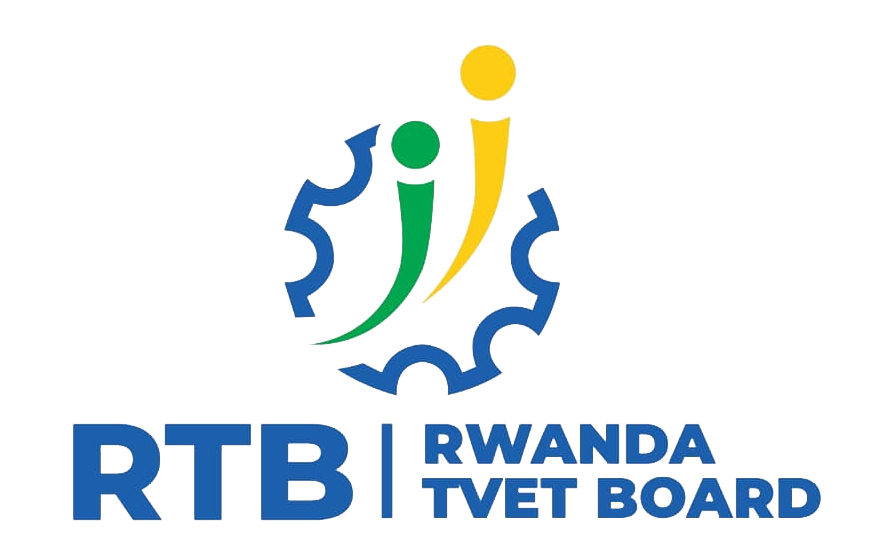
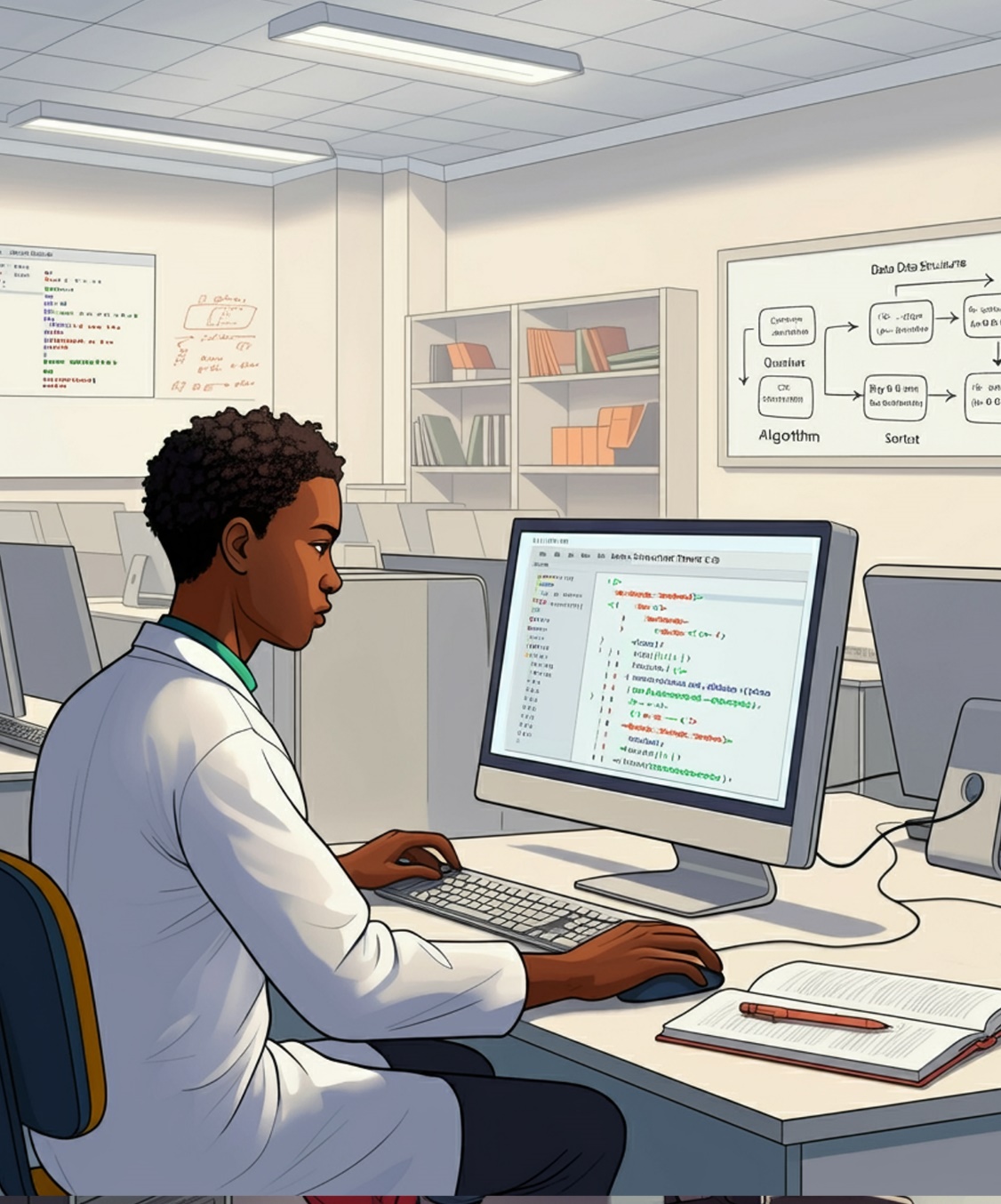
**RQF LEVEL 5**



***TRAINEE'S MANUAL***

**GENDS501**

**COMPUTER SYSTEM**

**AND ARCHITECTURE**

***October, 2024***

**Data Structures**

**And Algorithms**

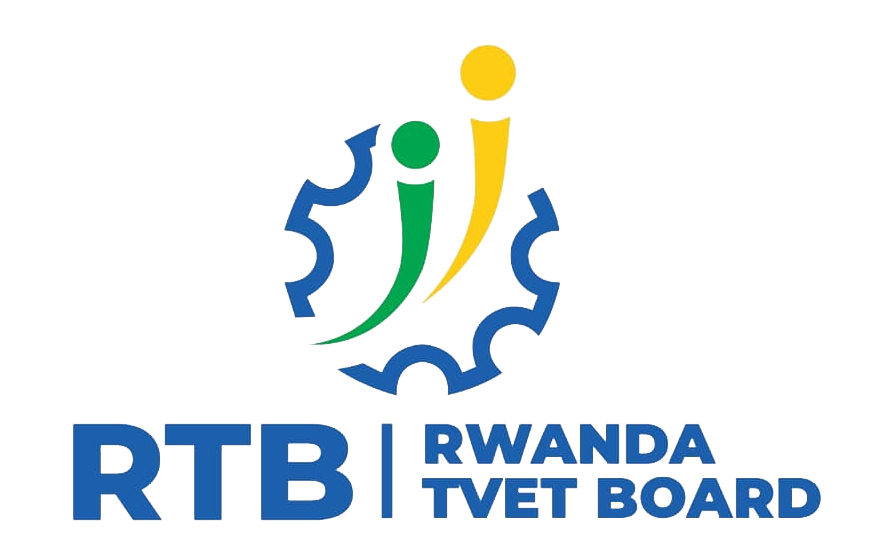
**Using C**

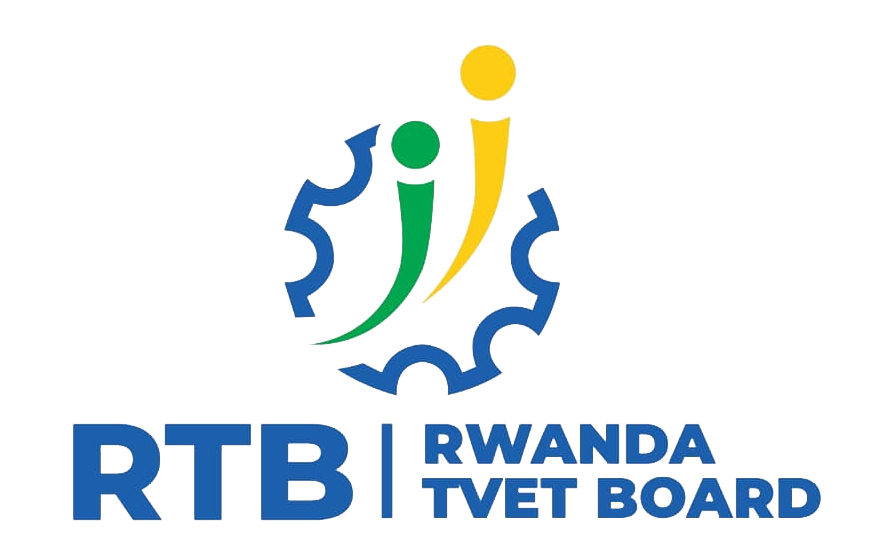
**2024**

2022

**DATA STRUCTURES AND ALGORITHMS USING C**



AUTHOR’S NOTE PAGE (COPYRIGHT)

The competent development body of this manual is Rwanda TVET Board ©, reproduce with permission.

All rights reserved.

* This work has been produced initially with the Rwanda TVET Board with the support from KOICA through TQUM Project
* This work has copyright, but permission is given to all the Administrative and Academic Staff of the RTB and TVET Schools to make copies by photocopying or other duplicating processes for use at their own workplaces.
* This permission does not extend to making of copies for use outside the immediate environment for which they are made, nor making copies for hire or resale to third parties.
* The views expressed in this version of the work do not necessarily represent the views of RTB. The competent body does not give warranty nor accept any liability
* RTB owns the copyright to the trainee and trainer’s manuals. Training providers may reproduce these training manuals in part or in full for training purposes only. Acknowledgment of RTB copyright must be included on any reproductions. Any other use of the manuals must be referred to the RTB.

© **Rwanda TVET Board**

*Copies available from:*

* *HQs: Rwanda TVET Board-RTB*
* *Web:* [*www.rtb.gov.rw*](http://www.rtb.gov.rw)
* **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer’s and trainee’s manuals for the TVET Certificate V in Computer System and Architecture, specifically for the module "**GENDS501: Data Structures and Algorithms Using C**".

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

**This training manual was developed:**

Under Rwanda TVET Board (RTB) guiding policies and directives



Under Financial and Technical support of



**COORDINATION TEAM**

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Asher Emmanuel

**Production Team**

**Authoring and Review**

IZABAYO Marie Rose

NIYONSABA Godelive

**Validation**

IKIREZI Pacifique

NAYIGIZIKI Theogene

**Conception, Adaptation and Editorial works**

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

NIYOMUGABO Silas

**Formatting, Graphics, Illustrations, and infographics**

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

HABIMANA Emmanuel

**Financial and Technical support**

KOICA through TQUM Project

TABLE OF CONTENT

[AUTHOR’S NOTE PAGE (COPYRIGHT) iii](#_Toc183172910)

[ACKNOWLEDGEMENTS iv](#_Toc183172911)

[TABLE OF CONTENT vii](#_Toc183172912)

[ACRONYMS ix](#_Toc183172913)

[INTRODUCTION 1](#_Toc183172914)

[MODULE CODE AND TITLE: GENDS501 DATA STRUCTURES AND ALGORITHMS USING C 2](#_Toc183172915)

[Learning Outcome 1: Identify Data Structures and Algorithms 3](#_Toc183172916)

[Key Competencies for Learning Outcome 1: Identify Data Structures and Algorithms 4](#_Toc183172917)

[Indicative content 1.1 : Identification of Data Structures Concepts 6](#_Toc183172918)

[Indicative content 1.2 : Description of Algorithms 20](#_Toc183172919)

[Indicative content 1.3: Analysing Algorithmic Concepts 32](#_Toc183172920)

[Indicative content 1.4: Writing Algorithms 40](#_Toc183172921)

[Indicative content 1.5: Preparation of C Programming Environment 50](#_Toc183172922)

[Learning outcome 1 end assessment 56](#_Toc183172923)

[**References** 59](#_Toc183172924)

[Learning Outcome 2: Apply Linear data structure 60](#_Toc183172925)

[Key Competencies for Learning Outcome 2: Apply Linear Data Structure 61](#_Toc183172926)

[Indicative Content 2.1: Identification of Linear Data Structure Concepts 64](#_Toc183172927)

[Indicative Content 2.2: Applying Arrays Data Structure 70](#_Toc183172928)

[Indicative Content 2.3: Applying Linked list Data Structure 81](#_Toc183172929)

[Indicative Content 2.4: Applying Queue Data Structure 96](#_Toc183172930)

[Indicative content 2.5: Applying Stack Data Structure 112](#_Toc183172931)

[Learning outcome 2 end assessment 126](#_Toc183172932)

[**References** 128](#_Toc183172933)

[Learning Outcome 3: Apply Non-linear Data Structure 129](#_Toc183172934)

[Key Competencies for Learning Outcome 3: Apply Non-linear Data Structure 130](#_Toc183172935)

[Indicative content 3.1: Identification of Non-Linear Data Structures Concepts 133](#_Toc183172936)

[Indicative content 3.2: Applying Tree Data Structure 142](#_Toc183172937)

[Indicative content 3.3: Applying graph data structure 156](#_Toc183172938)

[Indicative content 3.4: Applying Hash Table Data Structure 170](#_Toc183172939)

[Learning outcome 3 end assessment 186](#_Toc183172940)

[**References** 188](#_Toc183172941)

ACRONYMS

**! :** Logical NOT operator

**! =** Inequality comparison

**#:** Preprocessor directive (e.g., #include, #define)

**% :** Modulus operator (remainder)

**&&:** Logical AND operator

**& :**  Address-of operator or bitwise AND operator, used to get the address of a variable (e.g., &size).

**( ) :** Function parameters or precedence in expressions, Parentheses used in function calls and to define the order of operations.

**\* :** Multiplication operator or pointer Division operator

**: :** Ternary conditional operator or case label in switch

**; :** Semicolon, used to terminate statements.

**[] :** Array subscripting

**^ :** Bitwise XOR operator`

**{ } :** Curly braces used to define a block of code, such as function bodies or control statements.

**~ :** Bitwise NOT operator

**+ - :** Addition / Subtraction operator

**< > :** Less than / Greater than comparison

**<< :** Left shift operator

**<= >= :** Less than or equal to / Greater than or equal to comparison

**=:** Assignment operator

**== :** Equality comparison

**-> :** Member access operator (for pointers to structures)

**-> :** Used to access members of a structure through a pointer (e.g., node->data).

**>> :** Right shift operator

**ADT**: Abstract Data Type

**API**: Application Programming Interface

**arr**: Array (a common abbreviation used in code to denote an array variable)

**BFS:** Breadth-First Search

**BST**: Binary Search Tree

**C:** C Programming Language

**DAG**: Directed Acyclic Graph

**DFS**: Depth-First Search

**FIFO**: First-In-First-Out

**IDE:** Integrated Development Environment

**JPEG**: Joint Photographic Experts Group

**LIFO**: Last In, First Out

**MAX**: Maximum (often used to define the maximum size of an array or structure)

**n:** Number of elements (commonly used as a variable to represent the size of an array)

**O(log n):** Big O notation representing logarithmic time complexity

**O(n):** Big O notation representing linear time complexity

**O(V + E):** Big O notation representing time complexity in graph algorithms, where V is the number of vertices and E is the number of edges

**OOP**: Object-Oriented Programming

**PNG:** Portable Network Graphics

**RTB:** Rwanda TVET Board

**TQUM:** TVET Quality Management Project

**ZIP:** Zone Improvement Plan

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in Computer System and Architecture specifically for the module of **"Data Structures and Algorithms using C."** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: GENDS501 DATA STRUCTURES AND ALGORITHMS USING C

**Learning Outcome 1: Identify Data Structures and Algorithms**

**Learning Outcome 2: Apply Linear data structure**

**Learning Outcome 3: Apply Non-linear Data Structure**

Learning Outcome 1: Identify Data Structures and Algorithms



|  |
| --- |
| **Indicative contents**  **1.1 Identification of data structures concepts**  **1.2 Description of algorithms**  **1.3 Analysing algorithmic concepts**  **1.4 Writing algorithms**  **1.5 Preparation of C programming environment** |

## Key Competencies for Learning Outcome 1: Identify Data Structures and Algorithms

|  |  |  |
| --- | --- | --- |
| **Knowledge** | **Skills** | **Attitudes** |
| * Description of data structures and algorithms key terms * Classifications of data structures * Identification of Data types in algorithm * Description of Data structure operations * Description of algorithms * Description of analysis of Algorithmic cases and Complexity * Description of different ways of Developing Data structures * Identification of Tools in C programming environment | * Using data types in algorithm * Using Data structure operations in algorithm * Applying Searching and Sorting algorithms * Developing Data structures * Performing sorting and searching operations * Testing development environment * Installing tools in C programming environment | * Having Critical thinking while using data structure operations * Being innovative while applying searching and sorting algorithms * Being Creative while developing Data structures * Having problem solving skills while performing sorting and searching operations * Having Focus while testing development environment |

|  |  |  |
| --- | --- | --- |
| **Duration: 20 hrs** | | |
| **Learning outcome 1 objectives**:  By the end of the learning outcome, the trainees will be able to:   1. Describe clearly data structures and algorithms key terms based on intended use 2. Classify correctly data structures and data types in algorithm based on algorithm standards 3. Identify properly data types in algorithm based on intended use 4. Use properly data types in algorithm based on intended use 5. Describe effectively Data structure operations in algorithm based on algorithm standards 6. Use effectively Data structure operations in algorithm based on algorithm standards 7. Describe clearly different ways of developing data structures based on their use cases 8. Analyse correctly Algorithmic cases and Complexity based on intended use 9. Apply effectively Searching and Sorting algorithms and perform their operations in algorithm base on the intended use 10. Test effectively development environment based on programming language standards 11. Identify correctly tools in C programming environment based on language standards 12. Install effectively tools in C programming environment based on language standards | | |
| **Resources** | | |
| **Equipment** | **Tools** | **Materials** |
| * Computer * Projector | * C Compiler * Integrated Development Environment (IDE) * Text Editors * VisuAlgo | * Internet * Electricity |

## Indicative content 1.1 : Identification of Data Structures Concepts



****

**Duration: 5 hrs**



**Theoretical Activity 1.1.1: Description of data structures concepts**

**Tasks:**

1: Answer the following questions related to data structure:

1. What do you understand by the following terms?:

a. Data

b. Structure

c. Data structures

d. List

e. Searching

f. Sorting

g. Keys

h. Index

1. Classify data structures
2. Identify data types in algorithm
3. Explain List representation.
4. Describe Data structure operations

2: Write your answers on flipcharts/papers.

3: Present the findings/answers to the whole class or trainer

4: Ask questions where necessary.

5: Read the key readings 1.1.1.

