look at how to implement, traverse, and optimize these structures for different situations, giving you a solid foundation for solving computational challenges. complete introduction, we will be covering tree and graphs- the concepts, variations, applications and algorithms. Join us as So, in this

Structure Trees a Hierarchical Data

Key properties, which all trees have, are: approach. Some can be represented graphically. And, unlike linear data structures which include arrays and linked lists, trees provide data in a hierarchical Tree: tree is a collection of elements with parent-child relationship that

1. In every tree, this is called root node. There is a specific node present
2. Is the root node which does not have a parent node? only exception
3. Child nodes are a very useful feature for each node. Having zero or more
4. Following the edges. A tree has no cycles — which means that you cannot close a loop by
5. There is only one connected component. We have all nodes are connected through edges so that

Systems, org charts or family trees. obvious parent-child hierarchy. Popular examples are file 1 tree have inherent hierarchical nature which makes them suitable for modelling relationships where there is

Tree Terminology

The vocabulary: To better understand trees, it's helpful to learn

- Node: A part of the tree that holds data and links to its child nodes.
- Root: The point at the very top of a tree that doesn't have a parent.
- Edge: The link between two nodes.
- A parent node has at least one kid node.
- Child: node that is directly linked to another node as you move away from the root.

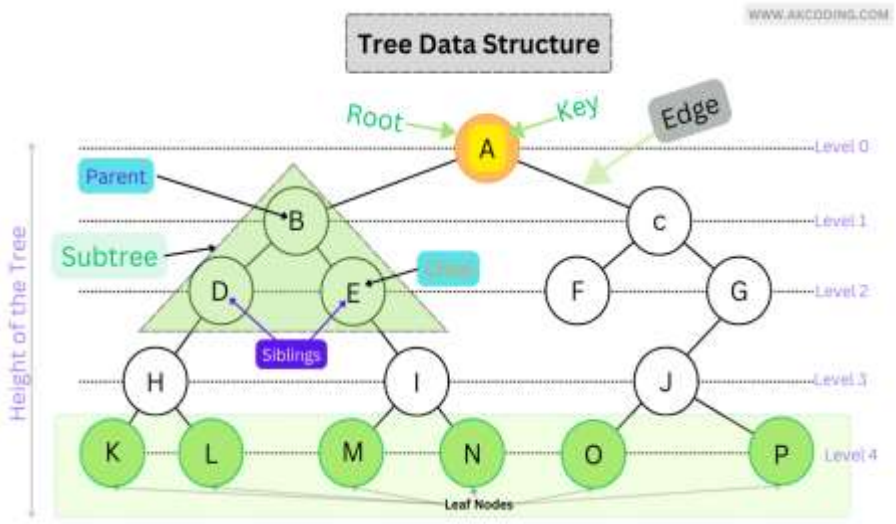

Figure 22: Tree Data Structure
[Source: https://akcoding.com/]

There is only one leaf node, which means it doesn't have any children. Peers are the nodes that have the same parent. Level tells

you how far away the root is (level 0 is the base). The biggest gap between the ground and a leaf. Its depth is the distance along the line from the root to a certain point. In a tree, a sub tree is made up of a node and all the nodes that come after it. Not the Same In Between Trees There is different kinds of trees, and each one is good for different things: There can be only two children at a time in a binary tree. These are called the left child and the right child. A lot of people use binary trees because they are simple to understand and quick to use. The most important thing is that a node can only have two children at most. Kids are sometimes called "left" and "right." If you want to build more complex tree types, the most nodes that can be at level i is 2^i. Two times the height of the tree, plus one, is the most nodes that can be in it. Two-level trees are what many other types of trees are made of. They are used a lot in game trees, term processing, and Binary Search Trees (BST). The root node will be worth more than the child nodes on the left. On the other hand, every node in the right subtree is worth more than the root node. Putting all of the nodes in the right order turns a binary tree into a binary search tree (BST). We can quickly find, add, and remove BSTs because they only take O(log n) time on average. These are the important points: To look through the data, use the binary search feature (left <parent < right). • All set to go • Get output in order with Sorting the database indexing and symbol tables in the computers so they work properly. Binary Search Trees (BSTs) are useful when you need a quick and flexible set of things that can be found, added to, or taken away. Trees That Are Equal related to the logarithm of the number of parts, making sure that the time it takes to do the steps is O(log n). To give some cases, Trees with balance: For the same reason we talked about above, healthy trees don't lose their height. The child trees of each node can be no more than one height apart. In an AVL tree, the two kid trees are the same height, so the tree can balance itself. Third, there are trees: balancing itself As things are added and taken away, binary search trees use colour information to keep the trees roughly balanced. black and red These nodes are great for databases and file systems because they can have more than just left and right. Binary trees can be helpful sometimes. used to keep things on disks. B-tree Soon, B-Trees, which are self-balancing search trees, will be able to get to the root faster. Splay Trees can be found in many language packages, file systems,

and database management systems. They are binary search trees that can change themselves and move parts that have been recently accessed closer together to make things go faster. There should be an even number of each tree type when you use full binary trees in data structures and priority queues. Heap The whole tree of bits: The last level of this full binary tree might be the only one that is empty. From the left to 2^h - 1 nodes, which is the height of the tree, the last level is full. Every node in a tree has two children, and all of the parent nodes are on the same level. There are these trees. What Kinds of Binary Trees There Are Binary Trees That Always Work: Every node in a full binary tree has two children, except for the leaves. There were several reasons why it was made, such as: Different types of trees can be used to store info. 1. It works the other way around for min heaps. is greater than or equal to C. One more thing about the pile is that it. The number of a max heap node C's parent node P It is basically a type of tree-based data format that gives you ability to which makes it great for quick lookups of prefixes, auto complete ideas, and spell checkers. of strings. Trie uses a tree to store strings. Trie: trie data structure is like a tree and is used to store a changeable set. It is also called a prefix tree.

Problems with geometry and range queries. These data structures are widely used in computing and can be used to quickly query info about intervals. Segment trees are: Data structures are ways to store data. Systems. Eight or children, per These are useful for geographic knowledge, computer graphics, and image processing. Quad There are many kinds of trees, including these, that can be used to divide up space. The nodes inside the trees have four sides. That offer fast methods for computing and manipulating prefix sums in a table of values. Binary Indexed Trees, or Fenwick Trees, are type of data structure. Crossings of trees common ways to move through a tree: link in a certain order. One thing is certain: there are a number of To explore a tree is to go through all of its nodes. In depth-first traversals, we go as far down one branch as we can before going back up: During First go to the left sub tree, then root, and finally the right sub tree (Left-Root-Right).

It gives you a list of nodes in increasing order for a BST. Applications: Getting info from a BST that has been sorted. To pre-order traversal, go to the root, then the left sub tree, and finally the

right sub tree (Root-Left-Right). Examples: making a copy of the tree, evaluating a prefix expression. Third, go to the left sub tree, then the right sub tree, and finally the root (Left-Right-Root). For example, you can delete a tree or check a postfix statement. Traversal by Breadth First Follow the level order and go from left to right to visit each spot. A queue data structure is used in this method to keep track of the next nodes to visit.

Usages: Processing levels one by one and finding the shortest way.

How Trees Are Used There are many uses for trees in computer science and other fields as well:

1. File Systems: Files and directories are set up like branches in a tree, with files as leaves and folders as the stems.
2. Expression Evaluation: Expression trees show mathematical formulas, which makes it easy to evaluate and change them.
3. Decision trees: These are used for classification and regression jobs in machine learning.
4. Syntax Trees: Abstract syntax trees are used by compilers to read and understand computer languages.
5. Game trees show all the possible game states and moves. They are used in game theory and AI. Spanning trees are used to keep network traffic from getting stuck in loops.
7. Hierarchical Clustering: Trees can show how data points that are grouped together are related in a hierarchical way.
8. Database Indexing: B-trees and other versions of them are used to make database indexes that work well.
9. DOM for XML and HTML: Document object models show text documents as tree structures.
10. Compression Algorithms: Huffman coding compresses data well by using trees.

Setting up trees: There are several ways to set up trees:

Implementation Based on Arrays

For some types of trees, like groups or full binary trees, arrays are a good way to show them:

• For a node at index i, the left child is at index 2i + 1, right child is at index 2i + 2, and parent is at index (i-1)/2. This is because integer division isn't very good for changing things. It works well with caches

and lets you quickly get to nodes at random, but
Based on nodes A more general way to implement it is with a node object that has pointers: More memory is needed for a class TreeNode that stores links (T data; children; // }). This makes processes more flexible, but it hurts relationships. Networks and graphs

5.2 Graph Introduction and Graph Traversal (Breadth-First Search, Depth-First Search)

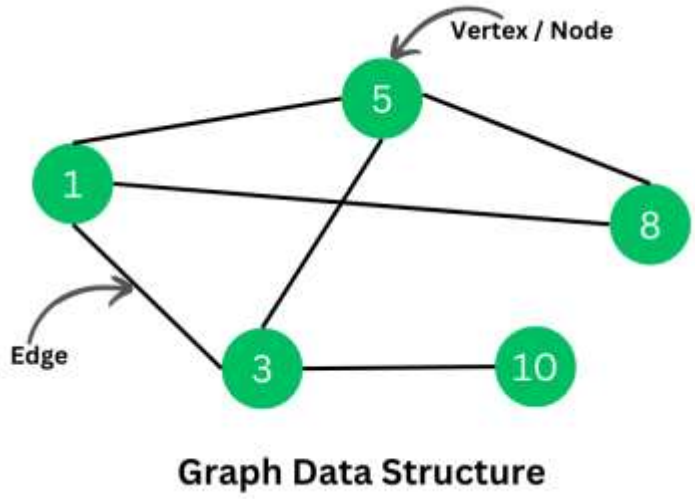

Figure 23:Graph Data Structure
[Source: https://blogger.googleusercontent.com/]

System that is based on levels and can include cycles. Graphs use edges to show how two things are related to each other. Graphs don't have a root like trees do. A graph is a non-linear data structure made up of points (called "vertices") that are linked together. Edges (connections) are important features.

• Nodes (points) and can be directed or not;
• Not trees;
• Can have cycles;

It can be wired or digital, weighted, unweighted, or a mix of the two. Objects are connected to each other. What good do huge, complicated graphs ever do? These are what real-world networks (like computer, transportation, social, and other networks) are built on.

Graph Terminology

concepts: You need to be familiar with the following terminologies to understand graph

building block of a graph, denoting an entity.

• Vertex (Node): basic(relationship)

• Edge: connection between two vertices vertices that share an edge.
• Neighboring Vertices: Two vertices and edges.
• Degree: The count of edges attached to a vertex.
• Path: A sequence of connected point.
• Cycle: A path where the destination is the same as the starting
Between every couple of vertices.
• Connected graph has a path that goes from any two points in the graph to another point in the graph.
• Connected Component: A part of the graph where there are no loops. Tree: A connected group of trees (also called a disconnected set of trees).
• Forest: Something else. Complete Graph: A complete graph has all of its vertices connected to all of its nodes. It can be split into two different sets, with each edge connecting only those two sets. The other type of graph is called bipartite graph. Figure out how to draw a graph in a plane so that no two edges cross each other.