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 1.1.1.: Description of data structures concepts**   1. **Description of key terms**    1. **Data**   In computing, data refers to raw, unorganized facts or values that can be processed or manipulated by an algorithm or program. It can represent information such as numbers, characters, or symbols.  **Example:** An integer like 42, a string like "Hello", or a boolean value like true are all examples of data.   * 1. **Structure**   Structure refers to the way data is organized, stored, and accessed in memory. In algorithms, structures are designed to efficiently manage and operate on data.  **Example:** Arrays, linked lists, and trees are examples of structures that organize data for efficient access and modification.   * 1. **Data Structures**   Data structures are specialized formats for organizing, storing, and managing data. They determine how data is arranged in memory and how algorithms interact with the data.  **Example:**  **Array:** Stores elements of the same type in contiguous memory locations.  **Linked List:** A sequence of nodes where each node points to the next node.  **Stack:** A data structure where elements are added or removed in a Last-In-First-Out (LIFO) manner.   * 1. **List**   A list is a linear data structure that represents a sequence of elements. Each element is typically stored in a specific position, and it allows for various operations such as addition, deletion, or access of elements.  **Example:**  **Array List:** A list implemented as an array where elements can be accessed by their index.  **Linked List:** A list where elements are stored in nodes connected by pointers.   * 1. **Searching**   Searching is the process of finding a specific element or a group of elements in a data structure. It’s one of the most fundamental operations in algorithms.  **Types:**   * **Linear Search:** Traverses the list sequentially to find the desired element. * **Binary Search:** A more efficient search for sorted lists where the list is repeatedly divided into halves.   **Example:** Finding a specific book in a library catalog by searching its title.   * 1. **Sorting**   Sorting is the process of arranging the elements of a list or array in a specific order, typically either ascending or descending. Sorting makes data more manageable and is crucial for efficient searching.  Common Sorting Algorithms:  **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order.  **Merge Sort:** Divides the list into smaller sub lists and then merges them in order.  **Example:** Sorting a list of student names alphabetically.   * 1. **Keys**   A key is a specific value or attribute in a data structure that uniquely identifies an element. Keys are often used for searching, sorting, and indexing data.  **Example:** In a database, each record might have a unique key, like a student ID, to identify and retrieve student information.   * 1. **Index**   An index refers to the position of an element within a data structure, particularly in arrays or lists. Indexing allows for direct access to an element in constant time (O(1)) if the data structure supports it.  **Example:** In an array, the first element is typically at index 0, the second at index 1, and so on.   1. **Classifications of data structures**    1. **Linear**   **Linear Data Structures:** Data is stored sequentially, and elements are arranged one after the other. Each element is connected to its previous and next element.  **Examples:**  **Array:** A collection of elements stored in contiguous memory locations.  **Linked List:** A series of nodes where each node points to the next one.  **Stack:** A LIFO (Last In, First Out) structure.  **Queue:** A FIFO (First In, First Out) structure.   * 1. **Non-linear**   **Non-Linear Data Structures:** Data elements are stored hierarchically or in a way that they cannot be traversed in a single sequence.  **Examples:**  **Trees:** A hierarchical data structure where each node points to its children.  **Graphs:** A set of nodes connected by edges, representing relationships or connections   1. **Data types**     1. **Built-in**   **Built-in data types** These are the basic data types provided by a programming language. They are predefined and directly supported by the compiler or interpreter.  Common examples include:  **Integer:** Represents whole numbers (e.g., -1, 0, 42).  **Float:** Represents decimal numbers (e.g., 3.14, -0.001).  **Character:** Represents single characters (e.g., 'a', 'Z').  **Boolean:** Represents truth values (e.g., true, false).   * 1. **Derived**   **Derived data types** are created from the built-in data types. They allow for the construction of more complex data structures**.**  **Examples include:**  **Array:** A collection of elements of the same type (e.g., int[]).  **Structure:** A user-defined collection of variables, possibly of different types, grouped together (e.g., struct in C).  **Union:** Similar to structures but can store different data types in the same memory location.  **Function:** A type that represents a function, allowing it to be passed as an argument.   * 1. **User defined**   **User-defined data types** are data types defined by the user to meet specific requirements of a program. They provide flexibility and enhance code readability.  **Examples include:**  **Class:** In object-oriented programming, a class is a blueprint for creating objects (e.g., class Person).  **Enumeration:** A type that consists of a set of named values (e.g., enum Color {RED, GREEN, BLUE}).  **Typedef:** A way to create a new name for an existing type (e.g., typedef int Integer;).   * 1. **Abstract Data Type (ADT)**   **An Abstract Data Type (ADT) is** a mathematical model for data types where the data type is defined by its behavior from the point of view of a user, specifically in terms of possible values and operations, rather than its implementation.  **Examples include:**  **List:** A collection of elements with operations like insert, delete, and search.  **Stack:** A collection that follows the Last In First Out (LIFO) principle, with operations like push and pop.  **Queue:** A collection that follows the First In First Out (FIFO) principle, with operations like enqueue and dequeue.   1. **List representation**   List representation refers to the way lists are organized, stored, and manipulated in computer memory. It defines how the elements of a list are laid out and how operations such as insertion, deletion, and traversal are performed.   1. **Data structure operations**    1. **Traversing**   Traversing means visiting each element of a data structure in order to process it (e.g., printing, counting, or modifying).  **Example** (Array Traversal):  **Algorithm:**  1. Start from the first element (index 0).  2. For each element in the array:  a. Print the element or perform any other required operation.  3. Repeat until the last element is processed.  **Explanation:** The algorithm starts from the first element and processes each element one by one in sequence.   * 1. **Searching**   Searching refers to finding an element in a data structure. The most common types of searching are linear search and binary search.  **Example**  (Linear Search):  **Algorithm:**  1. Start from the first element.  2. Compare each element with the target value.  3. If a match is found, return the index.  4. If no match is found, return -1.( to indicate that the element is not present/found.)  **Explanation:** The linear search algorithm checks each element one by one until the target is found or the search reaches the end of the array.   * 1. **Inserting**   Inserting is adding a new element into a data structure at a specific position.  **Example** (Array Insertion):  **Algorithm:**   1. **Check if the position is valid:**   Ensure that the position where you want to insert the new element is within the valid range of the array (between 0 and the current size of the array).   1. **Shift all elements from the given position to the right:**   Start at the last element and move each element one position to the right, creating space for the new element at the desired position.   1. **Insert the new element at the specified position:**   Place the new element into the correct position in the array.   1. **Increase the size of the array:**   After the insertion, increment the size of the array to reflect the addition of the new element.  **Real life example:**  Imagine you have a row of seats numbered 1, 2, 4, 5, and 6 in a theater, and you realize that seat number 3 is missing. To insert seat 3 in the correct place, you would ask everyone from seat 4 onwards to move one seat to the right. Once they've moved, you can place seat number 3 in its correct position. This is analogous to how array insertion works, where existing elements are shifted to make space for the new element.   * 1. **Deleting**   Deleting removes an element from a data structure, which may involve shifting elements to maintain the structure's order.  **Example** (Array Deletion):  **Algorithm:**   1. **If the position is valid:**   Ensure that the position from which the element is to be deleted is within the bounds of the array (between 0 and the current size of the array).   1. **Shift all elements from the position of deletion to the left:**   After deleting the element, move each subsequent element one position to the left, starting from the position of deletion.   1. **Reduce the size of the array:**   Decrease the size of the array by one to reflect the removal of the element.  **Real life example**  Imagine you have a shelf of five books, and you want to remove the third book. After you remove the third book, you must shift the fourth and fifth books one spot to the left to fill the gap. The bookshelf now contains only four books, and the empty space is at the end of the shelf. This is how deletion works in an array, where the subsequent elements shift to maintain the order.   * 1. **Sorting**   Sorting is the process of arranging elements of a data structure in a specified order (usually ascending or descending).  **Example** (Bubble Sort):  **Algorithm:**  1. Start from the first element.  2. Compare each element with the next element.  3. Swap them if they are in the wrong order.  4. Repeat the process for all elements until no swaps are needed.  **Explanation:** Bubble sort repeatedly swaps adjacent elements that are in the wrong order. The largest elements "bubble" to the top.   * 1. **Merging**   Merging is the process of combining two sorted data structures into one sorted data structure.  **Example** (Merging Two Sorted Arrays):  **Algorithm:**  1. Start at the beginning of both arrays.  2. Compare the current elements of both arrays.  3. Insert the smaller element into the merged array.  4. Repeat the process until all elements are merged. |



**Practical Activity 1.1.2: Performing Data structure operations**

**Task:**

**1:** Read the following task about data structure operations:

As a trainee in CSA, visit the computer lab to perform the following data structure operations using algorithm:

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

**2:** Apply instructions to perform data structure operations.

**3:** Present out the steps to perform data structure operations

**4:** Referring to the presented steps in the task 3, perform data structure operations.

**5:** Present your work to the trainer and whole class. Ask for clarification where necessary

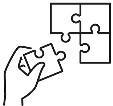
**6**: Read key reading 1.1.2 for more clarifications

**7:** Perform task provided in the application of learning 1.1

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 1.1.2.: Performing Data structure operations**   1. **Traversing** (Visiting all elements)   **Algorithm (Array):**  **Step1: Start from the First Element:** Initialize an index i to 0 (the first element).  **Step2: Process Current Element:** Print or perform an operation on the current element.  **Step3: Move to the Next Element:** Increment i by 1 (i.e., i = i + 1).  **Step4: Repeat:** Continue this process until you reach the last element of the array (i.e., when i == n, where n is the total number of elements).  **Example:**  Imagine you are a teacher taking attendance for a class. You have a list of student names:  ["Alice", "Bob", "Charlie", "David", "Emma"]  You want to go through this list and mark each student as present.  **Step1: Current List:**  ["Alice", "Bob", "Charlie", "David", "Emma"]  **Step2: Start from the First Element:**  Begin with i = 0, pointing to "Alice".  **Step3: Process Each Student:**   * Print "Alice" as present. * Increment i to 1 and move to "Bob". * Print "Bob" as present. * Increment i to 2 and move to "Charlie". * Print "Charlie" as present. * Continue this process until you reach "Emma".   **Step4: Finish:**  Once you print "Emma", you have traversed the entire list.   1. **Searching** (Finding an element)   Algorithm (Searching in Array list):  **Step1:** Start from the First Element: Initialize an index i to 0 (the first element).  **Step2:** Compare Current Element: Check if the current element is equal to the key (the element you are searching for).  **Step3:** If Found: If the current element matches the key, return the index i.  **Step4:** If Not Found: If the current element does not match, increment i by 1 (i.e., move to the next element).  **Step5:** Repeat: Continue this process until the key is found or until you reach the end of the array.  **Example:**  Imagine you are a librarian looking for a specific book in a shelf of books. The books are arranged in a line, and you want to find the book titled "The Great Gatsby."  **Step1:** Current Shelf:  ["1984", "To Kill a Mockingbird", "The Great Gatsby", "Moby Dick", "War and Peace"]  **Step2:** Start from the First Element:  Begin with i = 0, pointing to "1984".  **Step3:** Compare Each Book:  Check "1984" (not a match).  Move to i = 1 (next book): "To Kill a Mockingbird" (not a match).  Move to i = 2: "The Great Gatsby" (match found!).  **Step4:** If Found:  Since you found "The Great Gatsby" at index 2, you note the position and stop searching.  **Step5:** End of Search:  If you had reached the end of the shelf without finding the book, you would conclude that it's not available.   1. **Inserting** (Adding an element)   Algorithm (Insert in Array at Index):  Steps:  **Step1:** Shift Elements: Start from the last element of the array and shift all elements to the right of the specified index to make space for the new element.  **Step2:** Insert New Element: Place the new element at the specified index.  **Step3:** Increase Size: Update the size of the array to reflect the addition of the new element (e.g., n = n + 1).  **Example**  Imagine you are organizing a dinner party and you have a seating arrangement for your guests. Your current seating is as follows:  ["Alice", "Bob", "Charlie", "David"]  You want to add a new guest, **Emma**, to the seating arrangement at index 2  **Step1:** **Current Seating:**  ["Alice", "Bob", "Charlie", "David"]  **Step2:** **Shift Elements:**  To make space for Emma, you need to shift David and any guests after him one position to the right:  ["Alice", "Bob", "Charlie", (empty), "David"]  **Step3:** **Insert New Guest:**  Now, insert Emma at index 2:  ["Alice", "Bob", "Emma", "Charlie", "David"]  **Step4:** **Increase Size:**  The size of the seating arrangement has increased from 4 to 5.   1. **Deleting** (Removing an element)   Algorithm (Delete from Array):  **Step1: Identify the Index:** Determine the index from which the element is to be deleted.  **Step2: Shift Elements:** Starting from **index + 1**, move each element one position to the left, filling the gap created by the deletion.  **Step3: Decrease the Size of the Array:** Reduce the array size by 1 to reflect the removed element.  **Example**  Imagine you are organizing a list of guests for a birthday party. Your current list of attendees is:  ["Alice", "Bob", "Charlie", "David", "Emma"]  Suppose Bob can no longer attend, and you need to remove him from the list.  **Step1: Current List:**  ["Alice", "Bob", "Charlie", "David", "Emma"]  **Step2: Start at the Given Index:**  Bob is located at index 1 (the second position).  **Step3: Shift Elements:**  To maintain the order, you shift Charlie, David, and Emma one position to the left:  ["Alice", "Charlie", "David", "Emma", (empty)]  **Step4: Decrease Size:**  The list now effectively has 4 elements instead of 5.   1. **Sorting** (Arranging elements in a list)   Algorithm (Bubble Sort for Array):  **Step1: Compare Adjacent Elements:** Start from the first element of the array and compare it with the next element.  **Step2: Swap if Necessary:** If the first element is greater than the second (for ascending order), swap them.  **Step3: Repeat:** Continue this process for the entire array, making multiple passes until no swaps are needed, indicating that the array is sorted.  **Example**  Imagine you are a teacher who has a list of students along with their ages. You want to sort the students in ascending order based on their ages.  Initial List of Students:  [("Alice", 23), ("Bob", 19), ("Charlie", 22), ("David", 21), ("Emma", 20)]  **Bubble Sort Process**   1. **First Pass:**  * Compare Alice (23) with Bob (19) → Swap → [("Bob", 19), ("Alice", 23), ("Charlie", 22), ("David", 21), ("Emma", 20)] * Compare Alice (23) with Charlie (22) → Swap → [("Bob", 19), ("Charlie", 22), ("Alice", 23), ("David", 21), ("Emma", 20)] * Compare Alice (23) with David (21) → Swap → [("Bob", 19), ("Charlie", 22), ("David", 21), ("Alice", 23), ("Emma", 20)] * Compare Alice (23) with Emma (20) → Swap → [("Bob", 19), ("Charlie", 22), ("David", 21), ("Emma", 20), ("Alice", 23)] * End of First Pass: The oldest student (Alice) is now at the end.  1. **Second Pass:**  * Compare Bob (19) with Charlie (22) → No swap. * Compare Charlie (22) with David (21) → Swap → [("Bob", 19), ("David", 21), ("Charlie", 22), ("Emma", 20), ("Alice", 23)] * Compare Charlie (22) with Emma (20) → Swap → [("Bob", 19), ("David", 21), ("Emma", 20), ("Charlie", 22), ("Alice", 23)] * End of Second Pass: "Charlie" is now correctly positioned.  1. **Subsequent Passes:**  * Continue comparing and swapping until a pass is made without any swaps.   Final Sorted List of Students by Age:  **[("Bob", 19), ("Emma", 20), ("David", 21), ("Charlie", 22), ("Alice", 23)]**   1. **Merging** (Combining two sorted lists into one)     Algorithm (Merge Two Sorted Arrays):  **Step1:** **Initialize Pointers:** Start with pointers at the beginning of both arrays.  **Step2:** **Compare and Append:** Compare the current elements of both arrays. Append the smaller element to the result array.  **Step3:** **Move Pointer:** Move the pointer of the array from which the element was taken.  **Step4:** **Append Remaining Elements:** Once one array is exhausted, append all remaining elements from the other array to the result.  **Example**  Imagine you have two sorted lists of attendees for two different events:  Event A: [Alice, Charlie, Emma]  Event B: [Bob, David, Frank]  You want to create a single sorted list of attendees for a combined event.  **Merging Process:**  1.Start with Both Lists:  Event A: [Alice, Charlie, Emma]  Event B: [Bob, David, Frank]  Merged List: []  2.Compare Elements:  Compare Alice (Event A) and Bob (Event B).  Alice is smaller, so add Alice to the merged list.  Merged List: ['Alice']  3.Continue:  Now compare Charlie (Event A) and Bob (Event B).  Bob is smaller, so add Bob.  Merged List: ['Alice', 'Bob']  4.Repeat the Process:  Continue comparing and adding:  Merge step-by-step until all elements are added:  Merged List: ['Alice', 'Bob', 'Charlie', 'David', 'Emma', 'Frank'] |

** Points to Remember**

* When describing data structure, take into consideration the following:
* Data refers to raw, unorganized facts or values that can be processed or manipulated by an algorithm or program.
* Structure refers to the way data is organized, stored, and accessed in memory
* Data structures are specialized formats for organizing, storing, and managing data
* A list is a linear data structure that represents a sequence of elements.
* Searching is the process of finding a specific element or a group of elements in a data structure.
* Sorting is the process of arranging the elements of a list or array in a specific order, typically either ascending or descending.
* A key is a specific value or attribute in a data structure that uniquely identifies an element.
* An index refers to the position of an element within a data structure, particularly in arrays or lists
* Linear Data Structures: Data is stored sequentially, and elements are arranged one after the other.
* Non-Linear Data Structures: Data elements are stored hierarchically or in a way that they cannot be traversed in a single sequence.
* Built-in data types These are the basic data types provided by a programming language.
* Derived data types are created from the built-in data types. They allow for the construction of more complex data structures.
* User-defined data types are data types defined by the user to meet specific requirements of a program.
* An Abstract Data Type (ADT) is a mathematical model for data types where the data type is defined by its behaviour from the point of view of a user, specifically in terms of possible values and operations, rather than its implementation.
* List representation refers to the way lists are organized, stored, and manipulated in computer memory.
* When performing data structure operations take into consideration the following:
* Traversing: Prints all elements in the array.
* Searching: Finds the index of an element in the array.
* Inserting: Adds a new element at a specific position.
* Deleting: Removes an element from a specific position.
* Sorting: Uses bubble sort to arrange elements in ascending order.
* Merging: Merges two sorted arrays into one sorted array

**Application of learning 1.1.**

Suppose that your school needs to create a library management system for its library, thus you are asked to perform data structure operations to help school store detailed information about each book, allowing users (library staff and visitors) to Add new books to the library's database, Search for books by their title, Delete books from the system and Sort the collection based on the authors' names.

## Indicative content 1.2 : Description of Algorithms



****

**Duration: 5 hrs**



**Theoretical Activity 1.2.1: Description of algorithms**

**Tasks:**

1: Answer to the following questions related to algorithms:

1. Describe Asymptotic notation
2. Explain Searching algorithms
3. Explain Sorting algorithms
4. Describe Classification of sorting algorithms

2: Write the answers for the asked questions on flipchart/papers.

3: Present the findings/answers to the whole class

4: Ask for clarification where necessary.

5: For more clarification, read the key readings 1.2.1.