Graph Types:

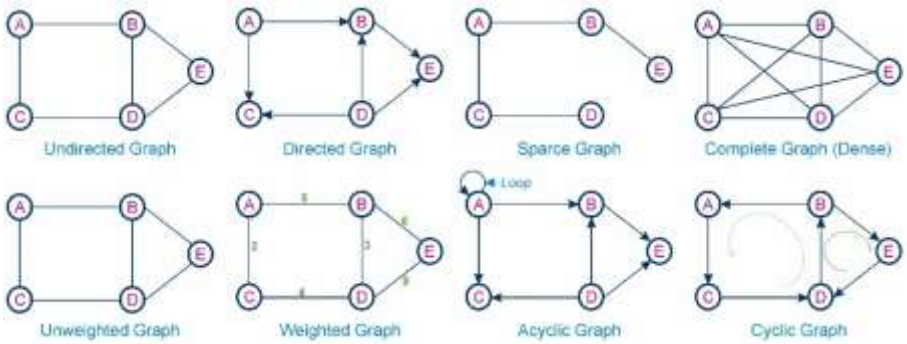

Figure 24:Type of Graph
[Source: https://studyglance.in/]

There are different types of graphs with different use cases and traits. Types so then knowledge Begin to learn from here: The relationships are symmetric if you have Programming. Graphs without a path: The edges don't go in any particular way, so when vertex A is connected to vertex B, vertex B is also connected to vertex A.
• Relationships and social networks are two examples. Graphs with Direction (Digraphs): Asymmetric does not mean that an edge goes from B to A; edges have a direction.
• A line going from A to B Web links, one-way streets, dependencies.

• Examples: Weighted Graphs: Edges have different weights, or costs, associated edge has a value (weight) indicating a cost, distance, capacity, etc. Weighted Graphs: Each

- Examples: Road networks (where weights might be distances or travel times), network flow problems.

Special Graph Types

1. Cyclic vs. Acyclic Graphs:
    - Cyclic graphs contain at least one cycle.
    - Acyclic graphs have no cycles.
    - Directed Acyclic Graphs (DAGs) are particularly important for representing dependencies and scheduling.
2. Dense vs. Sparse Graphs:
    - Dense graphs have many edges (close to the maximum possible).
    - Sparse graphs have relatively few edges.
    - The choice between dense and sparse affects representation and algorithm selection.

3. Bipartite Graphs: These are graphs whose vertices can be split into two separate sets, & each edge links two vertices from a different set. Matching problems, assignment problems, and job schedule are all examples of uses.

4. Planar Graphs: These are graphs that can be made on a flat surface without any edges crossing. It's important for circuit design and drawing maps.

5. Complete Graphs: These are graphs where each vertex is linked to every other vertex. There are n(n-1)/2 edges in full graph with n nodes. Representations of Graphs There are a few different ways to write graphs in code, and each has pros and cons: Matrix of Adjacency In a 2D collection called an adjacency matrix, each matrix[i][j] is an edge from vertex i to vertex j:

- For unweighted graphs: 1 indicates an edge exists, 0 indicates no edge.
- For weighted graphs: The weight value indicates an edge exists, a special value (often infinity or 0) indicates no edge.

Advantages:

- O(1) time to check if there is an edge between two vertices
- Simple to implement and use

- Efficient for dense graphs

Disadvantages:

- $O(V^2)$ space complexity, inefficient for sparse graphs
- $O(V^2)$ time to initialize or traverse all edges

Adjacency List

An adjacency list uses an array or list of linked lists, where each array entry represents a vertex and contains a list of adjacent vertices:

0 -> [1, 2] // Vertex 0 has edges to vertices 1 and 2

1 -> [0, 3] // Vertex 1 has edges to vertices 0 and 3

2 -> [0, 3] // Vertex 2 has edges to vertices 0 and 3

3 -> [1, 2] // Vertex 3 has edges to vertices 1 and 2

Advantages:

- Space-efficient for sparse graphs: O(V + E)
- Faster to traverse all edges: O(V + E)
- More efficient for most graph algorithms

Disadvantages:

- Checking if there is an edge between two specific vertices takes O(degree) time
- Slightly more complex to implement

Edge List

The graph, usually expressed as pairs (or triples for weighted graphs): An edge list is just a list of all the edges in

Unweighted [(0, 1), (0, 2), (1, 3), (2, 3)] #

(0, 2, 3), (1, 3, 1), (2, 3, 8)] // Weighted (s, d, w) [(0, 1, 5),

Advantages:

• Simple to represent be natural for edge-processing algorithms (e.g. Kruskal) • What would

• Space-efficient: O(E)

Disadvantages:

for most traversal algorithms • Not a good fit

Graph Traversals

Meaning we must mark vertices as we traverse to avoid getting in an infinite loop. Systematic visiting of all vertices. Graphs can have cycles as opposed to trees, Traversal of a graph is Breadth-First Search (BFS) as follows: vertices at the current depth prior to moving on to the nodes at & next depth level. The implementation of our queue-based solution is BFS visits all the source vertex and visit it. Begin from a Enqueue the source vertex. the adjacent vertices which

are not already visited into the queue and mark them as visited. vertex. b. Process the vertex. c. Add all While the queue is not empty: 1. Dequeue a

Applications:

- Shortest path in an unweighted graph
- Detecting bipartitions
- Searching for connected components• Web crawling
- Network broadcasting

E is the number of edges. Time Complexity: O(V + E), where V is the number of vertices and Depth-First Search (DFS) or a stack: possible along a branch and backtrack. It can be performed with a recursion Depth First Search: It explores as far as it as visited. Choose a source vertex, mark Process the current vertex. Adjacent vertex. Recursively see DFS for every not visited

Applications:

- Detecting cycles
- Path finding
- Topological sorting
- Connected components of directed graphs
- Solving puzzles like mazes
- Generating spanning trees

E) This algorithm has time complexity based on the number of vertices and edges: O(V + Graph Algorithms problems using dedicated algorithms:

Shortest Path Algorithms

1. shortest path Dijkstra's Algorithm: Find the
2. Extended Binary Tree Complete Binary Tree and
3. Introduction to Binary Trees

Algorithms and systems. two children, usually called the left child & the right child. With this simple constraint you create a powerful structure that has been applied to innumerable computer science. They are made up of nodes, each with at most Binary trees are among the most traditional hierarchical data structures known to Bring a middle ground between the relatively simplistic linear data structures (such as arrays & linked lists) and the more complex general graphs. world which makes it an important data structure in a programmer's toolkit. Binary trees binary tree is an elegant data structure that enables performing efficient searching, sorting, and organizing of

data. This hierarchical aspect resembles many relationships/problems in the real The Allow for certain algorithmic benefits. standard binary tree. These are specialized adaptations of the basic binary tree framework, but with specific constraints that Before diving into specific types of binary trees, let us take a moment to overview some specialized types of binary trees such as complete binary trees, extended binary trees that are quite different from the

Unit 13: Types of Binary Tree

5.3 Types of Binary Trees: Complete Binary Tree and Extended Binary Tree

In a very specific structure with important characteristics. The nodes are as far to the left as they can go. This kind of description follows If every level of a binary tree is full except the last one, and all of the nodes are in that level, from top to bottom and left to right, there will be a connection between each node and its parent and child nodes. This is called a "Complete Binary Tree." For node i in the list: In a Complete Binary Tree, nodes are numbered from 1 to n, with n being at position 2i. The left child (if there is one) is at position 2i+1, and the right child (if there is one) is at position 2i. Its parent is at place i/2, but not the root.

Complete Binary Trees are great for array-based solutions because they have a regular structure. In these cases, the tree structure is hidden in the array indices instead of being shown directly with pointer-based nodes.

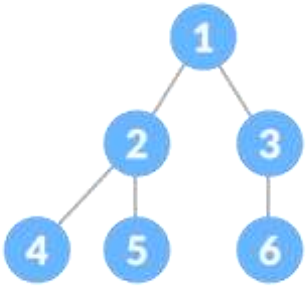

Figure 25: Complete Binary Tree
[Source: https://www.programiz.com/]

What Complete Binary Trees Can Do

Complete Binary Trees have a few important features that make them useful in many situations:

Weight: A Full Binary Tree with n nodes is $\lfloor \log_2(n)$ tall. which is the smallest height that a binary tree with n nodes can have. This property makes sure that processes that depend on the height of the tree work as efficiently as possible. Compact Representation: The regular structure lets you use an array-based implementation that takes up little room because it doesn't need explicit pointers. This saves memory. Predictable Structure: The positions of parents and children

follow a mathematical pattern, which means that easy math can be used instead of pointer traversals. Balance Guarantee: Complete Binary Trees aren't always exactly balanced, but they're usually pretty close. This keeps them from becoming highly skewed, which is the worst thing that could happen. It's easy to add new nodes; at the lowest level, they are always added from left to right, which makes adding nodes easy.

How to Show Complete Binary Trees in an Array One of the best things about Complete Binary Trees is that they can be represented efficiently using groups. It shows that:

• The root node is kept at index 1 (or sometimes 0 based on the implementation);

• A node at index i has a left child at index 2i (or 2i+1 if you start at 0).

• The right child of a node at index i is at index 2i+1 (or 2i+2 if you start at 0).

• The parent of a node at index i is at index ⌊i/2⌋ (or ⌊(i-1)/2⌋ if you start at 0). This approach based on arrays has several advantages:

• Less memory is used because there is no need for explicit pointer storage; • Better cache locality because memory is allocated all at once;

• Access to any node at any time based on its number

This representation, on the other hand, only saves room for Complete Binary Trees. For sparse trees, a lot of array places would be empty, which would waste memory.

Complete binary trees are used for many things.

A lot of important data structures and methods are built on top of complete binary trees.

1. Binary Heaps: Both min-heaps and max-heaps are implemented as Complete Binary Trees. They are the building blocks for efficient priority queues and the heap sort method.

2. More complex tree structures: A lot of self-balancing tree structures, like AVL trees and Red-Black trees, try to stay full or almost complete to make sure they work well.

3. Tournament Trees: These trees group elements in a tournament-like way so that they can be found quickly in selection algorithms and external sorting.

4. Huffman Coding Trees: These trees aren't always complete, but they are built and traversed in a way that takes advantage of completeness qualities.

5. Database Indexing: Complete Binary Trees are good indexing designs for database management systems that give you predictable performance.

Thoughts on Implementation

When making a Complete Binary Tree, there are a few design choices that affect how well it works:

Array vs. Node-Based: Arrays work well in most situations, but node-based implementations with explicit pointers may be better in settings that change often or where the tree size is hard to predict. Two types of indexing: zero-based indexing (root at index 0) works better with standard computer languages, while one-based indexing (root at index 1) makes it easier to figure out the parent-child index. Dynamic Resizing: When the tree size grows bigger than the original allocation, it's important to have plans for how to grow the array. Level Tracking: Some actions, like adding and removing records, can be made faster by keeping track of the current bottom level. Algorithms for crossing borders: For Complete Binary Trees, level-order navigation works naturally. Other traversal methods, like in-order, pre-order, and post-order, may need different approaches compared to general binary trees.