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 1.2.1.: Description of algorithms**   1. **Asymptotic Notation**   Asymptotic notation is a mathematical framework used to analyse the efficiency of algorithms, particularly concerning their time and space complexity as the input size grows. It helps in comparing the performance of different algorithms in a standardized way.   * 1. **Role of Asymptotic Notation** * **Efficiency Measurement**: It allows us to describe how the running time or space requirements of an algorithm increase with the size of the input. * **Comparison of Algorithms**: It provides a way to compare the performance of algorithms based on their growth rates rather than exact execution times.   1. **Types of Asymptotic Notation**      1. **Big O Notation (O):**   Represents the upper bound of an algorithm's growth rate.  Used to describe the worst-case scenario.  **Example**: If an algorithm has a time complexity of O(n^2), it means that, in the worst case, the time taken can grow quadratically with the input size.   * + 1. **Omega Notation (Ω):**   Represents the lower bound of an algorithm's growth rate.  Used to describe the best-case scenario.  **Example**: If an algorithm has a time complexity of Ω(n), it means that, in the best case, the time taken will be at least linear with the input size.   * + 1. **Theta Notation (Θ):**   Represents a tight bound on the growth rate, indicating that the algorithm's growth rate is both upper and lower bounded.  **Example**: If an algorithm has a time complexity of Θ(n log n), it means that the time taken will grow proportionally to n log n for large input sizes.   * 1. **Characteristics of asymptotic notation**   It provides a theoretical estimate of performance that is independent of specific machine characteristics or implementations.  **Simplification:** It simplifies the analysis of algorithms by providing a high-level view of their efficiency.  **Conclusion**  Asymptotic notation is crucial in algorithm design and analysis, allowing developers to make informed decisions about which algorithms to use based on their efficiency characteristics. Understanding these notations helps in optimizing algorithms for performance and scalability.   1. **Searching algorithm**   Searching algorithm is a method used to locate a specific element or a group of elements within a data structure. The goal is to efficiently find the desired data while minimizing the number of comparisons or operations needed.  There are various types of searching algorithms, including:   * 1. **Linear Search**   Linear search checks each element in a list one by one until the desired element is found or the list is exhausted.  **Example:**  **Input: List = [5, 3, 8, 4, 2], Target = 4**  **Process:**  Compare 5 (index 0) with 4 (not a match)  Compare 3 (index 1) with 4 (not a match)  Compare 8 (index 2) with 4 (not a match)  Compare 4 (index 3) with 4 (match found)  Output: Index 3  **Time Complexity:** O(n)   * 1. **Binary Search**   Binary search works on a sorted array. It repeatedly divides the search interval in half, checking if the target is equal to the middle element, or adjusting the search range accordingly  **Example:**  Input: Sorted List = [1, 2, 3, 4, 5, 6, 7], Target = 5  **Process:**  Middle element = 4 (index 3). 5 > 4, so search in the right half: [5, 6, 7]  New middle element = 5 (index 4). Match found.  Output: Index 4  Time Complexity: O(log n)   * 1. **Depth-First Search (DFS)**   DFS explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.  **Example: Graph**    **Process:**  Start at A. Visit A.  Move to B. Visit B.  Move to D. Visit D. (no further nodes, backtrack)  Backtrack to B, then to E. Visit E. (no further nodes, backtrack)  Backtrack to A, then to C. Visit C.  **Output**: Order of traversal: A, B, D, E, C  **Time Complexity**: O(V + E)   * 1. **Breadth-First Search (BFS)**   BFS explores all neighbors at the present depth prior to moving on to nodes at the next depth level. It is typically implemented using a queue.  **Example:** **Graph**    **Process:**  Start at A. Visit A.  Visit neighbors of A: B, C.  Next, visit neighbors of B: D, E.  Output: Order of traversal: A, B, C, D, E  **Time Complexity:** O(V + E)   1. **Classification of Sorting Algorithms**   A sorting algorithm is a method used to arrange the elements of a list or array in a specific order, typically in ascending or descending numerical or lexicographical order. Sorting is a fundamental task in computer science, as it helps in organizing data for efficient searching and retrieval.  Sorting algorithms can be classified based on various criteria.   * 1. **Number of Comparisons**   **Comparison-based Algorithms**: Require comparisons to determine the order (e.g., Quick Sort, Merge Sort).  **Non-comparison-based Algorithms:** Do not rely on comparisons (e.g., Counting Sort, Radix Sort).   * 1. **Number of Swaps**   **In-place Algorithms:** Sort the data without requiring additional space (e.g., Bubble Sort, Insertion Sort).  **Out-of-place Algorithms**: Require additional space to hold the sorted data (e.g., Merge Sort).   * 1. **Memory Usage**   **In-place Sorting:** Uses a small, constant amount of extra space (e.g., Quick Sort).  **Non in-place Sorting:** Requires additional memory proportional to the input size (e.g., Merge Sort).   * 1. **Recursion**   **Recursive Algorithms**: Use recursive calls (e.g., Quick Sort, Merge Sort).  **Iterative Algorithms**: Implement sorting without recursion (e.g., Bubble Sort, Insertion Sort).   * 1. **Stability**   **Stable Sorting Algorithms**: Maintain the relative order of equal elements (e.g., Merge Sort, Insertion Sort).  **Unstable Sorting Algorithms:** Do not guarantee the order of equal elements (e.g., Quick Sort, Heap Sort).   * 1. **Adaptability**   **Adaptive Sorting Algorithms**: Perform better on partially sorted data (e.g., Insertion Sort).  **Non-adaptive Sorting Algorithms:** Do not take advantage of existing order (e.g., Quick Sort).   * 1. **Internal Sorting**   **Internal Sorting Algorithms:** Operate on data that fits entirely in memory (e.g., Quick Sort, Heap Sort).   * 1. **External Sorting**   **External Sorting Algorithms**: Designed to handle large datasets that do not fit in memory (e.g., External Merge Sort).  This classification helps in selecting the appropriate sorting algorithm based on the specific requirements of a problem. |



**Practical Activity 1.2.2: Performing Sorting Algorithms**

**Task:**

1: Read the following task about sorting algorithms:

As a trainee in Computer system and architecture, visit the computer lab and perform these activities related to sorting algorithms:

1. Selection sort
2. Bubble sort
3. Insertion sort
4. Merge sort
5. Quick sort
6. Shell sort
7. Heap sort
8. Radix sort
9. Counting sort
10. Bucket sort

2: Apply instructions to perform sorting algorithms.

3: Present out the steps to perform sorting algorithms.

4: Referring to the presented steps in the task 3, perform sorting algorithms.

5: Present your work to the trainer and whole class. Ask for clarification where necessary

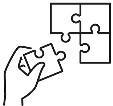
6: Read key reading 1.2.2 for more clarifications

7: Perform task provided in the application of learning 1.2

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 1.2.2.: Performing sorting and searching Algorithms**   1. **Selection Sort**   Selection sort divides the array into a sorted and an unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part.  **Steps:**   1. **Initialize**: Start with an unsorted array. 2. **Outer Loop**: Iterate over each element in the array:  * Set the current index as the minimum index.  1. **Inner Loop**: For each element after the current index:  * Compare it with the current minimum. * Update the minimum index if a smaller element is found.  1. **Swap**: After the inner loop, swap the minimum element with the current index. 2. **Repeat**: Continue until the entire array is sorted.   **Example:**  **Input:** [64, 25, 12, 22, 11]  **Process:**  Find the minimum (11), swap with 64: [11, 25, 12, 22, 64]  Next minimum (12), swap with 25: [11, 12, 25, 22, 64]  Next minimum (22), swap with 25: [11, 12, 22, 25, 64]  Remaining elements are already sorted.  Output: [11, 12, 22, 25, 64**]**  **Time Complexity:** O(n²)   1. **Bubble Sort**   Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.  **Steps:**   1. **Initialize**: Start with an unsorted array. 2. **Outer Loop**: Repeat for the length of the array:  * Set a flag to indicate if a swap occurred.  1. **Inner Loop**: Compare adjacent elements:  * If the first is greater than the second, swap them. * Set the flag to indicate a swap occurred.  1. **Check Flag**: If no swaps occurred during an iteration, the array is sorted, and you can exit early. 2. **Repeat**: Continue until the array is sorted.   **Example:**  **Input: [5, 1, 4, 2, 8]**  **Process:**  Compare 5 and 1, swap: [1, 5, 4, 2, 8]  Compare 5 and 4, swap: [1, 4, 5, 2, 8]  Compare 5 and 2, swap: [1, 4, 2, 5, 8]  Compare 5 and 8, no swap.  Repeat until sorted.  Output: [1, 2, 4, 5, 8]  **Time Complexity:** O(n²)   1. **Insertion Sort**   Insertion sort builds the sorted array one element at a time. It takes an element from the unsorted part and inserts it into the correct position in the sorted part.  **Steps:**   * 1. Initialize: Start with the second element (the first is considered sorted).   2. Outer Loop: Iterate from the second element to the end of the array: * Store the current element as a key. * Inner Loop: Compare the key with elements in the sorted portion.   1. Shift elements to the right until you find the correct position for the key.   2. Insert: Place the key in its correct position.   3. Repeat: Continue until the entire array is sorted.   **Example:**  **Input:** [12, 11, 13, 5, 6]  **Process:**  Start with 12, compare and insert 11: [11, 12, 13, 5, 6]  Insert 13, no change: [11, 12, 13, 5, 6]  Insert 5, move 11, 12, 13: [5, 11, 12, 13, 6]  Insert 6, move 11, 12, 13: [5, 6, 11, 12, 13]  Output: [5, 6, 11, 12, 13]  **Time Complexity:** O(n²)   1. **Merge Sort**   Merge sort is a divide-and-conquer algorithm that divides the list into halves, recursively sorts each half, and merges the sorted halves.  **Steps:**   1. Initialize: Start with an unsorted array. 2. Base Case: If the array has one or no elements, return it. 3. Divide: Split the array into two halves. 4. Recursion: Recursively apply merge sort to both halves. 5. Merge: Combine the two sorted halves:  * Create a temporary array. * Compare elements from both halves and add the smaller one to the temporary array. * Copy any remaining elements.  1. Return: Replace the original array with the merged array.   **Example:**  **Input:** [38, 27, 43, 3, 9, 82, 10]  **Process:**  Split: [38, 27, 43] and [3, 9, 82, 10]  Split further: [38], [27, 43] and [3, 9], [82, 10]  Merge: [27, 38, 43] and [3, 9, 10, 82]  Final merge: [3, 9, 10, 27, 38, 43, 82]  Output: [3, 9, 10, 27, 38, 43, 82]  **Time Complexity**: O(n log n)   1. **Quick Sort**   Quick sort selects a 'pivot' element, partitions the array around the pivot, and recursively sorts the sub-arrays**.**  **Steps:**   1. **Initialize**: Start with an unsorted array. 2. **Base Case**: If the array has one or no elements, return it. 3. **Choose a Pivot**: Select a pivot element (commonly the last element). 4. **Partition**: Rearrange the array so that:  * Elements less than the pivot are on the left. * Elements greater than the pivot are on the right.  1. **Recursion**: Recursively apply quick sort to the left and right sub-arrays. 2. **Return**: Combine the sorted sub-arrays and the pivot   **Example:**  **Input**: [10, 80, 30, 90, 40, 50, 70]  **Process:**  Choose 70 as pivot. Partition: [10, 30, 40, 50] | 70 | [80, 90]  Sort left: Quick sort [10, 30, 40, 50], pivot 30: [10] | 30 | [40, 50]  Sort right: No change for [80, 90**].**  **Output: [10, 30, 40, 50, 70, 80, 90]**  **Time Complexity:** O(n log n) (average), O(n²) (worst)   1. **Shell Sort**   Shell sort generalizes insertion sort to allow the exchange of items that are far apart. It sorts elements at specific intervals**.**  **Steps:**   1. **Initialize**: Start with an unsorted array and choose an initial gap (usually half the length). 2. **Outer Loop**: While the gap is greater than zero:  * Perform a modified insertion sort for elements that are gap apart.  1. **Inner Loop**: Iterate through the array:  * Compare and insert elements that are gap apart.  1. **Reduce Gap**: After completing the inner loop, reduce the gap (e.g., divide by 2). 2. **Repeat**: Continue until the gap is zero, resulting in a sorted array   **Example:**  **Input:** [12, 34, 54, 2, 3]  **Process:**  Start with a gap of 2: Compare 12 with 2, swap: [2, 34, 54, 12, 3]  Next gap 1: Perform an insertion sort: [2, 3, 12, 34, 54]  Output: [2, 3, 12, 34, 54]  **Time Complexity:** O(n log n) (depends on gap sequence)   1. **Heap Sort**   Heap sort uses a binary heap data structure. It builds a max heap from the input data and then repeatedly extracts the maximum element.  **Steps:**   1. **Initialize**: Start with an unsorted array. 2. **Build a Max Heap**: Transform the array into a max heap:  * For each non-leaf node, perform a heapify operation.  1. **Sort**:  * Swap the first element (maximum) with the last element. * Reduce the heap size by one. * Heapify the root of the tree to maintain the heap property.  1. **Repeat**: Continue until the heap size is reduced to one. 2. **Return**: The array is sorted.   **Example:**  **Input:** [3, 6, 5, 0, 8, 2, 7, 4, 1]  **Process:**  Build max heap: [8, 6, 7, 4, 3, 5, 2, 0, 1]  Extract 8, rebuild heap: [7, 6, 5, 4, 3, 2, 1, 0]  Continue extracting to get sorted array.  Output: [0, 1, 2, 3, 4, 5, 6, 7, 8]  **Time Complexity:** O(n log n)   1. **Radix Sort**   Radix sort sorts integers by processing individual digits. It groups numbers based on each digit from least significant to most significant.  **Steps:**   1. **Initialize**: Start with an array of non-negative integers. 2. **Find Maximum**: Determine the maximum number to know the number of digits 3. **Iterate Over Digits**: For each digit (starting from the least significant):  * Use a stable sorting algorithm (e.g., Counting Sort) to sort the array based on the current digit.  1. **Repeat**: Continue until all digits have been processed. 2. **Return**: The array is sorted   **Example:**  **Input**: [170, 45, 75, 90, 802, 24, 2, 66]  **Process:**  Sort by least significant digit: [170, 90, 2, 802, 24, 45, 75, 66]  Sort by next digit: [2, 24, 45, 66, 75, 90, 170, 802]  Output: [2, 24, 45, 66, 75, 90, 170, 802]  **Time Complexity**: O(nk) (where k is the number of digits)   1. **Counting Sort**   Counting sort counts the occurrences of each unique element and calculates the position of each element in the sorted array.  **Steps:**   1. **Initialize**: Start with an array of non-negative integers. 2. **Find Range**: Determine the maximum value in the array. 3. **Create Count Array**: Initialize a count array of size equal to the maximum value plus one. 4. **Count Occurrences**: Iterate through the input array and count occurrences of each value, storing them in the count array. 5. **Cumulative Count**: Transform the count array to hold the cumulative counts. 6. **Build Output Array**: Iterate through the input array in reverse order to build the output array using the count array. 7. **Return**: The output array is sorted.   **Example:**  Input: [4, 2, 2, 8, 3, 3, 1]  Process:  Count occurrences: [1, 1, 2, 2, 0, 0, 1]  Calculate positions and build sorted array: [1, 2, 2, 3, 3, 4, 8]  Output: [1, 2, 2, 3, 3, 4, 8]  **Time Complexity**: O(n + k)   1. **Bucket Sort**   Bucket sort divides the input into several buckets, sorts each bucket individually, and concatenates the sorted buckets.  **Steps:**   1. **Initialize**: Start with an array of floating-point numbers. 2. **Create Buckets**: Create a fixed number of empty buckets (sub-arrays). 3. **Distribute Elements**: Iterate through the original array and distribute the elements into the appropriate buckets based on their value. 4. **Sort Each Bucket**: Sort each non-empty bucket individually (using another sorting algorithm, like Insertion Sort). 5. **Concatenate**: Merge all the sorted buckets into a single output array. 6. **Return**: The array is sorted   **Example:**  **Input:** [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.55]  **Process:**  Place numbers in buckets based on their range: Bucket 0: [0.17, 0.21], Bucket 1: [0.39, 0.55, 0.72, 0.78], Bucket 2: [0.94]  Sort each bucket: [0.17, 0.21], [0.39, 0.55, 0.72, 0.78], [0.94]  Concatenate: [0.17, 0.21, 0.39, 0.55, 0.72, 0.78, 0.94]  Output: [0.17, 0.21, 0.39, 0.55, 0.72, 0.78, 0.94]  **Time Complexity**: O(n + k) (where k is the number of buckets) |

** Points to Remember**

* When describing algorithms, take into consideration the following:
* Asymptotic notation is crucial in algorithm design and analysis, allowing developers to make informed decisions about which algorithms to use based on their efficiency characteristics.
* Searching algorithm is a method used to locate a specific element or a group of elements within a data structure.
* A sorting algorithm is a method used to arrange the elements of a list or array in a specific order
* Sorting algorithms can be classified according to the : Number of comparisons, number of swaps, memory usage, recursion, stability, adaptability, internal sorting, external sorting
* While performing sorting algorithm, Remember the following :
* Selection sort divides the array into a sorted and an unsorted part
* Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
* Insertion sort builds the sorted array one element at a time. It takes an element from the unsorted part and inserts it into the correct position in the sorted part.
* Merge sort is a divide-and-conquer algorithm that divides the list into halves, recursively sorts each half, and merges the sorted halves.
* Quick sort selects a 'pivot' element, partitions the array around the pivot, and recursively sorts the sub-arrays.
* Shell sort generalizes insertion sort to allow the exchange of items that are far apart. It sorts elements at specific intervals.
* Heap sort uses a binary heap data structure. It builds a max heap from the input data and then repeatedly extracts the maximum element.
* Radix sort sorts integers by processing individual digits. It groups numbers based on each digit from least significant to most significant.
* Counting sort counts the occurrences of each unique element and calculates the position of each element in the sorted array.

 **Application of learning 1.2.**

Suppose that your school wants to build an inventory management system that helps the school to develop a retail chain for their stores. The store relies on this system to keep track of thousands of products, ensuring efficient management of stock, prices, and categories. The system will allow store managers and employees to perform essential tasks, such as searching for products by their name and viewing products sorted by price. Go in the computer lab and build the system in the question.

## Indicative content 1.3: Analysing Algorithmic Concepts



****

**Duration: 4 hrs**



**Theoretical Activity 1.3.1: Description of algorithmic concepts**

**Tasks:**

1: Answer to the following questions related to algorithm:

i. Describe the following algorithmic concepts:

1. Algorithmic cases
2. complexity

2: Provide your answers on flipcharts/papers.

3: Present the findings/answers to the whole class

4: Ask for clarification where necessary

5: Read the key readings 1.3.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 1.3.1.: Description of algorithmic concepts**   1. **Algorithmic Cases**   **Algorithmic Cases** refer to different performance scenarios for an algorithm, depending on the nature of the input. These cases help us understand how efficient an algorithm is under various conditions.  The most common algorithmic cases are:   * 1. **Worst-case**   The worst-case scenario refers to the maximum time or space an algorithm will take to complete for any input of size 𝑛. It occurs when the algorithm has to perform the most number of operations.  Critical for assessing the maximum time or space the algorithm could require. For instance, in a sorting algorithm like Quick Sort, the worst case occurs when the pivot is the smallest or largest element, leading to unbalanced partitions.  **Importance:** Worst-case analysis provides an upper bound on the running time, ensuring that even in the most challenging situations, the algorithm won’t exceed a certain time complexity.  **Example**  **Linear Search:** If we search for an element in an unsorted array and the element is either at the last position or not present at all, we need to check all elements. The worst-case time complexity is O(n).  **Quick Sort:** If the pivot selection is consistently poor, such as always choosing the smallest or largest element as the pivot, the worst-case time complexity becomes O(n2).   * 1. **Average-case**   The average-case scenario represents the expected time or space an algorithm will take over all possible inputs of size 𝑛.This analysis considers the probability of different inputs occurring.  Provides a more realistic measure of an algorithm's efficiency in practical situations. For example, the average case for a linear search would involve looking through half the elements on average.  **Importance:** Average-case analysis provides a realistic view of the algorithm’s behavior in everyday use, assuming inputs are random or follow a typical distribution.  **Example:**  **Linear Search:** If the target element is equally likely to be anywhere in the array, on average, we would search through half of the elements. The average-case time complexity is still (𝑛).  **Quick Sort:** With good pivot selections, the average-case time complexity of quicksort is  (𝑛 log 𝑛) even though its worst-case complexity is O(n2).   * 1. **Best-case**   The best-case scenario refers to the minimum time or space an algorithm will take for any input of size 𝑛  This represents the most favorable input where the algorithm finishes with minimal effort.  Useful for understanding the efficiency of an algorithm under ideal conditions. For example, in a search algorithm, the best case occurs when the target element is the first one checked.  **Importance:** Best-case analysis shows how well an algorithm can perform in ideal situations, but it is often not the most practical case since real-world inputs are rarely ideal.  **Example**:  **Linear Search:** If the target element is the very first element in the array, the algorithm stops immediately, resulting in a best-case time complexity of O(1).  **Bubble Sort:** If the array is already sorted, bubble sort only needs to make one pass without swapping any elements. The best-case time complexity is O(n).   1. **Describe Algorithm Complexity**   Complexity in algorithms refers to the resources required (time and space) to execute an algorithm as a function of the input size, usually denoted as n.  The two primary types of complexity are time complexity and space complexity, which measure the algorithm’s efficiency in terms of execution time and memory usage.   * 1. **Time Complexity:**   **Time complexity** is a computational complexity that describes the amount of time an algorithm takes to complete as a function of the length of the input.  It is a measure of the amount of time an algorithm takes to complete as a function of the input size n. It focuses on how the running time of the algorithm scales with the size of the input.  **Importance:** Time complexity helps predict the algorithm's performance and efficiency, particularly as the input size grows. It's a crucial factor when working with large datasets.  **Common Notations:**  O(1): Constant time—algorithm takes the same time regardless of input size.  O(logn): Logarithmic time—e.g., binary search.  O(n): Linear time—e.g., linear search.  O(nlogn): Linearithmic time—e.g., merge sort, quicksort (average case).  O(n2): Quadratic time—e.g., bubble sort, selection sort.  Time complexity is crucial for evaluating algorithms. For instance, an algorithm with O(n log n) time complexity (like MergeSort) is generally more efficient than one with O(n^2) (like Bubble Sort) for large inputs.  **Example:**  Binary Search: Has a time complexity of O(logn) because the problem size is halved with each iteration.  Bubble Sort: Has a time complexity of O(n2) because it requires comparing and swapping all elements in a nested loop.   * 1. **Space Complexity:**   **Space complexity** is a computational complexity that describes the amount of memory space an algorithm uses relative to the input size.  Space complexity refers to the amount of memory an algorithm uses in relation to the size of the input n. This includes both the memory needed for input storage and the extra memory required for the algorithm's internal operations.  **Importance:** Space complexity is crucial in environments with limited memory, as it ensures the algorithm does not consume excessive memory, particularly when scaling to large datasets.   * + 1. **Types of Space Complexity:** * **Auxiliary Space:** The extra space or temporary space used by the algorithm, excluding the input. * **Total Space:** The total memory used, including both the input data and the auxiliary space.   **Example:**  **Merge Sort:** Requires extra space for merging arrays, resulting in a space complexity of O(n) due to the auxiliary array needed for sorting.  **Quick Sort:** Has a space complexity of O(log n) due to recursive calls, but it doesn’t require extra space for a new array.   * 1. **Test Time and Space Complexity:**   Test time and space complexity refer to analyzing how much time and space are required while testing an algorithm during the development phase. This is especially important for understanding practical performance under real-world constraints.  **Importance:** When algorithms are tested on real data, it provides a practical sense of how time and space complexity behave under various conditions. This may vary from theoretical complexity due to hardware factors, caching, or memory management.   * + 1. **Test Time Complexity**   Measures the practical time taken by an algorithm to run test cases of various input sizes. It provides a more realistic view of performance when combined with empirical data (e.g., by profiling).   * + 1. **Test Space Complexity**   Measures the actual memory usage during test runs. While theoretical complexity gives an idea, testing helps identify any unexpected memory consumption (e.g., excessive stack usage in recursive algorithms).  **Example:**   * **Profiling a Sorting Algorithm:** When testing a quicksort algorithm, the actual runtime and memory usage may differ from theoretical values due to system load, memory hierarchy, or how the data is structured. * **Stress Testing with Large Data Sets:** When testing an algorithm with large inputs, the test space complexity helps identify whether the algorithm stays within memory limits and behaves as expected under pressure. |