Making Sure It's Complete It takes careful attention to keep the completeness feature during additions and deletions: When adding nodes, they must always go to the farthest left empty spot in the lowest level. If that spot is taken, a new level is created from the farthest left place.

When a node is removed, the rightmost node in the lowest level is usually used to replace it. Once the node has been replaced, the tree may need to be restructured to keep some properties, like the heap property in a binary heap. Analysis of Efficiency

The fact that simple operations on Complete Binary Trees take a long time shows how efficiently they are built: Access takes O(1) time with an array implementation and O(log n) time with pointer-based traversal. Search takes O(log n) time in most cases and O(n) time in the worst cases for data that is not in the right order. Insertion takes

O(log n) time when only completeness is maintained and O(1) time when adding to the end.

• Deletion takes O(log n) time when only completion is kept.

• Both array and pointer versions take up O(n) space.

Because they are efficient in these ways, Complete Binary Trees can be used in situations where speed needs to be guaranteed.

What are extended binary trees? What do they mean?

This type of tree has either 0 or 2 children for each node. It is also called a 2-tree or a correct binary tree. In other words, in an Extended Binary Tree, each internal node must have both a left and a right child. Node 0 has only one child. This condition makes a structure where the internal nodes (nodes with children) and external nodes (leaf nodes without children) are connected in a certain way. To show extended binary trees, "external nodes" or "null nodes" are sometimes added to show the children of leaf nodes that are not present in a normal

Binary tree- Properties of Structure

There are a few interesting things about the structure of extended binary trees:

To find the number of foreign nodes in an Extended Binary Tree, take n and add one to it. This is called the node relationship. When you add up the lengths of all the paths that go outside and inside a node, and then subtract the lengths of all the paths that go inside and outside that node, you get the number 2n.

1. **Height Bounds:** For an Extended Binary Tree with n internal nodes, the minimum height is $\lfloor \log_2(n) \rfloor$, achieved when the tree is complete, and the maximum height is n, occurring in a linear arrangement.
2. **Perfect Subtrees:** Every subtree of an Extended Binary Tree is also an Extended Binary Tree, preserving the 0-or-2 children property recursively.
3. **Isomorphism:** Extended Binary Trees have direct isomorphisms with full binary trees, making conversions between the two straightforward.

Mathematical Foundation

The mathematical elegance of Extended Binary Trees appears in several contexts:

- **Catalan Numbers:** The number of distinct Extended Binary Trees with n internal nodes is given by the nth Catalan number, Cn = (1/(n+1))(2n choose n).
- **Binary Tree Enumeration:** Extended Binary Trees provide a canonical form for enumerating and counting binary tree structures with specific properties.
- **Graph Theory Connections:** Extended Binary Trees represent a specific class of planar graphs with applications in computational geometry and network design.
- **Combinatorial Interpretations**: The structure of Extended Binary
- Trees see integer sequences like the Catalan numbers, where correspondences with other combinatorial objects exist, such as balanced parentheses expressions or lattice paths.

Representation Methods

Here are multiple ways to represent Extended Binary Trees:

- **Explicit Null Representation:** All potential positions for child nodes are filled; an actual node or an explicit null node is used, resulting in a uniform structure.
- Bit-String Encoding The structure of this tree can be encoded as a bit string by a preorder traversal, with internal nodes marked as 1 and external nodes as 0.
- **Parenthesis Notation**: The structure of non-empty Extended Binary Tree can be represented in the form of a balanced parenthesis expression, where matching pairs of parentheses represent the internal nodes.
- **Rray-Based Representation:** Similar to complete binary trees but with explicit markers for missing nodes to provide structural properties.
- **Pointer-Style Implementation:** Classic node structures, with left and right both as pointers, but with the constraint that (1) both are either null or (2) both point to another node.

Extended Binary Trees Application - Extended Binary Trees are used in different fields:

- Expression Trees: Extended Binary Trees represent operations (internal nodes) and operands (external nodes) naturally in the context of compiler design and evaluating mathematical expressions.
- Huffman Compression: The trees used in Huffman compression are extended binary trees in which the external nodes represent different characters and their frequencies.
- Extended Binary Trees: A specialized form of binary tree is the decision tree — in machine learning and decision analysis context.
- Extended Binary Trees for Game Trees: For games with a binary choice, Extended Binary Trees model the possible states of the game and their transitions, being of particular importance in the implementation of the minimax algorithm.
- Parse trees: Some parsing algorithms used in formal language theory and compiler construction will generate an Extended Binary Tree representation of the syntactic structure of some expressions.

Several algorithmic techniques are based on extended binary trees:

- From tree traversal algorithms: The uniform structure allows for simple tree traversal algorithms, particularly those that need to make explicit reference to external nodes.
- Tree Transformation: Algorithms for translating between various representations of trees frequently employ Extended Binary Trees as a working format behind the scenes, thanks to their regular shape.
- Another class of algorithms works with the nature of Extended Binary Trees, preserving height or path lengths.
- Regular Trees : Regular structure allows for pattern matching in hierarchical data stored as trees.
- Dynamic Programming on Trees Binary Trees can consider several structural properties that can be applied to the tree, such as maximum path length, optimal subtree selection, etc.