**Practical Activity 1.3.2: Analysing algorithmic concepts**

**Task:**

1: Read the following task about algorithm:

As a trainee in Computer system and architecture, go in the computer lab and use any type of searching algorithm (example: Binary Search) to apply these Key Algorithmic Concepts:

* 1. Algorithmic Cases
  2. Complexity

2: Apply instructions to perform binary search.

3: Present out the steps to apply binary search.

4: Referring to the presented steps in the task 3, apply binary search to analyse algorithmic concepts

5: Present your work to the trainer and whole class. Ask for clarification where necessary

6: Read key reading 1.3.2 for more clarifications

7: Perform task provided in the application of learning 1.3

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 1.3.2.: Analysing algorithmic concepts**  In this activity, we will practically explore the different algorithmic cases (Worst-case, Average-case, Best-case) using Binary Search. Binary Search is an efficient algorithm used to find the position of an element in a sorted array. It works by dividing the search space in half each time, which results in faster search times compared to linear search.  **1. Worst-case Scenario**  The worst-case scenario happens when the target element is not present in the array. The algorithm has to search the entire search space.  **Example:**  Array: [10, 20, 30, 40, 50, 60, 70]  Target: 75 (not present in the array)  **Steps:**  Compare 75 with the middle element (40), search the right half.  Compare 75 with 60, search the right half.  Compare 75 with 70, search the right half.  Since no elements remain, return -1.  Time Complexity: O(log n)  **2. Average-case Scenario**  The average-case scenario happens when the target element is present somewhere in the middle of the array. On average, the algorithm searches about half of the array.  **Example:**  Array: [10, 20, 30, 40, 50, 60, 70]  Target: 40 (middle element)  **Steps:**  Compare 40 with the middle element (40), found at index 3.  Time Complexity: O(log n)  **3. Best-case Scenario**  The best-case scenario happens when the target element is the middle element of the array. The algorithm only needs one comparison.  **Example:**  Array: [10, 20, 30, 40, 50, 60, 70]  Target: 40 (middle element)  **Steps:**  Compare 40 with the middle element (40), found at index 3 on the first try.  Time Complexity: O(1) |

** Points to Remember**

* While describing algorithmic concepts, take into consideration the following:
* Algorithmic cases refer to different performance scenarios for an algorithm, depending on the nature of the input.
* The worst-case scenario refers to the maximum time or space an algorithm will take to

complete for any input of size 𝑛.

* The average-case scenario represents the expected time or space an algorithm will take over all possible inputs of size 𝑛.
* The best-case scenario refers to the minimum time or space an algorithm will take for any input of size 𝑛.
* Complexity in algorithms refers to the resources required (time and space) to execute an algorithm as a function of the input size, usually denoted as n.
* Time complexity is a computational complexity that describes the amount of

time an algorithm takes to complete as a function of the length of the input.

* Space complexity is a computational complexity that describes the amount of

memory space an algorithm uses relative to the input size.

* Test time and space complexity refer to analysing how much time and space are

required while testing an algorithm during the development phase.

* Test Time Complexity: Measures the practical time taken by an algorithm to run

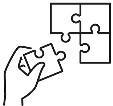
test cases of various input sizes.

* Test Space Complexity: Measures the actual memory usage during test runs.
* While analysing algorithmic concepts, remember to apply the following:
* The worst-case scenario happens when the target element is not present in the

array.

* The average-case scenario happens when the target element is present somewhere in the middle of the array.
* The best-case scenario happens when the target element is the middle element

of the array.

 **Application of learning 1.3.**

Visit the school’s computer lab and design The library's catalog which will be used by students and staff to find books quickly. The catalog contains thousands of book titles that are already sorted alphabetically. As part of your task, you need to ensure that the search function in the system works efficiently by analyzing different search scenarios. You must determine the behavior of Best-case scenario, Average-case scenario and Worst-case scenario.

## Indicative content 1.4: Writing Algorithms



****

**Duration: 3 hrs**



**Theoretical Activity 1.4.1: Explanation about writing algorithms**

**Tasks:**

1: Answer the following questions related to writing algorithms:

What do you understand by the following?:

1. Develop data structures
2. Perform sorting operations
3. Perform searching operations

2: Write the answers for the asked questions on flipchart/papers.

3: Present the findings/answers to the whole class

4: Ask for clarification where necessary.

5: Read the key readings 1.4.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 1.4.1.: Explanation about writing algorithms**   1. **Develop Data structures**   When writing algorithms, it's crucial to design and represent the data structures effectively. This can be done through structured English or structured pseudocode.   * 1. **Using Structured English**   Structured English is a simplified way of writing algorithms in plain, human-readable language. It focuses on clarity, using simple statements to describe the steps of an algorithm without complex syntax.  Here, data structures are described in a very basic and intuitive manner, helping beginners understand the logic without worrying about programming details.   * 1. **Using structured pseudocodes**   Pseudocode is a more formalized way of representing an algorithm, which mimics code but isn't tied to any specific programming language. It uses a structured approach to define data structures and their operations in a way that can be easily converted into actual code.  **Comparing Structured English and Structured Pseudocode**  **Structured English:**   * Focuses on explaining the algorithm in plain language. * Best for conceptual understanding and communication. * Does not use programming syntax, making it easier for non-technical people to understand.   **Structured Pseudocode:**   * More formal, mimics real code but in a generalized way. * Helps in understanding the exact steps needed for programming. * Easier to translate into actual code in any programming language.   **Conclusion:**  **Developing data structures using structured English** is ideal for beginners, non-programmers, or during the planning stage where you need a clear explanation without syntax.  **Developing data structures using structured pseudocode** is more suited for programmers who are designing algorithms and preparing to implement them in code. It bridges the gap between conceptual design and actual coding.   1. **Perform sorting operations**   **Sorting operations** are fundamental tasks in computer algorithms, where the goal is to arrange the elements of a list or array in a particular order (usually ascending or descending). There are several sorting algorithms, each with varying time complexities and use cases. Here, we'll explore a few common sorting algorithms.   * 1. **Bubble Sort**   **Bubble Sort** is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.  **How It Works:** The largest element "bubbles up" to the top in each pass. The process repeats until no more swaps are required.  **Example:**  Given an array [5, 1, 4, 2], Bubble Sort works as follows:  First pass: [1, 4, 2, 5]  Second pass: [1, 2, 4, 5]  **Time Complexity:**  Best Case: O(n) (when the list is already sorted)  Average Case: O(n²)  Worst Case: O(n²)   * 1. **Insertion Sort**   **Insertion Sort** builds the sorted array one item at a time by inserting elements into their correct position relative to the already sorted part of the array.  How It Works: It starts with the second element, compares it to elements in the sorted part of the array (initially just the first element), and places it in the correct position.  **Example:**  Given an array [3, 1, 4, 2], Insertion Sort works as follows:  First pass: [1, 3, 4, 2]  Second pass: [1, 3, 4, 2] (no changes)  Third pass: [1, 2, 3, 4]  **Time Complexity:**  Best Case: O(n) (when the list is already sorted)  Average Case: O(n²)  Worst Case: O(n²)   * 1. **Selection Sort**   **Selection Sort** repeatedly finds the minimum element from the unsorted part of the list and swaps it with the first unsorted element.  **How It Works:** It starts by finding the smallest element and placing it at the beginning of the list, then continues with the rest of the list.  **Example:**  Given an array [4, 2, 7, 1], Selection Sort works as follows:  First pass: [1, 2, 7, 4]  Second pass: [1, 2, 7, 4] (no changes)  Third pass: [1, 2, 4, 7]  **Time Complexity:**  Best Case: O(n²)  Average Case: O(n²)  Worst Case: O(n²)   * 1. **Merge Sort**   **Merge Sort** is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges the two sorted halves together.  **How It Works:** The array is split in half until each piece has only one element, then merged back together in sorted order.  **Example:**  Given an array [4, 1, 3, 2], Merge Sort works as follows:  Split into [4, 1] and [3, 2]  Recursively sort to [1, 4] and [2, 3]  Merge to [1, 2, 3, 4]  **Time Complexity:**  Best Case: O(n log n)  Average Case: O(n log n)  Worst Case: O(n log n)   * 1. **Quick Sort**   **Quick Sort** is another divide-and-conquer algorithm. It picks a "pivot" element and partitions the array so that elements less than the pivot come before it and elements greater than the pivot come after it. It then recursively sorts the sub-arrays.  **How It Works:** The pivot is chosen (usually the last element), and the array is partitioned so that elements smaller than the pivot are on the left, and larger elements are on the right.  **Example:**  Given an array [3, 6, 1, 5], Quick Sort works as follows:  Partition around pivot 5 → [3, 1] | 5 | [6]  Recursively sort [3, 1] to [1, 3]  **Time Complexity:**  Best Case: O(n log n)  Average Case: O(n log n)  Worst Case: O(n²)   1. **Perform searching operations**   When writing algorithms, performing searching operations is a fundamental task that involves finding a specific element or set of elements within a data structure. Here are the key points related to searching operations:  **Types of Searching Algorithms**   * 1. **Linear Search:**   This is the simplest searching algorithm. It checks each element in a list one by one until it finds the target element or reaches the end of the list.  **Time Complexity:** O(n), where n is the number of elements in the list.  **Use Case:** Best for small or unsorted data sets.   * 1. **Binary Search:**   This algorithm is more efficient than linear search and works on sorted lists. It repeatedly divides the search interval in half, comparing the target value to the middle element.  **Time Complexity:** O(log n).  **Use Case:** Suitable for large, sorted arrays or lists. |



**Practical Activity 1.4.2: Writing algorithms**

**Task:**

1: Introduce the activity and ask trainees to read the following task:

As a trainee in Computer system and architecture, go in the computer lab and perform the following activities: Develop Data structures, perform sorting operations and perform searching operations

2: Apply instructions to write algorithm.

3: Present out the steps to write algorithm.

4: Referring to the presented steps in the task 3, write algorithm using pseudocode and structured English

5: Present your work to the trainer and whole class and ask for clarification where necessary

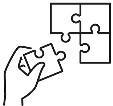
6: Read key reading 1.4.2 for more clarifications

7: Perform task provided in the application of learning 1.4

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 1.4.2.: Writing algorithms**   1. **Develop Data Structures Using Structured English**   **Task 1: Insert an Element into an Array**  **Structured English:**   1. Start with a list of items (array). 2. Check if the array has space for a new item. 3. If yes, insert the new item into the next available position. 4. If no, print "Array is full, cannot insert." 5. End.   **Task 2: Delete an Element from an Array**  **Structured English:**   1. Start with a list of items. 2. Find the item you want to remove. 3. If the item is found, remove it by shifting all items after it one position to the left. 4. If the item is not found, print "Item not found." 5. End.   **Task 3: Search for an Element in an Array**  **Structured English:**   1. Start with a list of items. 2. Loop through each item in the array. 3. If you find the item you’re searching for, return its position. 4. If you finish the loop without finding the item, print "Item not found." 5. End.   **Task 4: Traverse an Array (Visit Each Element)**  **Structured English:**   1. Start with a list of items. 2. Visit each item in the list, one by one, starting from the first. 3. For each item, print its value. 4. End. 5. **Perform sorting operations**   Let's perform a sorting operation using Bubble Sort in a practical example with a specific array of numbers.  arr = [64, 34, 25, 12, 22, 11, 90]  **Step1.** **Initialize the Array:**  arr = [64, 34, 25, 12, 22, 11, 90]  **Step2. Perform Bubble Sort:**  **Pass 1:**  Compare 64 and 34: Swap → [34, 64, 25, 12, 22, 11, 90]  Compare 64 and 25: Swap → [34, 25, 64, 12, 22, 11, 90]  Compare 64 and 12: Swap → [34, 25, 12, 64, 22, 11, 90]  Compare 64 and 22: Swap → [34, 25, 12, 22, 64, 11, 90]  Compare 64 and 11: Swap → [34, 25, 12, 22, 11, 64, 90]  Compare 64 and 90: No swap needed → [34, 25, 12, 22, 11, 64, 90]  **Pass 2:**  Compare 34 and 25: Swap → [25, 34, 12, 22, 11, 64, 90]  Compare 34 and 12: Swap → [25, 12, 34, 22, 11, 64, 90]  Compare 34 and 22: Swap → [25, 12, 22, 34, 11, 64, 90]  Compare 34 and 11: Swap → [25, 12, 22, 11, 34, 64, 90]  Compare 34 and 64: No swap needed → [25, 12, 22, 11, 34, 64, 90]  **Pass 3:**  Compare 25 and 12: Swap → [12, 25, 22, 11, 34, 64, 90]  Compare 25 and 22: Swap → [12, 22, 25, 11, 34, 64, 90]  Compare 25 and 11: Swap → [12, 22, 11, 25, 34, 64, 90]  Compare 25 and 34: No swap needed → [12, 22, 11, 25, 34, 64, 90]  **Pass 4:**  Compare 12 and 22: No swap needed → [12, 22, 11, 25, 34, 64, 90]  Compare 22 and 11: Swap → [12, 11, 22, 25, 34, 64, 90]  Compare 22 and 25: No swap needed → [12, 11, 22, 25, 34, 64, 90]  **Pass 5:**  Compare 12 and 11: Swap → [11, 12, 22, 25, 34, 64, 90]  Compare 12 and 22: No swap needed → [11, 12, 22, 25, 34, 64, 90]  Pass 6: No further swaps needed as the array is sorted.  **Step3. Final Sorted Array:**  **arr = [11, 12, 22, 25, 34, 64, 90]**   1. **Perform searching operations**   Let's perform a searching operation practically using Linear Search, which is a straightforward searching algorithm. We’ll use an example array and search for a specific element.  **Task:** Search for an Element in an Array Using Linear Search  **Example Array:**  **arr = [34, 7, 23, 32, 5, 62, 10]**  Target Element to Search:  target = 23  **Steps of Linear Search:**   1. **Initialize the Array and Target:**   arr = [34, 7, 23, 32, 5, 62, 10]  target = 23   1. **Perform Linear Search:**   Iteration 1:  Compare arr[0] (34) with 23 → No match.  Iteration 2:  Compare arr[1] (7) with 23 → No match.  Iteration 3:  Compare arr[2] (23) with 23 → Match found!  Return index 2.   1. **Result:**   Target found at index: 2 |

** Points to Remember**

* While explaining the ways of writing algorithm, remember the following:
* Developing data structures refers to the process of creating and implementing efficient ways to store, organize, and manage data in a program.
* Sorting is the process of arranging data in a specific order, usually in ascending or descending order.
* Searching refers to finding an element within a collection of data.
* Structured English is a simplified way of writing algorithms in plain, human-readable language. It focuses on clarity, using simple statements to describe the steps of an algorithm without complex syntax.
* Pseudocode is a more formalized way of representing an algorithm, which mimics code but isn't tied to any specific programming language. It uses a structured approach to define data structures and their operations in a way that can be easily converted into actual code.
* Sorting operations are fundamental tasks in computer algorithms, where the goal is to arrange the elements of a list or array in a particular order (usually ascending or descending).
* **While Writing algorithms take into consideration the following:**
* **Develop Data Structures (example: Using Structured English):**
* Insert into Array: Check for space, insert if available, otherwise, show an error.
* Delete from Array: Find the element, shift others left after deletion, or show an error if not found.
* Search in Array: Loop through the array and return the position if found, otherwise show "not found."
* Traverse Array: Visit and display each element sequentially.
* **Perform Sorting (Bubble Sort):**
* Repeatedly compare and swap adjacent elements if needed, until the array is sorted.
* After multiple passes, the largest unsorted elements move to their correct positions.
* **Perform Searching (Linear Search):**
* Check each element one by one against the target.
* Return the index if found, otherwise indicate that the element is not present.

 **Application of learning 1.4.**

Visit your school computer lab and implement an efficient search feature for the library’s book catalog. The books in the catalog are sorted alphabetically by title, and the library staff and visitors need to find books quickly using the digital system. To simulate this in a real-world scenario, consider the following list of book titles already sorted alphabetically:

Library Catalog: ["Algorithms", "Artificial Intelligence", "Computer Networks", "Data Structures", "Database Systems", "Machine Learning", "Operating Systems", "Software Engineering"]

You are tasked with using Binary Search algorithm to find the book titled "Data Structures" in the given sorted catalog.

## Indicative content 1.5: Preparation of C Programming Environment



****

**Duration: 3hrs**



**Theoretical Activity 1.5.1: Description of C programming environment preparation**

**Tasks:**

1: Answer the following questions related to C programming environment preparation.

i. What do you understand by the following:

1. Tools installation
2. Test development environment

2: Write your answers on flipcharts/papers.

3: Present the findings/answers to the whole class

4: Ask questions where necessary.