Construction and Maintenance

- When constructing and maintaining Extended Binary Trees, the following considerations should be taken into account:
- Top-Down Construction: Beginning with a root, progressively builds subtrees while preserving the 0-or-2 children property at every stage.
- Heuristic Search: Use of a branch and bound technique to reduce complexity when searching an extended binary tree.
- Conversion from General Binary Trees: By inserting external nodes in accordance with the missing children, any binary tree can be converted into an Extended Binary Tree.

Operation Adjustment: Rotation, specialized for balancing trees, changes the properties of the tree preserving the Binary Tree while trying to balance the path length or height of the tree. Incremental Updates: It may take more complex actions to ensure that insertions and deletions preserve the Extended Binary Tree property, as compared to general binary trees.

Comparison and Relationships

Complete vs. Extended Binary Trees: Complete Binary Trees and Extended Binary Trees are both special types of binary trees, but they are different in several key aspects:

1 Structural Focus:

- Complete Binary Trees focus on the arrangement of nodes across levels, ensuring fullness from left to right.
- Extended Binary Trees focus on the branching pattern, requiring each node to have either 0 or 2 children.

2. Node Distribution:

- Complete Binary Trees have a predictable distribution of nodes across levels, with possibly only the last level being partially filled.
- Extended Binary Trees can have more variable level populations, as long as the 0-or-2 children constraint is maintained.

3. Mathematical Properties:

- Complete Binary Trees are characterized by minimal height and level-order fullness.
- Extended Binary Trees are characterized by internal-external node count relationships and Catalan number enumeration.

4. Implementation Efficiency:

- Complete Binary Trees excel in array-based implementations due to their regular level structure.
- Extended Binary Trees may favor node-based implementations that explicitly enforce the branching constraint.

5. Application Domains:

- Complete Binary Trees are often used in applications requiring efficient searching and sorting.
- Extended Binary Trees are frequently used in expression evaluation and decision modeling.

Hybridization and Specialized Forms

Combinations of Complete and Extended Binary Tree properties create specialized structures:

- Perfect Binary Trees: Trees that are both complete and extended, where all internal nodes have exactly two children and all leaf nodes are at the same level.
- Heap-Ordered Extended Trees: Extended Binary Trees with heap-ordering properties, used in specialized priority queue implementations.
- Almost Complete Extended Trees: Extended Binary Trees that minimize height variance, combining completeness goals with branching constraints.
- Balanced Extended Trees: Extended Binary Trees with additional balance constraints, ensuring logarithmic height while maintaining the 0-or-2 children property.
- Threaded Extended Trees: Extended Binary Trees with threading to facilitate traversal, combining the structural properties with traversal optimization.

Transformation Algorithms

Converting between different tree types involves specific algorithms:

- Complete to Extended Transformation: Involves adding explicit external nodes to a Complete Binary Tree at appropriate positions to satisfy the 0-or-2 children constraint.
- Extended to Complete Transformation: Typically requires restructuring and potentially adding or removing nodes to ensure the level-order fullness property.
- General Binary Tree to Complete/Extended: Transformation algorithms that convert arbitrary binary trees to either specialized form, often used in preprocessing for specific algorithms.
- Minimal Transformation Approaches: Algorithms that find the minimal number of operations to convert between tree types, useful in tree edit distance problems.
- Incremental Transformation: Methods that maintain both properties simultaneously during incremental construction, avoiding costly whole-tree transformations.

Here’s a properly formatted and structured version of your explanation along with Python code samples for implementing a Complete Binary Tree using an array-based approach and an Extended Binary Tree using a node-based approach.

Implementing a Complete Binary Tree (Array-Based Approach)

A Complete Binary Tree (CBT) is a type of binary tree where all levels are completely filled except possibly the last, which is filled from left to right. The array-based implementation simplifies parent-child relationships using index calculations.

Python Implementation

```
class CompleteBinaryTree:
    def __init__(self):
self.array = [None]  # Index 0 is unused for easier arithmetic
self.size = 0

    def insert(self, value):
        """Inserts a new value into the complete binary tree."""
self.size += 1
self.array.append(value)

    def left_child_index(self, parent_index):
        """Returns the index of the left child."""
        return 2 * parent_index

    def right_child_index(self, parent_index):
        """Returns the index of the right child."""
        return 2 * parent_index + 1

    def get_parent_index(self, child_index):
        """Returns the index of the parent node."""
        return child_index // 2

    def has_left_child(self, index):
        """Checks if the node at the given index has a left child."""
        return self.left_child_index(index) <= self.size

    def has_right_child(self, index):
        """Checks if the node at the given index has a right child."""
        return self.right_child_index(index) <= self.size

    def left_child(self, index):
        """Returns the left child value."""
```

```
        return self.array[self.left_child_index(index)] if self.has_left_child(index) else None

    def right_child(self, index):
        """Returns the right child value."""
        return self.array[self.right_child_index(index)] if self.has_right_child(index) else None

    def parent(self, index):
        """Returns the parent node value."""
        return self.array[self.get_parent_index(index)] if index > 1 else None

# Example Usage
tree = CompleteBinaryTree()
tree.insert(10)
tree.insert(20)
tree.insert(30)
tree.insert(40)
tree.insert(50)

print("Parent of node at index 3:", tree.parent(3))  # Output: 10
print("Left child of node at index 1:", tree.left_child(1))  # Output: 20
print("Right child of node at index 1:", tree.right_child(1))  # Output: 30
```

Implementing an Extended Binary Tree (Node-Based Approach)

An Extended Binary Tree is a variation where external nodes (leaf placeholders) are explicitly represented. This allows for easier manipulation of tree structure, commonly used in expression trees, Huffman coding trees, and decision trees.

Python Implementation

```
from typing import Optional

class Node:
    def __init__(self, value: Optional[int] = None):
        """Creates a node with optional value. If value is None, it is considered an external node."""
self.value = value
```

```
self.left = None
self.right = None
self.is_external = (value is None)

class ExtendedBinaryTree:
    def __init__(self):
        """Initializes an empty tree with no root."""
self.root = None

    def create_internal_node(self, value):
        """Creates an internal node with the given value and two external children."""
        node = Node(value)
node.left = Node()  # External node (Leaf placeholder)
node.right = Node()  # External node (Leaf placeholder)
        return node

    def replace_external_with_internal(self, parent, is_left, value):
        """Replaces an external node with an internal node and returns the new node."""
new_internal = self.create_internal_node(value)
        if is_left:
parent.left = new_internal
        else:
parent.right = new_internal
        return new_internal

    def insert(self, value, start_index=0):
        """Inserts a value into the binary tree, replacing external nodes when needed."""
        if self.root is None:
self.root = self.create_internal_node(value)
            return self.root

        current = self.root
        while True:
            if current.left.is_external:
```

```
            return self.replace_external_with_internal(current, True, value)
elifcurrent.right.is_external:
            return self.replace_external_with_internal(current, False, value)
        else:
            # Traverse to the left or right child for further insertion
            current = current.left if start_index % 2 == 0 else current.right

# Example Usage
ebt = ExtendedBinaryTree()
ebt.insert(10)
ebt.insert(20)
ebt.insert(30)

print("Root Node:", ebt.root.value)  # Output: 10
print("Left Child of Root:", ebt.root.left.value)  # Output: 20
print("Right Child of Root:", ebt.root.right.value)  # Output: 30
```

Crossing (DFS, BFS) In this lesson, you will learn about graphs, operations research, and graphs. that they go both ways between two points. Graphs are used in many places, such as computer networks, social networks, and to map out places. A lot of algorithm problems in AI are solved by going from one point to the next. Graphs without edges, on the other hand, don't have lines that go in any one direction. In other words, a graph can be either directed or not directed. In a directed graph, the points (vertices) are linked by lines. In computer science, edges are types of data structures that show how things are linked to each other. A graph is made up of nodes (also called edges) and points (also known as edges). An important part of a graph is the point. Trees and other types of acyclic graphs don't have rings. Every line in an unweighted graph looks the same. Graphs can also be either acyclic or cyclic. There must be at least one cycle in a cyclic graph. A cycle is a line that starts and ends at the same weighted or unweighted graph. Graphs with Weights: In a weighted graph, each edge has a number value that goes with it. This value is generally a cost, distance, or capacity. You can only have one edge between each pair of points; there can't be any loops. The graph can also be a

multigraph, which means that two vertices can be linked by more than one line. Another group is made based on weights that are determined by how they are built and what features they have. To put it another way, they can be easy graphs. You can group graphs into ones that are thick but take up more space. pair of points or not. This way of showing things works well for plots with points. Another name for an adjacency matrix is a 2D array. The lengths of the array are the vertices, and the number of each cell tells you if that cell has an edge. This turns a graph into a list. It takes up less memory to show sparse graphs this way, where the number of lines is less than the square of the number of nodes. There are two main ways to show a graph in memory: adjacency matrices and adjacency lists. To make graph methods work well, you need an adjacency list for Represent graph. There are two. To search, find paths, and do general graph network analysis, as well as other hard computer jobs, it's easier when the data is in the right format. Our most basic graph traversal methods are Breadth-First Search (BFS) and Depth-First Search (DFS). If you want to graph in an organized way, most people use lambda. We can at least break This is the most basic steps in the study of graphs. We check out each vertex of a source vertex and mark them using a queue. This way, the vertices of tiles are checked out in the order that they were found. Beginning with a before the next group of points. These things are done in real life. Breadth-First Search (BFS) finds its way through a maze. goes to all of a vertex's neighbors. It is a type of level-order navigation. E = edges A lot of the time, BFS is used in network transmission, shortest path algorithms, and puzzles like "find the shortest path." BFS will take O(V + E) time, where V is the number of vertices and E is the number of edges. It will process each vertex and put all the nodes next to it that haven't been visited into a queue. This keeps happening until all reachable vertices have been visited and put in the queue. Then, while something is still in the queue, it removes it and checks to see if it's connected. O(V + E) DFS is used to sort graphs by their topology, find cycles, and figure out how to get out of mazes. We haven't looked into DFS Time Complexity yet. At the same time, if all the vertices next to it are visited, it goes back to the last one it visited and keeps going using stack (it can be explicit stack using stack data structure or implicit stack using recursion). This algorithm begins at a source vertex,

marks it as visited, and then visits all nearby vertices in a loop before going backwards. DFS with stack | It's Depth-First Search (DFS) is another important graph traversal method that uses graph theory to solve a wide range of real-world problems by going as far down a branch as possible. and growing over and over DFS are made to be used in certain situations. When looking at a path in an unweighted graph, traversals play a big role. On the other hand, DFS works better for searching deeper in the search space. These algorithms come in different versions that are better, such as bidirectional BFS. DFS each have their own strengths that depend on the issue. When looking for the shortest path in a graph, BFS is often thought to be the best method because it gets the shortest path.