5: Read the key readings 1.5.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 1.5.1.: Description of C programming environment preparation**   * 1. **Introduction**   **Programming Environment:** “A Programming Environment is the collection of tools used in the development of software.”  **C programming** is a general-purpose programming language developed in the early 1970s by Dennis Ritchie at Bell Labs. It is a low-level language commonly used for system programming, such as operating systems, device drivers, embedded systems, and other applications requiring high performance and efficient memory management.  When starting with C programming, you need to prepare the development environment. This includes installing necessary tools such as an Integrated Development Environment (IDE), a C compiler, and setting up environment variables. Here's a description of the steps involved in preparing the environment:   1. **Tools installation**    1. **IDE Installation**   **An IDE (Integrated Development Environment)** simplifies the process of writing, testing, and debugging C programs. Some popular IDEs for C programming are:   * Code: Blocks: A free, open-source IDE that supports multiple compilers. * Eclipse CDT: A powerful IDE with a rich plugin ecosystem. * Visual Studio: A feature-rich IDE, especially for Windows users. * Dev-C++: A lightweight IDE ideal for beginners.   1. **Compiler Installation**   **A C compiler** translates the C code into machine-readable code that can be executed by the computer. Common compilers include:   * GCC (GNU Compiler Collection): A widely used open-source compiler. * Clang: A compiler based on LLVM, known for its speed and diagnostics. * Microsoft Visual C++: Part of Visual Studio, tailored for Windows development.   1. **Setup Environment Variable Path**   Setting up the environment variable path allows the operating system to recognize and use the C compiler from any directory in the terminal or command prompt.   1. **Test development environment**   Once you have set up your C programming environment by installing the necessary tools (IDE and compiler), the next step is to test the environment by executing a default program and writing sample programs for common case scenarios. This ensures that everything is working properly.   * 1. **Execute Default Program**   To verify that your C development environment is working, execute a simple "Hello, World!" program.   * 1. **Write Samples for Case Scenarios (Using C Programming)**   Now that the environment is functional, you can write and test more advanced case scenarios. Below are sample C programs for basic operations such as arithmetic calculations, conditional statements, and loops. |



**Practical Activity 1.5.2: Preparing C programming environment**

**Task:**

**1:** Read the following task about preparing c programming environment and perform the following:

As a trainee in Computer system and architecture, go in the computer lab and perform the following task: Tools installation and Test development environment

**2:** Apply instructions to prepare C programming environment

**3:** Present out the steps to prepare C programming environment

**4:** Referring to the presented steps in the task 3, install tools and test development environment

**5:** Present your work to the trainer and whole class and Ask for clarification where necessary

**6:** Read key reading 1.5.2 for more clarifications

**7:** Perform task provided in the application of learning 1.5

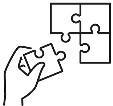
|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 1.5.2.: Preparing C programming environment**   1. **Tools installation**    1. **IDE Installation**   **Installation Steps:**  • Download the IDE installer from the official website.  • Run the installer and follow the prompts.  • Choose the components you wish to install (if applicable).  • Complete the installation and launch the IDE.   * 1. **Compiler Installation**   **Installation Steps:**  1. Download the compiler suitable for your OS.  For GCC, you can use MinGW (Windows) or install via package managers like apt (Linux) or Homebrew (macOS).  2. Follow the installation instructions specific to the compiler.  3. Verify the installation by running a command in the terminal/command prompt (e.g., gcc --version for GCC).  **1.3. Setup Environment Variable Path**  **Installation Steps on Windows:**   1. Right-click on "This PC" or "My Computer" and select "Properties". 2. Click on "Advanced system settings". 3. In the System Properties window, click on "Environment Variables". 4. In the "System variables" section, find and select the "Path" variable, then click "Edit". 5. Add the path to your compiler’s bin directory (e.g., C:\MinGW\bin) and the IDE's installation directory if needed. 6. Click "OK" to save changes.   **2.** **Test development environment**  **2.1. Execute Default Program**  **Steps to Execute a Default Program in C:**  1. Open your IDE (e.g., Code:: Blocks or Visual Studio).  2. Create a new project (select the C language).  3. In the main source file (usually named main.c), write the following code:  #include <stdio.h>  int main() {  // Output to console  printf("Hello, World!\n");  return 0;  }  4. Compile the program (in most IDEs, you can press F9 or select Build and Run from the menu).  5. If the environment is correctly set up, the console will display:  Hello, World!  This confirms that your environment is working properly.   * 1. **Write Samples for Case Scenarios (Using C Programming)**   **Sample : Arithmetic Calculations**  This program demonstrates simple arithmetic operations like addition, subtraction, multiplication, and division.  #include <stdio.h>  int main() {  int a = 10, b = 5;  // Arithmetic operations  printf("Addition: %d + %d = %d\n", a, b, a + b);  printf("Subtraction: %d - %d = %d\n", a, b, a - b);  printf("Multiplication: %d \* %d = %d\n", a, b, a \* b);  printf("Division: %d / %d = %d\n", a, b, a / b);  return 0;  }  **Output:**  Addition: 10 + 5 = 15  Subtraction: 10 - 5 = 5  Multiplication: 10 \* 5 = 50  Division: 10 / 5 = 2 |

** Points to Remember**

* While describing C programming environment, take this into consideration:
* Tools Installation refers to the process of setting up software applications or
* development tools on a computer or server.
* Test Development Environment refers to a controlled setting where software
* applications can be developed, tested, and debugged.
* A Programming Environment is the collection of tools used in the development of

software.

* C programming is a low-level language commonly used for system programming
* An IDE (Integrated Development Environment) simplifies the process of writing, testing, and debugging C programs.
* A C compiler translates the C code into machine-readable code that can be executed by the computer.
* Setting up the environment variable path allows the operating system to recognize and use the C compiler from any directory in the terminal or command prompt.
* After setting up your C programming environment with an IDE and compiler, the next step is to test it.
* While preparing C programming environment, remember the following:
* IDE Installation: Download the IDE, follow installation prompts, choose components, and launch the IDE.
* C Compiler Installation: Download a suitable compiler, follow instructions, and verify installation via terminal (e.g., gcc --version).
* Set Environment Path (Windows): Edit system "Path" to include the compiler’s bin directory via system properties.
* Run C Program: Open IDE, create a project, write code, compile, and view the output to confirm the environment is correctly set up.

 **Application of learning 1.5.**

Visit your school’s computer lab and Set up the development environment on your machine and ensure that the C compiler is installed correctly. Once everything is set up, you need to write a simple C program that prints "Welcome home!" to the console to confirm that your environment is functioning properly

.

## Learning outcome 1 end assessment



**Written assessment**

1. **Match the elements from the column A with their corresponding concepts in the column B:**

|  |  |  |
| --- | --- | --- |
| **Answer** | **Column A** | **Column B** |
| 1…………. | 1. A linear data structure that follows the Last In First Out (LIFO) principle | a. Array |
| 2……………… | 1. A collection of key-value pairs that allows for fast data retrieval. | b. Linked List |
| 3……………… | 1. A linear data structure where elements are stored in contiguous memory locations. | c. Stack |
| 4……………… | 1. A linear data structure that follows the First In First Out (FIFO) principle. | d. Queue |
| 5………………… | 1. A linear collection of nodes where each node points to the next. | e. Hash Table |
|  |  | f.Tree |

1. **Answer by True (T) if the statements below are correct or by False (F) if they are incorrect.**
2. An algorithm is a step-by-step procedure to solve a problem.
3. Bubble sort has a time complexity of O(n log n) in the worst case.
4. In a queue, elements are added at the rear and removed from the front.
5. A binary tree is a type of linear data structure.
6. An algorithm can only be implemented in one programming language.
7. Big O notation can describe both upper and lower bounds of an algorithm's complexity.
8. **Fill in the blanks by using appropriate words in the bracket (searching, binary tree, Boolean, merging, stack, tree)**

i. A data structure that stores data in a hierarchical format is called a \_\_\_\_\_\_\_\_\_\_.

ii. A \_\_\_\_\_\_\_\_\_\_ is a special type of tree in which each node has at most two children.

iii. In a \_\_\_\_\_\_\_\_\_\_, the last element added is the first one to be removed.

iv. The process of finding a specific item in a data structure is called \_\_\_\_\_\_\_\_\_\_.

* + - 1. \_\_\_\_\_\_\_\_\_\_ is used to combine two lists into one.
      2. \_\_\_\_\_\_\_\_\_\_ is a data type which can store a true/false value in algorithm.

**4. Choose the best answer**

1. What does Big O notation describe?
2. The exact running time of an algorithm
3. The upper bound of an algorithm's time complexity
4. The lower bound of an algorithm's time complexity
5. The average case performance
6. Which of the following is a linear search algorithm?
7. Binary Search
8. Jump Search
9. Interpolation Search
10. Sequential Search
11. Which sorting algorithm has the best average-case time complexity?
12. Bubble Sort
13. Insertion Sort
14. Merge Sort
15. Selection Sort
16. Which of the following algorithms is not a comparison-based sorting algorithm?
17. Merge Sort
18. Heap Sort
19. Counting Sort
20. Quick Sort
21. What is the time complexity of Quick Sort in the average case?
22. O(n)
23. O(n log n)
24. O(n^2)
25. O(log n)
26. Which tool is used to convert C code into machine-readable code?
27. Editor
28. Compiler
29. Debugger
30. Linker

**Practical assessment**

Visit your school’s computer lab and use algorithm and C program to build a Student Management System (SMS). This system will store a list of student records and provide operations such as adding new students, removing students, searching for students by ID, sorting students by grade, and displaying the list of students.

**END**

**References**

### Books

**Sedgewick, R., & Wayne, K.** (2011). Algorithms (4th ed.). Addison-Wesley.

**Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). Introduction to algorithms (3rd ed.). MIT Press.

**Sears, M., & Haggard, L.** (2019). Data structures and algorithm analysis in C (4th ed.). Pearson.

**Kumar, A., & Singh, S.** (2017). Data structures and algorithms using C. Khanna Publishing.

**Knuth, D. E.** (1997). The art of computer programming, Volume 1: Fundamental algorithms (3rd ed.). Addison-Wesley.

Hill.

**Bhagat, A.** (2013). Mastering data structures through C. BPB Publications.

### Web Links

GeeksforGeeks. (n.d.). Data Structures. Retrieved from https://www.geeksforgeeks.org/data-structures/

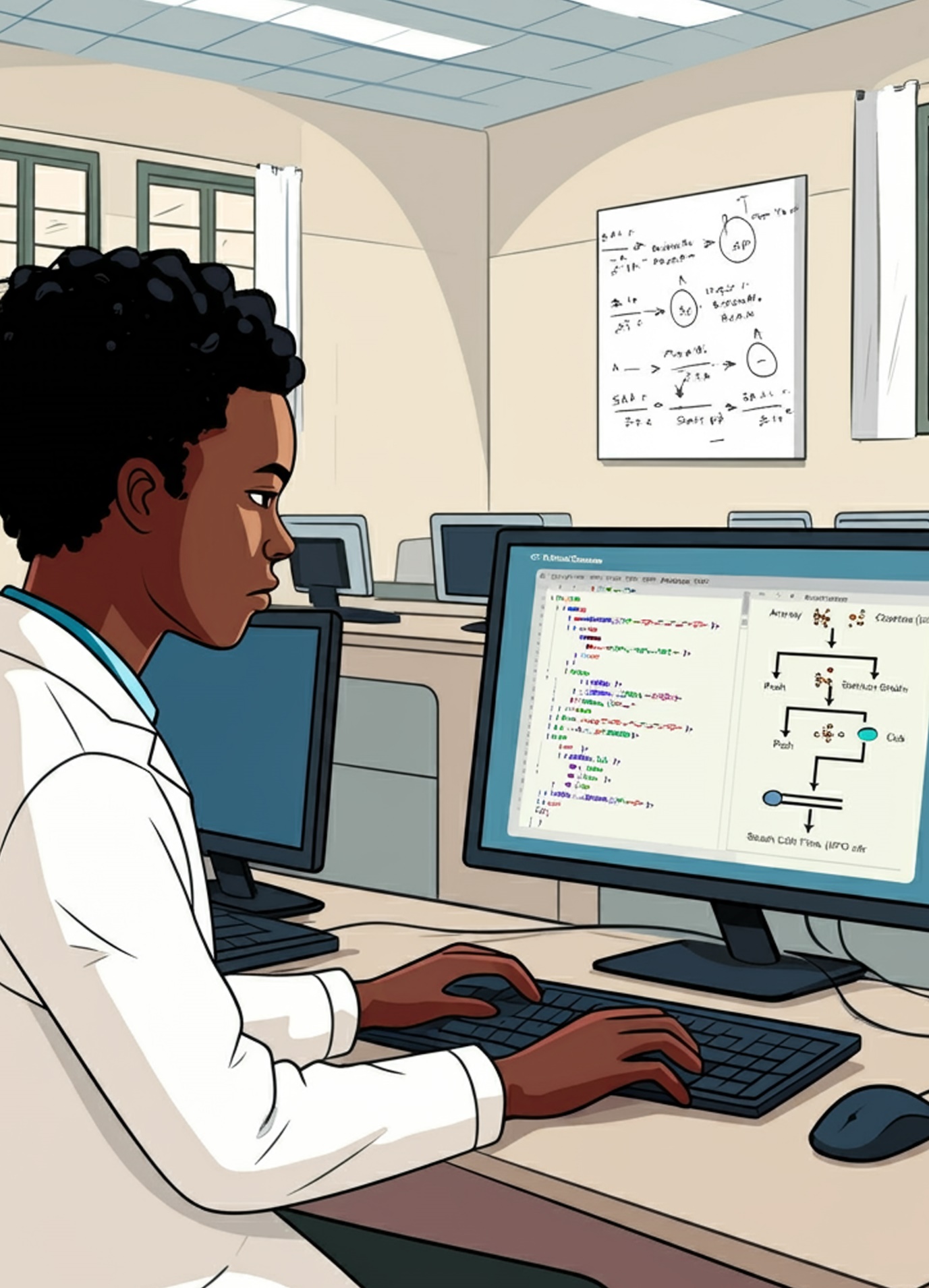
TutorialsPoint. (n.d.). C Data Structures. Retrieved from https://www.tutorialspoint.com/cprogramming/c\_data\_structures.htm

Programiz. (n.d.). Data Structures and Algorithms in C. Retrieved from https://www.programiz.com/dsa

Khan Academy. (n.d.). Algorithms. Retrieved from <https://www.khanacademy.org/computing/computer-science/algorithms>

W3Schools. (n.d.). C Programming Language - Learn C with Examples. Retrieved from https://www.w3schools.com/c/

Learning Outcome 2: Apply Linear data structure



|  |
| --- |
| **Indicative contents**  **2.1 Identification of linear data structure concepts**  **2.2 Applying arrays data structure**  **2.3 Applying linked list data structure**  **2.4 Applying queue data structure**  **2.5 Applying stack data structure** |

## Key Competencies for Learning Outcome 2: Apply Linear Data Structure

|  |  |  |
| --- | --- | --- |
| **Knowledge** | **Skills** | **Attitudes** |
| * Identification of linear data structure * Description of Array types * Description of linked list data structure * Description on representation of linked list * Description of queue data structure * Description of stack data structure | * Performing Operations on linear data structures. * Applying array basic operations in algorithms * Implementing of array basic operations in C * Applying linked lists basic operations in algorithms. * Implementing of linked list basic operations in C * Applying queue basic operations in algorithms. * Implementing of queue basic operations in C | * Having Adaptability in Performing Operations on linear data structures. * Being decision maker in applying array basic operations in algorithms * Being Practical oriented in implementing of array basic operations in C * Have Creativity in applying linked lists basic operations in algorithms. * Have critical thinking in implementing of linked list basic operations in C * Having problem solving skins in applying queue basic operations in algorithms. * Being innovative in Implementing of queue basic operations in C |

|  |  |  |
| --- | --- | --- |
| **Duration: 40 hrs** | | |
| **Learning outcome 2 objectives**:  By the end of the learning outcome, the trainees will be able to:  1. Identify clearly Linear Data Structures concepts based on intended use.  2. Describe array correctly based on data structure concepts  3. Apply properly Arrays based on their use cases.  4. Describe clearly linked list data structure on algorithm based on algorithm standards  5. Apply properly linked list based on algorithm standards  6. Describe correctly queue data structure based on intended use.  7. Apply properly Queue based on algorithm standards.  8. Describe correctly stack data structure based on intended use.  9. Implement effectively stack operations based on C programming language standards | | |
| **Resources** | | |
| **Equipment** | **Tools** | **Materials** |
| * Computer | * C Compiler * Integrated Development Environment (IDE) * VisuAlgo * Text editors | * Internet * Electricity |

## Indicative Content 2.1: Identification of Linear Data Structure Concepts



****

**Duration: 6 hrs**



**Theoretical Activity 2.1.1: Description of linear data structure**

**Tasks:**

1: Answer to the following questions:

* 1. What do you understand by linear data structure?
  2. What are types of linear data structure?
  3. What are key characteristics of linear data structure?
  4. Explain the Operations performed on linear data structures.

2: Write your answers on a papers/flipcharts

3: Present your findings to the whole class.

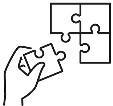
4: Ask for clarifications if necessary

5: Read the key reading 2.1.1. in trainee manual

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 2.1.1.: Description of linear data structure**  **1. Definition of linear data structure**  A linear data structure is a type of data structure in which elements are arranged in a sequential manner. In this structure, each element is connected to its previous and next element, forming a linear sequence. The primary characteristic of linear data structures is that they allow for simple and efficient traversal of their elements.  **2. Types of linear data structure**  **2.1. Arrays**  **An Array** is a data structure used to collect multiple data elements of the same data type into one variable. Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable.  The key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.    Figue: Array  Arrays can be classified into different types:  **One-Dimensional Array**: An Array with only one row of data elements is known as a One-Dimensional Array.  **Two-Dimensional Array**: An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.  **Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.   * 1. **Linked Lists**   A Linked List is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.    **Figure:** Linked list  Linked List can be classified into different types:  **2.2.1 Singly Linked List**: Each node points to the next node; traversal is one-way.  **2.2.2 Doubly Linked List**: Each node points to both the next and the previous nodes; allows traversal in both directions.  **2.2.3 Circular Linked List:** The last node points back to the first node, forming a circle.   * 1. **Stack**   is a Linear Data Structure that follows the LIFO (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List.  **The primary operations in the Stack are as follows:**  Push: Operation to insert a new element in the Stack is termed as Push Operation.  Pop: Operation to remove or delete elements from the Stack is termed as Pop Operation.    **Figure:** Stack   * 1. **Queues**   **A Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.    **Figure:** A Real-life Example of Queue  **The following are the primary operations of the Queue:**   * **Enqueue**: The insertion or Addition of some data elements to the Queue.   The element insertion is always done with the help of the rear pointer.   * **Dequeue**: Deleting or removing data elements from the Queue is termed Dequeue.   The deletion of the element is always done with the help of the front pointer    **Figure:** Queue   1. **Key Characteristics of linear data structure:**    1. **Sequential Organization**   Elements are arranged in a linear sequence, with a clear order.   * 1. **Fixed Size or Dynamic Size**   Fixed Size: Structures like arrays have a predetermined size.  Dynamic Size: Structures like linked lists can grow or shrink as needed.   * 1. **Contiguous Memory Allocation**   In arrays, elements are stored in contiguous memory locations, allowing for fast access.   * 1. **Direct Access**   Elements can be accessed directly via their index (in arrays) or through pointers (in linked lists).   * 1. **Single Level of Data Organization**   Each element in a linear data structure is connected to at most two other elements (previous and next), forming a single level.   * 1. **Ease of Traversal**   Linear data structures allow for straightforward traversal from one element to the next.   * 1. **Insertion and Deletion Complexity**   Insertion and deletion operations can vary in complexity:  Arrays: May require shifting elements.  Linked Lists: Can be done with pointer adjustments, making it more efficient in some cases.   1. **Operations performed on linear data structures.**    1. **Array**  * **Insertion**: Adding an element at a specified index. * **Deletion**: Removing an element from a specified index. * **Traversal**: Accessing each element in the array sequentially. * **Searching**: Finding an element by value (linear or binary search). * **Updating**: Modifying the value of an element at a specified index.   1. **Linked List** * **Insertion**: Adding a node at the beginning, end, or a specified position. * **Deletion**: Removing a node from the beginning, end, or a specified position. * **Traversal**: Accessing each node sequentially from the head. * **Searching**: Finding a node by value. * **Updating**: Modifying the value of a node.   1. **Stack** * **Push**: Adding an element to the top of the stack. * **Pop**: Removing the top element from the stack. * **Peek/Top**: Accessing the top element without removing it. * **IsEmpty**: Checking if the stack is empty.   1. **Queue** * **Enqueue**: Adding an element to the rear of the queue. * **Dequeue**: Removing the front element from the queue. * **Front/Peek**: Accessing the front element without removing it. * **IsEmpty**: Checking if the queue is empty.   1. **Deque (Double-Ended Queue)** * **Insert Front**: Adding an element to the front. * **Insert Rear**: Adding an element to the rear. * **Delete Front**: Removing an element from the front. * **Delete Rear**: Removing an element from the rear. * **Peek Front**: Accessing the front element without removing it. * **Peek Rear**: Accessing the rear element without removing it.   1. **String** * **Concatenation**: Combining two or more strings. * **Substring**: Extracting a portion of the string. * **Searching**: Finding a character or substring. * **Replacing**: Modifying part of the string with another string. * **Length**: Determining the number of characters. |

** Points to Remember**

* In description of Linear data structure, keep in mind that:
* Elements are arranged sequentially, with each element connected to its predecessor and successor
* The types of linear data structure are: Arrays, Linked Lists, Stacks and Queues
* Each type of Linear Data Structures has its own ways of Access, Insertion, Deletion, Traversal, Search and sorting its elements.
* Key Characteristics of linear data structure are: sequential organization, fixed size or dynamic size, contiguous memory allocation, direct access , single level of data organization, ease of traversal, insertion and deletion complexity
* Linear Data Structures has different technics to access to elements such as: Sequential Order, Single Level and Direct Access
* Operations performed on linear data structures are: Insertion, Deletion, Traversal, Searching, Updating

**Application of learning 2.1.**

You are tasked to visit the nearest bank of your school to see how the bank manages customer requests . Based on the observation done, elaborate the report on the following:

1. What linear data structure that represents the queue of customers waiting for their turn?
2. Identify other types of linear data structures that could be used for different purposes in the bank’s system.
3. What are the key characteristics of linear data structures that make them suitable for handling customer requests in an orderly manner?
4. What operations needed to perform on the customer queue to manage it effectively?

## Indicative Content 2.2: Applying Arrays Data Structure



****

**Duration: 10 hrs**



**Theoretical Activity 2.2.1: Description of array data structure**

**Tasks:**

1: Answer the following questions related to array data structures.

1. What do you understand by the following terms below?
2. Declare arrays
3. Initialize arrays
4. Access array Elements
5. Iterate over arrays
6. Array types
7. Array basic operations in algorithms

2: Write your answers on flipcharts/papers.

3: Present the findings/answers to the whole class

5: Ask questions where necessary

6: Read the key readings 2.2.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 2.2.1.: Description of array data structure**   1. **Declaring Arrays**   **Algorithm:**  Step 1: Define the data type for the array.  Step 2: Specify the size of the array.  Step 3: Name the array.  **Example (C-style syntax):**  // Step 1: Declare an array of integers with a size of 5  int array[5];   1. **Initializing Arrays**   **Algorithm:**  Step 1: For each position in the array, assign an initial value.  Step 2: If no values are assigned, default values will depend on the language (e.g., 0 for integers in C/C++).  **Example:**  // Step 1: Initialize the array with specific values  int array[5] = {1, 2, 3, 4, 5};   1. **Accessing Array Elements**   **Algorithm:**  Step 1: Access any element of the array using its index.  Step 2: Use the array name followed by the index in square brackets.  Example:  // Accessing the third element of the array (index 2)  int thirdElement = array[2]; // thirdElement = 3   1. **Iterating Over Arrays**   **Algorithm:**  Step 1: Start a loop that runs from index 0 to the size of the array minus 1.  Step 2: In each iteration, access the element at the current index and perform the desired operation.  Step 3: Continue until all elements are processed.  **Example (C-style syntax):**  // Step 1: Start a loop from 0 to the size of the array  for(int i = 0; i < 5; i++) {  // Step 2: Access and print each element  printf("%d ", array[i]);  }   1. **Array types**    1. **One Dimensional Array**   A one-dimensional array is a simple linear collection of elements stored in contiguous memory locations. It is accessed using a single index.  **5.1.1. Algorithm for One Dimensional Array:**  Step 1: Declare the array by specifying its type and size.  Step 2: Initialize the array by assigning values to its elements.  Step 3: Access the elements of the array using the index.  Step 4: Iterate over the array to perform operations on each element.  **Example:**  // Step 1: Declare an integer array with 5 elements  int array[5];  // Step 2: Initialize the array  array[0] = 10;  array[1] = 20;  array[2] = 30;  array[3] = 40;  array[4] = 50;  // Step 3: Access the third element  int thirdElement = array[2]; // thirdElement = 30  // Step 4: Iterate over the array and print all elements  for(int i = 0; i < 5; i++) {  printf("%d ", array[i]);  }  **5.2 . Multidimensional Arrays**  A multidimensional array is an array of arrays, where each element is itself an array. The most common form is a two-dimensional array (matrix), but higher dimensions are also possible.  **5.2.1 For Two-Dimensional Arrays:**  Step 1: Declare the array by specifying its type and dimensions (rows and columns).  Step 2: Initialize the array by assigning values to its elements.  Step 3: Access elements using two indices (one for row, one for column).  Step 4: Iterate over the array using nested loops to perform operations.  **5.2.2 For Three-Dimensional Arrays:**  Step 1: Declare the array by specifying type and three dimensions (x, y, z).  Step 2: Initialize the array by assigning values to its elements.  Step 3: Access elements using three indices (one for each dimension).  Step 4: Iterate over the array using nested loops for operations.   1. **Array Basic Operations in Algorithms**    1. **Declaration**   Array declaration is the process of specifying the size and data type of the array. In most programming languages, this is done before initializing the array.  **6.1.1 Algorithm for Array Declaration:**  1. Start  2. Define the size of the array (size = n).  3. Declare an array of a specific data type (int, float, char, etc.).  4. End  **6.1.2 Array Declaration in C**  **Example:**  **int array [5]; // Declare an integer array of size 5.**   * 1. **Initialization**   Array initialization involves assigning values to the elements in the array either at the time of declaration or later.  **6.2.1 Algorithm for Array Initialization:**  1. Start  2. For each index i from 0 to size - 1:  a. Assign a value to array[i].  3. End  **6.2.2 Array Initialization in C**  **Example:**  int array[5] = {1, 2, 3, 4, 5}; // Initialize an array with 5 elements.   * 1. **Accessing Elements**   **6.3.1 Algorithm for Accessing Elements:**  1. Start  2. Check if index is valid (0 ≤ index < size).  3. If valid, return array[index].  4. Else, print "Index out of bounds."  5. End  **6.3.2 Accessing Elements in C**  **Example:**  int x = array[2]; // Access the element at index 2.   * 1. **Updating Elements**   Updating an element involves changing the value at a specific index.  **6.4.1 Algorithm for Updating Elements:**  1. Start  2. Check if index is valid (0 ≤ index < size).  3. If valid, update array[index] with newValue.  4. Else, print "Index out of bounds."  5. End  **6.4.2 Updating Elements in C**  **Example:**  array[3] = 10; // Update the element at index 3 with the value 10.   * 1. **Deleting Elements**   Deleting an element in an array requires shifting all elements after the deleted element to the left. However, arrays have a fixed size, so we only simulate deletion by shifting elements.  **6.5.1 Algorithm for Deleting Elements:**  1. Start  2. Check if index is valid (0 ≤ index < size).  3. If valid, shift all elements after the index one position to the left.  4. Decrement the size of the array.  5. Else, print "Index out of bounds."  6. End  **6.5.2 Deleting Elements in C**  **Example:**  For array = {1, 2, 3, 4, 5}, deleting element at index 2 results in {1, 2, 4, 5}.   * 1. **Iterating Over the Array**   Iteration involves going through each element in the array, one by one.  **6.6.1 Algorithm for Iterating Over the Array:**  1. Start  2. For each index i from 0 to size - 1:  a. Print array[i].  3. End  **6.6.2** **Iterating Over the Array in C**  **Example:**  For i = 0 to size - 1:  Print array[i];   1. **Array basic operations in C**    1. **Declaration of Arrays in C**   Declaring an array in C means defining its type and size. The size of an array must be a constant expression.  **Syntax:**  data\_type array\_name[size];  Example:  int arr[5]; // Declare an integer array of size 5   * 1. **Initialization of Arrays in C**   An array can be initialized when it is declared. Initialization assigns specific values to the elements in the array.  **Syntax:**  data\_type array\_name[size] = {value1, value2, ..., valueN};  Example:  int arr[5] = {1, 2, 3, 4, 5}; // Declare and initialize an array of 5 integers  If fewer elements are provided than the size, the remaining elements are initialized to zero.   * 1. **Accessing Array Elements in C**   Array elements are accessed using an index. The index of an array starts from 0, meaning the first element is at index 0, the second at index 1, and so on.  **Syntax:**  array\_name[index];  **Example:**  int value = arr[2]; // Access the element at index 2 (third element)  printf("%d", value); // Output: 3   * 1. **Updating Array Elements in C**   You can update an element in an array by directly accessing it using its index.  **Syntax:**  array\_name[index] = new\_value;  **Example:**  arr[3] = 10; // Update the element at index 3 with the value 10   * 1. **Deleting Array Elements in C**   Arrays in C have a fixed size, so elements cannot be deleted directly. However, you can shift elements to "simulate" deletion by shifting the remaining elements one position to the left.  **Example:**  // Function to delete an element from an array  void deleteElement(int arr[], int \*size, int index) {  if (index < \*size) {  for (int i = index; i < \*size - 1; i++) {  arr[i] = arr[i + 1];  }  (\*size)--;  }  }  In this example, the deleteElement function shifts elements to the left after the specified index, effectively "deleting" the element.   * 1. **Iterating Over Arrays in C**   You can use a loop to iterate over all the elements in an array. The for loop is commonly used for this purpose.  **Example:**  for (int i = 0; i < 5; i++) {  printf("%d ", arr[i]); // Output: 1 2 3 4 5  }  This loop iterates over the entire array and prints each element. |



**Practical Activity 2.2.2: Applying array basic operations**

**Task:**

1: Referring to key reading 2.2.2, Perform the following task:

As a trainee in Computer System and Architecture, apply array basic operations using algorithm and C program

2: Present steps to apply array basic operations

3: Present your work to the trainer and the whole class

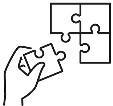
4: Ask for clarifications if any

5: Perform the activity in the application of learning 2.2

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 2.2.2.: Applying array basic operations**  Below are the basic operations on arrays along with their algorithms and C implementations. The operations included are:   * **Insertion** * **Deletion** * **Traversal** * **Searching**   **1. Insertion**  **1.1 Algorithm:**   * If the array is full, return an error. * Shift elements to the right from the specified index. * Insert the new element at the specified index.   **1.2 C Program:**  #include <stdio.h>  #define MAX 100  void insert(int arr[], int \*n, int index, int value) {  if (\*n >= MAX) {  printf("Array is full!\n");  return;  }  for (int i = \*n; i > index; i--) {  arr[i] = arr[i - 1];  }  arr[index] = value;  (\*n)++;  }  int main() {  int arr[MAX], n = 5; // Current number of elements  arr[0] = 10; arr[1] = 20; arr[2] = 30; arr[3] = 40; arr[4] = 50;  int index = 2, value = 25;  insert(arr, &n, index, value);  printf("Array after insertion: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **2. Deletion**  **2.1 Algorithm**   * If the array is empty, return an error. * Shift elements to the left from the specified index. * Decrease the size of the array.   **2.2 C Program:**  #include <stdio.h>  #define MAX 100  void delete(int arr[], int \*n, int index) {  if (\*n <= 0) {  printf("Array is empty!\n");  return;  }  for (int i = index; i < \*n - 1; i++) {  arr[i] = arr[i + 1];  }  (\*n)--;  }  int main() {  int arr[MAX], n = 5; // Current number of elements  arr[0] = 10; arr[1] = 20; arr[2] = 30; arr[3] = 40; arr[4] = 50;  int index = 2;  delete(arr, &n, index);  printf("Array after deletion: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  }  **3. Traversal**  **3.1. Algorithm:**   * Start from the first element and go to the last element. * Print each element.   **3.2. C Program:**  #include <stdio.h>  void traverse(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  int main() {  int arr[] = {10, 20, 30, 40, 50};  int n = sizeof(arr) / sizeof(arr[0]);  printf("Traversing the array: ");  traverse(arr, n);  return 0;  }  **4. Searching**   * 1. **Algorithm (Linear Search):** * Start from the first element and compare it with the target. * If found, return the index. * If not found, continue to the next element. * If the end of the array is reached, return -1.   1. **C Program:**   #include <stdio.h>  int linear\_search(int arr[], int n, int target) {  for (int i = 0; i < n; i++) {  if (arr[i] == target) {  return i; // Return the index  }  }  return -1; // Not found  }  int main() {  int arr[] = {10, 20, 30, 40, 50};  int n = sizeof(arr) / sizeof(arr[0]);  int target = 30;  int result = linear\_search(arr, n, target);  if (result != -1) {  printf("Element %d found at index %d.\n", target, result);  } else {  printf("Element %d not found in the array.\n", target);  }  return 0;  } |

** Points to Remember**

* While describing array data structure , remember the following:
* **Declaring Arrays**: Arrays are declared by specifying the **data type**, **size**, and **name**.
* **Initializing Arrays**: Arrays can be initialized during declaration or later by assigning values to each index.
* **Accessing Array Elements**: Array elements are accessed using their **index**, starting from 0.
* **Iterating Over Arrays**: To process each element, use a loop that runs from index 0 to the array size minus one.
* **Array Types**:
* **One-dimensional Array**: A simple list of elements accessed using one index.
* **Multidimensional Array**: An array of arrays (e.g., 2D or 3D) accessed using multiple indices.
* While applying basic array operations, remember the following:
  + **Insertion steps**
* **Identify Position:** Determine where to insert the new element.
* **Shift Elements**: Move elements from the position to the right.
* **Insert Element**: Place the new element at the specified position.
* **Update Size**: Increase the size of the array.
* **Deletion steps**
* **Identify Position:** Determine which element to delete.
* **Shift Elements**: Move elements from the position to the left.
* **Update Size:** Decrease the size of the array.
* **Traversal** **steps**
  + **Iterate Through Array:** Loop through each element.
  + **Access Elements:** Perform the desired operation (e.g., print) on each element.
* **Searching (Linear Search) steps**
  + **Iterate Through Array:** Loop through each element.
  + **Check for Match:** Compare each element to the target value.
* **Return Index:** If found, return the index; if not found, return an indication (e.g., -1)

 **Application of learning 2.2.**

Visit your school computer lab and Use algorithm and C program to declare an array of 10 integers that holds product stock levels and initialize a separate array of 5 floating-point numbers for product prices. The array will access and print the third name from an array of customer orders, and iterate through the product stock array to display each value.

## Indicative Content 2.3: Applying Linked list Data Structure



****

**Duration: 10 hrs**



**Theoretical Activity 2.3.1:** **Description of linked list data structure**

**Tasks:**

1: Respond to the following questions:

i. What do you understand by the following terms?:

a) Link

b) Next

c) Linked List

ii. Explain the way in which the linked list is represented

iii. What are types of linked list?

2: Write the answers on papers/flip chart

3: Present the findings to the whole class

4: For clarification, ask questions if any

5: For more clarifications, read the key readings 2.3.1

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 2.3.1.: Description of linked list data structure**   1. **Explanation of the terms related to linked lists**    1. **Link**   **A link is** a connection between two elements (or nodes) in a data structure, particularly in linked lists. It allows traversal and organization of data in a non-contiguous manner.  **Function**: In a linked list, each link (or pointer) points to the next element in the sequence, enabling the structure to maintain its order without needing contiguous memory allocation.   * 1. **Next**   **The term "next"** refers to a pointer or reference in a node of a linked list that points to the subsequent node in the sequence.  **Function:** The "next" pointer is essential for traversing the list and linking elements together. In a singly linked list, each node contains one "next" pointer, while in a doubly linked list, nodes typically have both "next" and "previous" pointers.   * 1. **Linked List**   **A linked list** is a linear data structure where elements, called nodes, are stored in a non-contiguous manner. Each node contains data and a pointer to the next node in the sequence.  **Function:** Linked lists allow for efficient insertion and deletion of elements since they do not require shifting elements like arrays do. They can grow dynamically as needed.   1. **Representation of Linked Lists**   A linked list is represented using a series of nodes, where each node consists of two main components:   * 1. **Data** * The information or value stored in the node.   **2.2. Link/Pointer**   * A reference to the next node in the list.  1. **Types of Linked Lists**   There are several types of linked lists, each suited for different purposes:   * 1. **Singly Linked List** * Each node contains data and a single pointer to the next node. * Traversal is one-way, from the head to the end of the list.   1. **Doubly Linked List** * Each node contains data, a pointer to the next node, and a pointer to the previous node. * Allows traversal in both directions (forward and backward).   1. **Circular Linked List** * The last node's "next" pointer points back to the first node, creating a circular structure.Can be singly or doubly linked; traversal can continue indefinitely.   1. **Circular Doubly Linked List** * Combines features of both circular and doubly linked lists. * Each node has pointers to both the next and previous nodes, and the last node points back to the first node, forming a circular structure. |



**Practical Activity 2.3.2:** **Implementing Linked list basic operations**

**Task:**

1: Referring to the key reading 2.3.2, read the following task:

As a trainee in Computer System and Architecture, you are asked to go to the computer lab to Implement linked list basic operations in Algorithm and in C program then display the output.