MCQs:

1. **Which of the following is NOT a property of a tree?**
   a) Acyclic structure
   b) One root node
   c) Multiple parent nodes per child
   d) Connected nodes
2. **What is the maximum number of children a node can have in a binary tree?**
   a) 1
   b) 2
   c) 3
   d) Unlimited
3. **Which data structure is used to implement Breadth-First Search (BFS)?**
   a) Stack
   b) Queue
   c) Linked List
   d) Heap
4. **What is the difference between a tree and a graph?**
   a) A tree has cycles, while a graph does not
   b) A graph allows multiple connections, while a tree follows a hierarchy
   c) A graph has only one root node
   d) Trees do not store data, but graphs do
5. **Which of the following is a type of binary tree?**
   a) Complete Binary Tree

b) Linked List
c) Queue Tree
d) Graph Tree

6. **What is the time complexity of Depth-First Search (DFS) in an adjacency list?**
   a) O(n)
   b) O(log n)
   c) O(V + E)
   d) O(V * E)
7. **In a full binary tree, every node has either:**
   a) 1 child or 2 children
   b) 0 or 2 children
   c) 3 children
   d) Unlimited children
8. **Which of the following data structures is used to implement DFS?**
   a) Queue
   b) Stack
   c) Priority Queue
   d) Hash Table
9. **Which traversal method visits all children of a node before moving deeper?**
   a) DFS
   b) BFS
   c) Inorder
   d) Postorder
10. **What is the minimum number of edges required to form a connected graph with N nodes?**
    a) N-1
    b) N+1
    c) 2N
    d) N^2

A Few Questions:

1. What does the word "tree" mean in mathematics?
2. Describe what binary trees are and how they work.

What is the difference between a full binary tree and a binary tree that has been extended?

4. Explain what a graph is and the different kinds of them.

5. What are some good things about using graphs in data structures?
6. How does Breadth-First Search (BFS) work?
7. Talk about the Depth-First Search (DFS) method.
8. How are BFS and DFS different from each other?
9. How long does it take to solve DFS and BFS?
10. How is an adjacency list used to show a graph?

Long Questions:

1. Give some examples to show how trees and graphs work.
2. Draw and explain the different kinds of binary trees.
3. Write a C++ program that adds items to a binary tree and moves through it.
4. List the ways that trees and graphs are similar and different in terms of form and use.
5. Talk about the pros and cons of the different ways to traverse a tree.
6. Come up with a way to do Breadth-First Search (BFS) and Depth-First Search (DFS).
7. Talk about how the Adjacency Matrix and Adjacency List that show graphs work.
8. Make a program in C++ that shows DFS on a graph.
9. Talk about how graphs are used in networking and AI in the real world.
10. Compare BFS and DFS in terms of how long they take and how they are used.

References

Chapter 1: Introduction to Data Structure

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2023). Algorithms (4th ed.). Addison-Wesley Professional.
3. Drozdek, A. (2022). Data Structures and Algorithms in C++ (5th ed.). Cengage Learning.
4. McDowell, G. L. (2021). Cracking the Coding Interview: 189 Programming Questions and Solutions (6th ed.). CareerCup.
5. Malik, D. S. (2022). C++ Programming: From Problem Analysis to Program Design (8th ed.). Cengage Learning.

Chapter 2: Array

1. Knuth, D. E. (2020). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley Professional.
2. Lafore, R. (2021). Data Structures and Algorithms in Java (3rd ed.). Sams Publishing.
3. Karumanchi, N. (2022). Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles (6th ed.). CareerMonk Publications.
4. Weiss, M. A. (2023). Data Structures and Algorithm Analysis in C++ (5th ed.). Pearson.
5. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2022). Data Structures and Algorithms in Java (7th ed.). Wiley.

Chapter 3: Stack

1. Horowitz, E., & Sahni, S. (2022). Fundamentals of Data Structures in C++ (3rd ed.). W. H. Freeman.
2. Levitin, A. (2021). Introduction to the Design and Analysis of Algorithms (4th ed.). Pearson.
3. Morin, P. (2023). Open Data Structures: An Introduction. AU Press.
4. Preiss, B. R. (2022). Data Structures and Algorithms with Object-Oriented Design Patterns in C++ (2nd ed.). Wiley.
5. Shaffer, C. A. (2023). Data Structures and Algorithm Analysis (4th ed.). Dover Publications.

Chapter 4: Linked List

1. Stroustrup, B. (2022). The C++ Programming Language (5th ed.). Addison-Wesley Professional.
2. Skiena, S. S. (2020). The Algorithm Design Manual (3rd ed.). Springer.
3. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2021). Algorithms (2nd ed.). McGraw-Hill Education.
4. Stephens, R. (2023). Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C++ (2nd ed.). Wiley.
5. Kleinberg, J., & Tardos, É. (2022). Algorithm Design (2nd ed.). Pearson.

Chapter 5: Tree and Graph

1. Sedgewick, R. (2022). Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching (3rd ed.). Addison-Wesley Professional.
2. Manber, U. (2020). Introduction to Algorithms: A Creative Approach (2nd ed.). Addison-Wesley Professional.
3. Mehta, D. P., & Sahni, S. (2022). Handbook of Data Structures and Applications (2nd ed.). Chapman and Hall/CRC.
4. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2023). Compilers: Principles, Techniques, and Tools (3rd ed.). Pearson.
5. Even, S. (2021). Graph Algorithms (2nd ed.). Cambridge University Press.

MATS UNIVERSITY

MATS CENTER FOR OPEN & DISTANCE EDUCATION

UNIVERSITY CAMPUS : Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 M : 9109951184, 9755199381 Toll Free : 1800 123 819999

eMail : admissions@matsuniversity.ac.in Website : www.matsodl.com