2: Present steps to apply linked list basic operations

3: Present your work to the trainer and whole class

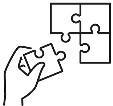
4: Ask clarification where necessary

5: Perform the task provided in application of learning 2.3

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 2.3.2.: Implementing Linked list basic operations**  **Basic Operations in Linked List**  The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below:   1. **Insertion:** Adds an element at the beginning of the list. 2. **Deletion**: Deletes an element at the beginning of the list. 3. **Display:** Displays the complete list. 4. **Search**: Searches an element using the given key. 5. **Delete:** Deletes an element using the given key.   **1.Insertion**  **1.1 Insertion Algorithm**  1. START  2. Create a node to store the data  3. Check if the list is empty  4. If the list is empty, add the data to the node and  assign the head pointer to it.  5. If the list is not empty, add the data to a node and link to the  current head. Assign the head to the newly added node.  6. END  **1.2 Insertion in C program**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  struct node {  int data;  struct node \*next;  };  struct node \*head = NULL;  struct node \*current = NULL;  // display the list  void printList(){  struct node \*p = head;  printf("\n[");  //start from the beginning  while(p != NULL) {  printf(" %d ",p->data);  p = p->next;  }  printf("]");  }  //insertion at the beginning  void insertatbegin(int data){  //create a link  struct node \*lk = (struct node\*) malloc(sizeof(struct node));  lk->data = data;  // point it to old first node  lk->next = head;  //point first to new first node  head = lk;  }  void main(){  int k=0;  insertatbegin(12);  insertatbegin(22);  insertatbegin(30);  insertatbegin(44);  insertatbegin(50);  printf("Linked List: ");  // print list  printList();  }  **Out put**  Linked List:  [ 50 44 30 22 12]  **2. Deletion Operation**  **2.1 Deletion Algorithm**  1. START  2. Assign the head pointer to the next node in the list  3. END  **2.2 Deletion in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  struct node {  int data;  struct node \*next;  };  struct node \*head = NULL;  struct node \*current = NULL;  // display the list  void printList(){  struct node \*p = head;  printf("\n[");  //start from the beginning  while(p != NULL) {  printf(" %d ",p->data);  p = p->next;  }  printf("]");  }  //insertion at the beginning  void insertatbegin(int data){  //create a link  struct node \*lk = (struct node\*) malloc(sizeof(struct node));  lk->data = data;  // point it to old first node  lk->next = head;  //point first to new first node  head = lk;  }  void deleteatbegin(){  head = head->next;  }  int main(){  int k=0;  insertatbegin(12);  insertatbegin(22);  insertatbegin(30);  insertatbegin(40);  insertatbegin(55);  printf("Linked List: ");  // print list  printList();  deleteatbegin();  printf("\nLinked List after deletion: ");  // print list  printList();  }  **2.3 Output**  Linked List:  [ 55 40 30 22 12 ]  Linked List after deletion:  [ 40 30 22 12 ]  **3.Reversal Operation**  **3.1 Reversal Algorithm**  1. START  2. We use three pointers to perform the reversing:  prev, next, head.  3. Point the current node to head and assign its next value to  the prev node.  4. Iteratively repeat the step 3 for all the nodes in the list.  5. Assign head to the prev node.  **3.2 Reversal in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  struct node {  int data;  struct node \*next;  };  struct node \*head = NULL;  struct node \*current = NULL;  // display the list  void printList(){  struct node \*p = head;  printf("\n[");  //start from the beginning  while(p != NULL) {  printf(" %d ",p->data);  p = p->next;  }  printf("]");  }  //insertion at the beginning  void insertatbegin(int data){  //create a link  struct node \*lk = (struct node\*) malloc(sizeof(struct node));  lk->data = data;  // point it to old first node  lk->next = head;  //point first to new first node  head = lk;  }  void reverseList(struct node\*\* head){  struct node \*prev = NULL, \*cur=\*head, \*tmp;  while(cur!= NULL) {  tmp = cur->next;  cur->next = prev;  prev = cur;  cur = tmp;  }  \*head = prev;  }  void main(){  int k=0;  insertatbegin(12);  insertatbegin(22);  insertatbegin(30);  insertatbegin(40);  insertatbegin(55);  printf("Linked List: ");  // print list  printList();  reverseList(&head);  printf("\nReversed Linked List: ");  printList();  }  **Out put**  Linked List:  [ 55 40 30 22 12 ]  Reversed Linked List:  [ 12 22 30 40 55 ]   1. **Search Operation**   **4.1 Search Algorithm**  1 START  2 If the list is not empty, iteratively check if the list  contains the key  3 If the key element is not present in the list, unsuccessful search   1. END   **4.2. Search Operation in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  struct node {  int data;  struct node \*next;  };  struct node \*head = NULL;  struct node \*current = NULL;  // display the list  void printList(){  struct node \*p = head;  printf("\n[");  //start from the beginning  while(p != NULL) {  printf(" %d ",p->data);  p = p->next;  }  printf("]");  }  //insertion at the beginning  void insertatbegin(int data){  //create a link  struct node \*lk = (struct node\*) malloc(sizeof(struct node));  lk->data = data;  // point it to old first node  lk->next = head;  //point first to new first node  head = lk;  }  int searchlist(int key){  struct node \*temp = head;  while(temp != NULL) {  if (temp->data == key) {  return 1;  }  temp=temp->next;  }  return 0;  }  void main(){  int k=0;  insertatbegin(12);  insertatbegin(22);  insertatbegin(30);  insertatbegin(40);  insertatbegin(55);  printf("Linked List: ");  // print list  printList();  int ele = 30;  printf("\nElement to be searched is: %d", ele);  k = searchlist(30);  if (k == 1)  printf("\nElement is found");  else  printf("\nElement is not found in the list");  }  **Output**  Linked List:  [ 55 40 30 22 12 ]  Element to be searched is: 30  Element is found  **5. Linked List - Traversal Operation**   * 1. **Traversal Operation Algorithm**   1. START  2. While the list is not empty and did not reach the end of the list  in the data in each node  3. END  **5.2. Traversal Operation in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  struct node {  int data;  struct node \*next;  };  struct node \*head = NULL;  struct node \*current = NULL;  // display the list  void printList(){  struct node \*p = head;  printf("\n[");  //start from the beginning  while(p != NULL) {  printf(" %d ",p->data);  p = p->next;  }  printf("]");  }  //insertion at the beginning  void insertatbegin(int data){  //create a link  struct node \*lk = (struct node\*) malloc(sizeof(struct node));  lk->data = data;  // point it to old first node  lk->next = head;  //point first to new first node  head = lk;  }  void main(){  int k=0;  insertatbegin(12);  insertatbegin(22);  insertatbegin(30);  printf("Linked List: ");  // print list  printList();  }  **Output:**  Linked List:  [ 30 22 12] |

** Points to Remember**

* While describing linked list data structure, take into consideration the following:
* Link/Pointer Connects nodes together, allowing traversal.
* Head: The first node in the list, or NULL if the list is empty.
* Each node has at least two components: data and a pointer to the next node.
* The head pointer is critical for accessing the list.
* Each Type of Linked Lists has its own way to arrange its nodes.
* Linked list has 4 operations such as: Insertion, Deletion, Traversal and Searching
* A linked list is a linear data structure where elements (nodes) are not stored in contiguous memory locations
* A linked list is represented by a node structure
* Each node contains an integer data and a pointer to the next node.
* While implementing Linked list basic operations in C ,take into consideration the following:
* Insertion: Adds an element at the beginning of the list.
* Deletion: Deletes an element at the beginning of the list.
* Display: Displays the complete list.
* Search: Searches an element using the given key.
* Delete: Deletes an element using the given key.

 **Application of learning 2.3.**

As you have learnt the linked lists, and their basic operations (add, remove, and display) using algorithm and C program, you are requested to go in computer Lab of your school; then develop a simple Student Management System where you can maintain a list of students using linked lists. The system should allow you to add, remove, and display students.

## Indicative Content 2.4: Applying Queue Data Structure



****

**Duration: 7 hrs**



**Theoretical Activity 2.4.1: Description of queue data structure**

**Tasks:**

1: Describe the following terms in data structure:

1. Queue
2. Representation
3. Circular queue
4. Queue primary functions.
5. Queue basic operations in algorithms.

2: Provide your answers on flipcharts/papers.

3: Present the findings/answers to the whole class or trainer

4: Ask questions where necessary.

5: For more clarifications, read the key readings 2.4.1.

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 2.4.1.: Description of queue data structure**   * 1. **Introduction to Queue data structure**   A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. This means that the element added first to the queue will be the first one to be removed, much like a real-world queue or line of people. Queues are useful when you want to process elements in the order they arrive.  The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.  A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.     1. **Definition of Key Terms**    1. **FIFO (First In, First Out):** This is the underlying principle of a queue. The element that enters first will be the first to exit, which is the opposite of a stack (LIFO - Last In, First Out).    2. **Enqueue:** The operation of adding an element to the back of the queue.    3. **Dequeue:** The operation of removing an element from the front of the queue.    4. **Front (Head):** The first element in the queue, which will be removed next during a dequeue operation.    5. **Rear (Tail):** The last element in the queue, where new elements are added during the enqueue operation. 2. **Basic Terminologies**    1. **Queue:** The overall data structure that holds a collection of elements.    2. **Front (Head):** Refers to the position of the element that will be dequeued next. It’s the first element of the queue.    3. **Rear (Tail):** Refers to the position where the most recent element was enqueued. It’s the last element of the queue.    4. **Enqueue Operation:** The process of adding a new element at the rear of the queue.    5. **Dequeue Operation:** The process of removing the element from the front of the queue.    6. **Empty Queue:** A queue that has no elements. In this case, both front and rear are often set to null or an indicator of emptiness.    7. **Full Queue (for bounded queues):** Some implementations of queues are bounded, meaning they have a fixed size. A full queue occurs when the number of elements in the queue reaches its predefined maximum size.    8. **Circular Queue:** A type of queue where the end is connected back to the front, making the queue circular. It helps to utilize the empty spaces created by dequeuing elements from the front.    9. **Priority Queue:** A variation of the queue where each element is associated with a priority. Elements with higher priority are dequeued before elements with lower priority, regardless of their order of arrival.    10. **Size:** The number of elements currently in the queue.    11. **Peek (Front):** This function retrieves the element at the front without removing it.    12. **Is Empty:** This function checks if the queue is empty.    13. **Is Full:** This function checks if the queue is full (for fixed-size implementations).   **Example of a Queue**  Consider a queue with the following operations:   * **Initial Queue:** Empty * **Enqueue(1)** → Queue: [1] * **Enqueue(2)** → Queue: [1, 2] * **Dequeue()** → Queue: [2] (1 is removed, as it was first in) * **Enqueue(3)** → Queue: [2, 3]  1. **Queue Representation**   Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers. As a small example in this tutorial, we implement queues using a one-dimensional array.    A **queue** can be represented in different ways, most commonly as:   * **Array-based Queue:** A queue can be implemented using a fixed-size array. This is simple but can be inefficient in terms of space utilization since after several dequeue operations, space at the front of the array becomes unused unless shifted. * **Linked List-based Queue:** A dynamic implementation where each element in the queue is a node in a linked list. The front of the queue is the head of the linked list, and the rear is the tail. This approach allows for flexible queue sizes, but requires more memory due to the storage of pointers.  1. **Circular Queue**   A **Circular Queue** is a special type of queue where the last position is connected back to the first position to make a circle. This improves the efficiency of a regular queue, especially when it is implemented with arrays, by reusing the empty spaces created by dequeued elements.   * In a **normal queue** implemented with an array, if the rear reaches the last position, you can't enqueue more elements even if there is space in the front. * In a **circular queue**, the rear wraps around to the beginning of the array if there is space, ensuring efficient use of the allocated memory.   **Example**  Consider a circular queue with size 5:   * **Initial Queue:** Empty [\_, \_, \_, \_, \_] * **Enqueue(1), Enqueue(2), Enqueue(3), Enqueue(4), Enqueue(5)** → Queue: [1, 2, 3, 4, 5] * **Dequeue()** → Queue: [\_, 2, 3, 4, 5] * **Enqueue(6)** → Queue: [6, 2, 3, 4, 5] (6 is enqueued at the position where 1 was dequeued)  1. **Queue Primary Functions** 2. **Enqueue (Add):** Adds an element to the rear of the queue. 3. **Dequeue (Remove):** Removes and returns the element at the front of the queue. 4. **isEmpty():** Checks whether the queue is empty. 5. **isFull():** Checks whether the queue is full (useful for bounded queues or circular queues). 6. **Peek (Front):** Returns the element at the front without removing it. 7. **Queue Basic Operations in Algorithms** 8. **Enqueue Operation:**    * Check if the queue is full (for bounded queues).    * Add the element at the rear of the queue.    * Update the rear pointer. 9. **Dequeue Operation:**  * Check if the queue is empty. * Remove the element from the front of the queue. * Update the front pointer.  1. **Peek Operation:**   Return the element at the front without dequeuing. |



**Practical Activity 2.4.2: Implementing Queue Basic Operations**

**Task:**

1: Referring to key reading 2.4.2, Perform the following task:

As a trainee in Computer system and architecture, Implement Queue Basic Operations in Algorithm and C program

2: Present steps to implement queue basic operations

3: Present your work to the trainer and whole class

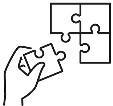
4: Ask for clarifications if any

5: Perform the task provided in application of learning 2.4

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 2.4.2.: Implementing Queue Basic Operations**   * 1. **Basic Operations in Queue**   The most fundamental operations in the queue ADT include:  **1. Enqueue:**Adds an element to the back of the queue.  **2. Dequeue:** Removes and returns the element from the front of the queue.  **3. Front/Peek:** Returns the front element of the queue without removing it.  **4. IsEmpty:** Checks whether the queue is empty.  **5. IsFull:Checks** whether the queue is full (for a fixed-size queue).  These are all built-in operations to carry out data manipulation and to check the status of the queue.  Queue uses two pointers − **front** and **rear**. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).  **1.1 Queue Insertion Operation: Enqueue()**  **The enqueue()** is a data manipulation operation that is used to insert elements into the stack. The following algorithm describes the enqueue() operation in a simpler way.  **1.1.1 Queue Insertion Algorithm**  1. START  2. Check if the queue is full.  3. If the queue is full, produce overflow error and exit.  4. If the queue is not full, increment rear pointer to point  the next empty space.  5. Add data element to the queue location, where the rear  is pointing.  6. return success.  7. END  **1.1.2 Queue Insertion in c**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX 6  int intArray[MAX];  int front = 0;  int rear = -1;  int itemCount = 0;  bool isFull(){  return itemCount == MAX;  }  bool isEmpty(){  return itemCount == 0;  }  int removeData(){  int data = intArray[front++];  if(front == MAX) {  front = 0;  }  itemCount--;  return data;  }  void insert(int data){  if(!isFull()) {  if(rear == MAX-1) {  rear = -1;  }  intArray[++rear] = data;  itemCount++;  }  }  int main(){  insert(3);  insert(5);  insert(9);  insert(1);  insert(12);  insert(15);  printf("Queue: ");  while(!isEmpty()) {  int n = removeData();  printf("%d ",n);  }  }  **Output**  Queue: 3 5 9 1 12 15  **1.2. Queue Deletion Operation: dequeue()**  **The dequeue()** is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.  **1.2.1 Queue Deletion Algorithm**  1. START  2. Check if the queue is empty.  3. If the queue is empty, produce underflow error and exit.  4. If the queue is not empty, access the data where front  is pointing.  5. Increment front pointer to point to the next available  data element.  6. Return success.  7. END  **1.2.2 Queue Deletion in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX 6  int intArray[MAX];  int front = 0;  int rear = -1;  int itemCount = 0;  bool isFull(){  return itemCount == MAX;  }  bool isEmpty(){  return itemCount == 0;  }  void insert(int data){  if(!isFull()) {  if(rear == MAX-1) {  rear = -1;  }  intArray[++rear] = data;  itemCount++;  }  }  int removeData(){  int data = intArray[front++];  if(front == MAX) {  front = 0;  }  itemCount--;  return data;  }  int main(){  int i;  /\* insert 5 items \*/  insert(3);  insert(5);  insert(9);  insert(1);  insert(12);  insert(15);  printf("Queue: ");  for(i = 0; i < MAX; i++)  printf("%d ", intArray[i]);  // remove one item  int num = removeData();  printf("\nElement removed: %d\n",num);  printf("Updated Queue: ");  while(!isEmpty()) {  int n = removeData();  printf("%d ",n);  }  }  **1.2.3 Output**  Queue: 3 5 9 1 12 15  Element removed: 3  Updated Queue: 5 9 1 12 15  **1.3 Queue - The peek () Operation**  The peek () is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.  **1.3.1 Queue - The peek () Algorithm**  1. START  2. Return the element at the front of the queue  3. END  **1.3.2 Queue - The peek () In C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX 6  int intArray[MAX];  int front = 0;  int rear = -1;  int itemCount = 0;  int peek(){  return intArray[front];  }  bool isFull(){  return itemCount == MAX;  }  void insert(int data){  if(!isFull()) {  if(rear == MAX-1) {  rear = -1;  }  intArray[++rear] = data;  itemCount++;  }  }  int main(){  int i;  /\* insert 5 items \*/  insert(3);  insert(5);  insert(9);  insert(1);  insert(12);  insert(15);  printf("Queue: ");  for(i = 0; i < MAX; i++)  ("%d ", intArray[i]);  printf("\nElement at front: %d\n",peek());  }  **Output**  Queue: 3 5 9 1 12 15  Element at front: 3  **1.4 Queue The isFull() Operation**  The isFull() operation verifies whether the stack is full.  **1.4.1 The isFull() Operation in Algorithm**  1. START  2. If the count of queue elements equals the queue size,  return true  3. Otherwise, return false  4. END  **1.4.2 The isFull() Operation in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX 6  int intArray[MAX];  int front = 0;  int rear = -1;  int itemCount = 0;  bool isFull(){  return itemCount == MAX;  }  void insert(int data){  if(!isFull()) {  if(rear == MAX-1) {  rear = -1;  }  intArray[++rear] = data;  itemCount++;  }  }  int main(){  int i;  /\* insert 5 items \*/  insert(3);  insert(5);  insert(9);  insert(1);  insert(12);  insert(15);  printf("Queue: ");  for(i = 0; i < MAX; i++)  printf("%d ", intArray[i]);  printf("\n");  if(isFull()) {  printf("Queue is full!\n");  }  }  **Output**  Queue: 3 5 9 1 12 15  Queue is full!  **1.5 Queue - The isEmpty() operation**  The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.  **1.5.1 The isEmpty() operation Algorithm**  1. START  2. If the count of queue elements equals zero, return true  3. Otherwise, return false  4. END  **1.5.2 The isEmpty() operation in C**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX 6  int intArray[MAX];  int front = 0;  int rear = -1;  int itemCount = 0;  bool isEmpty(){  return itemCount == 0;  }  int main(){  int i;  printf("Queue: ");  for(i = 0; i < MAX; i++)  printf("%d ", intArray[i]);  printf("\n");  if(isEmpty()) {  printf("Queue is Empty!\n");  }  }  **Output**  Queue: 0 0 0 0 0 0  Queue is Empty! |

** Points to Remember**

* While describing queue data structure, remember the following:
* A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle.
* A queue can be represented using Array and Linked List
* A circular queue connects the end of the queue back to the front, effectively using space more efficiently.
* Apply Queue Primary Functions: Enqueue, Dequeue, Peek, isEmpty and isFull
* Check if the queue is full in Enqueue Algorithm
* Check if the queue is empty while doing Dequeue Algorithm
* Visit each element from front to rear in Traversal operations
* While Implementing Queue Basic Operations, take into consideration the following:
* Enqueue: Adds an element to the back of the queue.
* Dequeue: Removes and returns the element from the front of the queue.
* Peek (Front): Returns the front element without removing it.
* IsEmpty: Checks if the queue is empty.
* IsFull: Checks if the queue is full (for fixed-size queues).

 **Application of learning 2.4.**

Your task is to demonstrate your understanding of queue data structures. By attending the computer lab from your school, write an algorithm and a C program that creates a queue to handle customer service requests, where people wait in line to be served. The first person in line gets served first, and new requests are added at the end of the queue.

## Indicative content 2.5: Applying Stack Data Structure



****

**Duration: 7 hrs**



**Theoretical Activity 2.5.1: Description of stack data structure**

**Tasks:**

1: Answer the following questions related to stack data structure:

i. What do you understand by stack data structure in Algorithm and define following terminologies:

a. Stack

b. Push

c. Pop

d. Peek (or Top)

ii. How stack data structure is represented in algorithm?

iii. What are stack primary functions?

iv. Explain stack basic operations in algorithms.

2: Provide the answers and write on a paper

3: Present your findings to the whole class.

4: Address any questions or concerns

5: For more clarifications, read the Key readings 2.5.1 in the trainee manuals.

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 2.5.1.: Description of stack data structure**   * 1. **Definition of key terms**   2. **Stack**   A stack is a linear data structure that follows the Last In First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. Stacks are commonly used in various applications, including expression evaluation, backtracking algorithms, and managing function calls in programming (call stack).  A stack is a collection of elements with two main operations: push (to add an item) and pop (to remove an item). It allows access only to the top element.  It is named stack because it has the similar operations as the real-world stacks, for example a pack of cards or a pile of plates, etc.    **Characteristics**: It operates on the LIFO principle.   * 1. **Push**   The push operation adds an element to the top of the stack.  **Functionality**: It increases the stack size by one and places the new element above the current top.   * 1. **Pop**   The pop operation removes the top element from the stack.  **Functionalit**y: It decreases the stack size by one and returns the value of the removed element. If the stack is empty, popping raises an underflow condition.   * 1. **Peek (or Top)**   The peek operation retrieves the value of the top element without removing it from the stack.  **Functionality:** It allows inspection of the top element while maintaining the stack's state   1. **Representation of Stack in Algorithms**   A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.    A stack can be represented in algorithms in two primary ways:  **2.1 Array-Based Representation:**  Using a fixed-size array to store stack elements.  A variable (usually called top) keeps track of the index of the top element.  **Example:**  int stack[MAX\_SIZE];  int top = -1; // Indicates an empty stack  **2.2 Linked List-Based Representation:**  Using nodes where each node contains data and a pointer to the next node.  The top of the stack is represented by the head of the linked list.  **Example:**  struct Node {  int data;  struct Node\* next;  };  struct Node\* top = NULL; // Indicates an empty stack  **2.3 The primary functions of a stack**   * **Push**: To add an element to the top of the stack. * **Pop:** To remove the top element from the stack. * **Peek/Top:** To get the value of the top element without removing it. * **IsEmpty:** To check if the stack is empty. * **IsFull:** (In array implementation) To check if the stack is full.   **2.4.** **Basic Operations of Stack in Algorithms**  **2.4.1 Push Operation**  **Algorithm:**  function push (stack, value):  if stack is full:  return "Stack Overflow"  top = top + 1  stack[top] = value  **Description**: Check if the stack is full; if not, increment the top index and place the value in the stack at that index.  **2.4.2 Pop Operation**  **Algorithm:**  function pop(stack):  if stack is empty:  return "Stack Underflow"  value = stack[top]  top = top - 1  return value  **Description:** Check if the stack is empty; if not, retrieve the top value, decrement the top index, and return the value.  **2.4.3 Peek Operation**  **Algorithm:**  function peek(stack):  if stack is empty:  return "Stack is empty"  return stack[top]  **Description:** Check if the stack is empty; if not, return the value at the top index without modifying the stack.  **2.4.4. IsEmpty Operation**  Algorithm:  function isEmpty(stack):  return top == -1  **Description:** Return true if top is -1, indicating the stack is empty.  **2.4.5. IsFull Operation (for array-based stack)**  Algorithm:  function isFull(stack):  return top == MAX\_SIZE - 1  **Description**: Return true if top equals MAX\_SIZE - 1, indicating the stack is full.  **Summary**  **A stack** is a fundamental data structure that operates on the LIFO principle, supporting essential operations such as push, pop, and peek. It can be represented using arrays or linked lists, and its basic operations are crucial for various algorithms and applications in computer science. Understanding these concepts is key to effectively using stacks in programming and algorithm development |



**Practical Activity 2.5.2: Implementing stack basic operations**

**Task:**

1: Referring to the key reading 2.5.2 and perform the tasks below:

As trainee in computer system and architecture, you are asked to go to the computer lab to implement a stack data structure in Algorithm and in C program and display the output.

2: Present steps to apply stack basic operations

3: Present the findings to trainer and the whole class

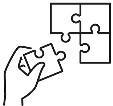
4: Ask for clarification if any

5: Perform the task in application of learning 2.5

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 2.5.2.: Implementing stack basic operations**  **1.Basic Operations on Stacks**  Complete implementation of a stack data structure including all the basic operations such as:   1. Push 2. Pop 3. Peek 4. isEmpty 5. isFull.   These are all built-in operations to carry out data manipulation and to check the status of the stack.  Stack uses pointers that always point to the topmost element within the stack, hence called as the top pointer.  **1.1 Stack Insertion: push()**  The push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.  **1.1.1 Stack Insertion push()Algorithm**  1. Checks if the stack is full.  2. If the stack is full, produces an error and exit.  3. If the stack is not full, increments top to point next  empty space.  4. Adds data element to the stack location, where top  is pointing.  5. Returns success.  **1.1.2 Stack Insertion push() in C**  #include <stdio.h>  int MAXSIZE = 8;  int stack[8];  int top = -1;  /\* Check if the stack is full\*/  int isfull(){  if(top == MAXSIZE)  return 1;  else  return 0;  }  /\* Function to insert into the stack \*/  int push(int data){  if(!isfull()) {  top = top + 1;  stack[top] = data;  } else {  printf("Could not insert data, Stack is full.\n");  }  }  /\* Main function \*/  int main(){  int i;  push(44);  push(10);  push(62);  push(123);  push(15);  printf("Stack Elements: \n");  // print stack data  for(i = 0; i < 8; i++) {  printf("%d ", stack[i]);  }  return 0;  }  **Output**  Stack Elements:  44 10 62 123 15 0 0 0  **1.2. Stack Deletion: pop()**  The pop() is a data manipulation operation which removes elements from the stack.  **1.2.1. Stack Deletion: pop() Algorithm**  1. Checks if the stack is empty.  2. If the stack is empty, produces an error and exit.  3. If the stack is not empty, accesses the data element at  which top is pointing.  4. Decreases the value of top by 1.  5. Returns success.  **1.2.2. Stack Deletion: pop() in C**  #include <stdio.h>  int MAXSIZE = 8;  int stack[8];  int top = -1;  /\* Check if the stack is empty \*/  int isempty(){  if(top == -1)  return 1;  else  return 0;  }  /\* Check if the stack is full\*/  int isfull(){  if(top == MAXSIZE)  return 1;  else  return 0;  }  /\* Function to delete from the stack \*/  int pop(){  int data;  if(!isempty()) {  data = stack[top];  top = top - 1;  return data;  } else {  printf("Could not retrieve data, Stack is empty.\n");  }  }  /\* Function to insert into the stack \*/  int push(int data){  if(!isfull()) {  top = top + 1;  stack[top] = data;  } else {  printf("Could not insert data, Stack is full.\n");  }  }  /\* Main function \*/  int main(){  int i;  push(44);  push(10);  push(62);  push(123);  push(15);  printf("Stack Elements: \n");  // print stack data  for(i = 0; i < 8; i++) {  printf("%d ", stack[i]);  }  /\*printf("Element at top of the stack: %d\n" ,peek());\*/  printf("\nElements popped: \n");  // print stack data  while(!isempty()) {  int data = pop();  printf("%d ",data);  }  return 0;  }  **Output**  Stack Elements: 44 10 62 123 15 0 0 0  Elements popped: 15 123 62 10 44  **1.3 Peek () operation**  The peek () is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.  **1.3.1 Peek () operation Algorithm**  1. START  2. return the element at the top of the stack  3. END  **1.3.2 Peek () operation in C**  #include <stdio.h>  int MAXSIZE = 8;  int stack[8];  int top = -1;  /\* Check if the stack is full \*/  int isfull(){  if(top == MAXSIZE)  return 1;  else  return 0;  }  /\* Function to return the topmost element in the stack \*/  int peek(){  return stack[top];  }  /\* Function to insert into the stack \*/  int push(int data){  if(!isfull()) {  top = top + 1;  stack[top] = data;  } else {  printf("Could not insert data, Stack is full.\n");  }  }  /\* Main function \*/  int main(){  int i;  push(44);  push(10);  push(62);  push(123);  push(15);  printf("Stack Elements: \n");  // print stack data  for(i = 0; i < 8; i++) {  printf("%d ", stack[i]);  }  printf("\nElement at top of the stack: %d\n" ,peek());  return 0;  }  **Output**  Stack Elements:  44 10 62 123 15 0 0 0  Element at top of the stack: 15  **1.4. isFull() Operation**  The isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.  **1.4.1. isFull()Operation algorithm**  1. START  2. If the size of the stack is equal to the top position of the stack,  the stack is full. Return 1.  3. Otherwise, return 0.  4. END  **1.4.2. isFull()Operation in C**  #include <stdio.h>  int MAXSIZE = 8;  int stack[8];  int top = -1;  /\* Check if the stack is full \*/  int isfull(){  if(top == MAXSIZE)  return 1;  else  return 0;  }  /\* Main function \*/  int main(){  printf("Stack full: %s\n" , isfull()?"true":"false");  return 0;  }  **Output**  Stack full: false |

** Points to Remember**

* While describing stack data structure, remember the following:
* A stack is a linear data structure that follows the Last In First Out (LIFO) principle
* The push operation adds an element to the top of the stack.
* The pop operation removes the top element from the stack.
* The peek operation retrieves the value of the top element without removing it from the stack.
* Representation of Stack in Algorithms: A stack allows all data operations at one end only.
* The primary functions of a stack are:Push, Pop, Peek/Top, IsEmpty, IsFull
* While implementing stack basic operations in C, take into consideration the following:
* Push: Inserts an element at the top of the stack.
* Pop: Removes and returns the top element.
* Peek: Retrieves the top element without removing it.
* isEmpty: Checks if the stack is empty.
* isFull: Checks if the stack is full (for fixed sizes).

**Application of learning 2.5.**

Visit Library of your school, observe all activities related to how books are borrowed and returned, then go to the computer Lab and develop a simplified Library Management System using stack operations, where books borrowed and returned follow Last In, First Out (LIFO) principles by using algorithm and C program.

## Learning outcome 2 end assessment



**Written assessment**

* 1. Read the following questions and encircle the letter with the correct answer:

1. In a linked list, the time complexity for searching an element is:
2. O(1)
3. O(n)
4. O(log n)
5. O(n^2)
6. Which of the following operations is not possible with an array?
7. Inserting an element
8. Deleting an element
9. Dynamic memory allocation
10. Accessing an element by index
11. Which of the following data structures is most efficient for implementing recursion?
12. Array
13. Stack
14. Queue
15. Linked List
16. Which of the following operations is performed first in a queue?
17. Insert
18. Peek
19. Delete
20. Dequeue
21. Which type of array has more than one index?
22. 1D Array
23. Multi-dimensional Array
24. Circular Array
25. Sparse Array
    1. Read the following questions and answer by true if correct and false if not correct
26. A stack can be implemented using both arrays and linked lists.
27. In a linked list, insertion at the beginning of the list has a time complexity of O(1).
28. An array can dynamically increase its size during runtime.
29. A queue follows the FIFO (First In First Out) principle.
    1. Differentiate linked list from an array:
    2. What are the applications of queues in computing?

**Practical assessment**

Visit your school computer lab and develop a Playlist Management System in algorithm and C program that allows users to perform a variety of operations on a dynamic playlist of songs. The system should utilize both linked lists for dynamic song management and arrays for fixed-size song operations. Additionally, implement a queue to manage song requests.

**END**

**References**

### Books

**Aho, A. V., Hopcroft, J. E., & Ullman, J. D.** (2006). Data structures and algorithms (2nd ed.). Addison-Wesley.

**Goodrich, M. T., Tamassia, R., & Mount, D. H.** (2011). Data structures and algorithms in C (2nd ed.). Wiley.

**Sears, M., & Haggard, L.** (2019). Data structures and algorithm analysis in C (4th ed.). Pearson.

**Kumar, A., & Singh, S.** (2017). Data structures and algorithms using C. Khanna Publishing.

### Web Links

GeeksforGeeks. (n.d.). Data Structures in C. Retrieved from https://www.geeksforgeeks.org/data-structures/

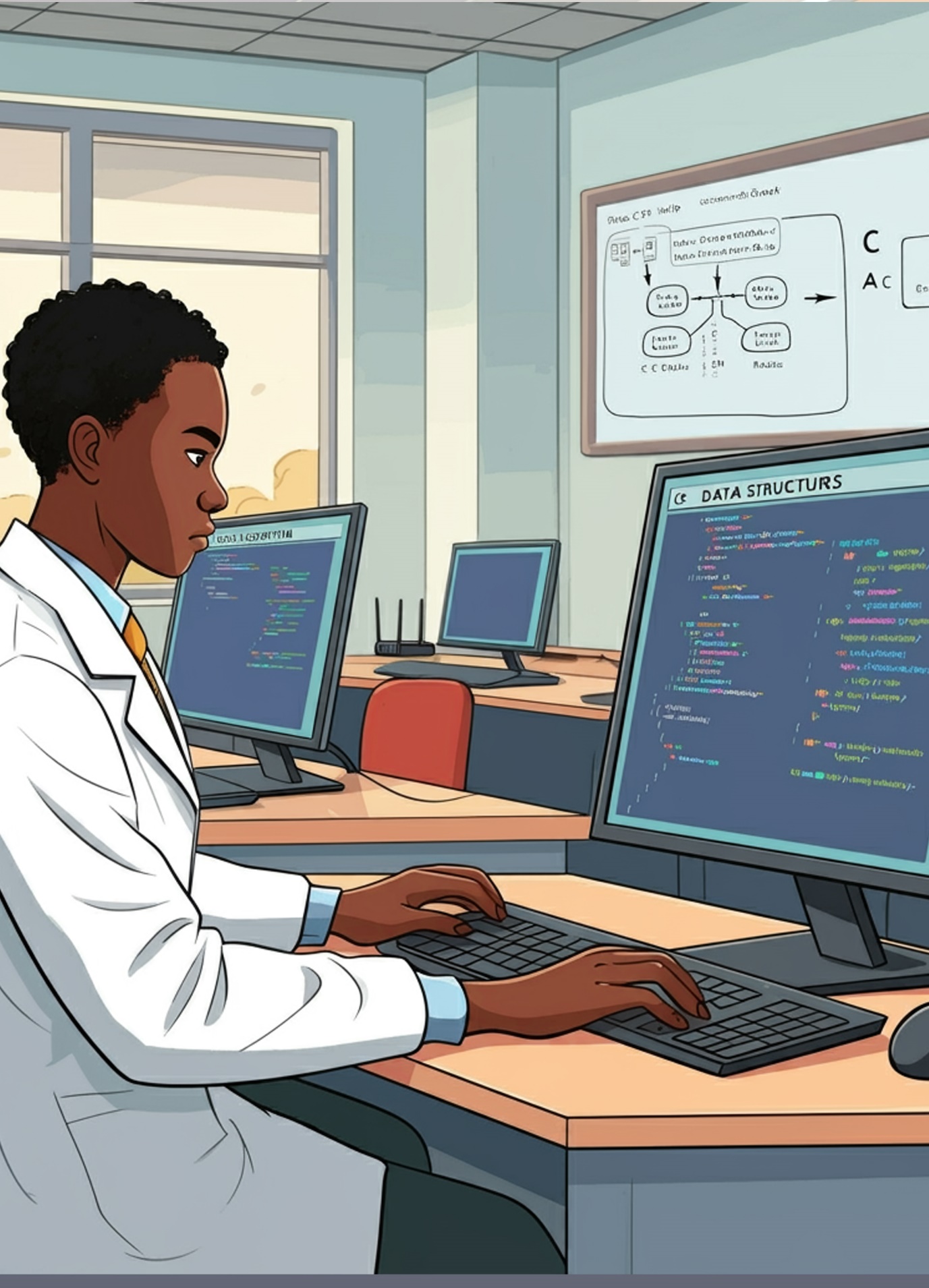
TutorialsPoint. (n.d.). C Data Structures. Retrieved from https://www.tutorialspoint.com/cprogramming/c\_data\_structures.htm

Programiz. (n.d.). C Programming - Data Structures and Algorithms. Retrieved from https://www.programiz.com/dsa

Khan Academy. (n.d.). Algorithms. Retrieved from <https://www.khanacademy.org/computing/computer-science/algorithms>

W3Schools. (n.d.). C Programming Language - Learn C with Examples. Retrieved from https://www.w3schools.com/c/

Learning Outcome 3: Apply Non-linear Data Structure



|  |
| --- |
| **Indicative contents**  **3.1 Identification of non-linear data structures concepts**  **3.2 Applying tree data structure**  **3.3 Applying graph data structure**  **3.4 Applying hash table data structure** |

## Key Competencies for Learning Outcome 3: Apply Non-linear Data Structure

|  |  |  |
| --- | --- | --- |
| **Knowledge** | **Skills** | **Attitudes** |
| * Description of non-linear data structures * Identification the operations performed on non- linear data structures * Description of tree data structure * Description of graph data structure * Identification of graph representation * Description of table data structure | * Applying tree basic operations algorithms * Implementing tree operations in C * Applying graph basic operations algorithms * Applying Shortest path algorithms * Applying hashing functions algorithms * Applying collision resolution techniques * Implementing tables data structure in C | * Having critical thinking in applying tree basic operations algorithms * Having curiosity and open-mindedness in implementing tree operations in C * Having analytical approach to Apply graph basic operations algorithms * Being collaborative and Discussion to Apply shortest path algorithms * Having problem-solving mindset to Apply hashing functions algorithms * Having collaboration and Feedback to work with others in collaboration to apply collision resolution techniques * Having critical thinking in implementing tables data structure in C |

|  |  |  |
| --- | --- | --- |
| **Duration: 40 hrs** | | |
| **Learning outcome 3 objectives**:  By the end of the learning outcome, the trainees will be able to:   1. Identify correctly non-Linear Data structures concepts clearly based on intended use. 2. Apply correctly tree basic operations algorithms based on their use cases 3. Identify clearly the Operations performed on non-linear data structures base on their use cases 4. Describe clearly tree data structure based on Non-Linear Data structures concepts 5. Implement effectively tree operations in C based on programming language standards 6. Describe correctly graph data structure based on Non-Linear Data structures concepts 7. Apply correctly graph basic operations algorithms in C based on C language standard 8. Apply effectively Shortest path algorithms based on their use cases 9. Describe correctly table data structure based on algorithm standards 10. Apply effectively hashing functions algorithms based on their use cases 11. Implement correctly tables data structure in C based on language standards | | |
| **Resources** | | |
| **Equipment** | **Tools** | **Materials** |
| * Computer | * C Compiler * Integrated Development Environment (IDE) * Text Editors * VisuAlgo | * Internet Connection |

## Indicative content 3.1: Identification of Non-Linear Data Structures Concepts



****

**Duration: 10 hrs**



**Theoretical Activity 3.1.1: Description of non-linear data structures concepts**

**Tasks:**

1: Answer to the following questions related to non-linear data structures concepts:

1. Describe non-linear data structures
2. What are Types of non-linear data structures?
3. Give and Explain Characteristics of non-linear data structures?
4. List the Operations performed on non- linear data structures?
5. Where non-linear data structures can be used?

2: Provide your answers on flipcharts/papers.

3: Present the findings/answers to the whole class

4: Follow the trainer’s expert view and ask questions if any

5: Read the key readings 3.1.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 3.1.1.: Description of non-linear data structures concepts**  **1.Description of Non-Linear Data Structures**  Non-linear data structures are types of data structures in which data elements are not arranged sequentially or linearly. Instead, they can be connected in multiple ways, allowing a more complex relationship between elements. This means that, unlike linear structures (like arrays or linked lists), elements can point to multiple other elements, creating a hierarchical or interconnected organization.  **2.Types of Non-Linear Data Structures**   * 1. **Graph Data Structure** * A graph is a non-linear data structure with a finite number of vertices and edges, and these edges are used to connect the vertices. * The graph itself is categorized based on some properties; if we talk about a complete graph, it consists of the vertex set, and each vertex is connected to the other vertexes having an edge between them. * The vertices store the data elements, while the edges represent the relationship between the vertices. * A graph is very important in various fields; the network system is represented using the graph theory and its principles in computer networks.      * 1. **Trees Data Structure** * The tree is a non-linear data structure that is comprised of various nodes. The nodes in the tree data structure are arranged in hierarchical order. * It consists of a root node corresponding to its various child nodes, present at the next level. The tree grows on a level basis, and root nodes have limited child nodes depending on the order of the tree.      * 1. **Hash table data structure**   Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.     1. **Characteristics of non-linear data structures**    1. **Hierarchical Relationship**   Non-linear data structures often have a hierarchical organization. For **example,** in a tree structure, nodes can have parent-child relationships, allowing for a clear representation of data hierarchy (e.g., organizational charts, file systems).   * 1. **Flexible Connections**   Elements in non-linear data structures can connect to multiple other elements. This flexibility allows for more complex relationships, as seen in graphs where nodes can be interconnected in various ways, representing networks, social connections, etc.   * 1. **Non-Sequential Access**   Unlike linear data structures where elements are accessed sequentially, non-linear structures allow for more complex traversal methods. For example, in trees, we can traverse in depth-first or breadth-first order, while in graphs, algorithms like Dijkstra's or A\* can be employed for searching.   * 1. **Dynamic Size**   Non-linear data structures can grow or shrink dynamically as elements are added or removed. For instance, trees and graphs can adapt to changing data without a fixed size, making them versatile for various applications.   * 1. **Complexity of Implementation**   Implementing non-linear data structures often requires more complex algorithms and handling compared to linear structures. Operations such as insertion, deletion, and traversal may involve multiple steps and considerations, particularly for balancing (in trees) or managing connections (in graphs).   * 1. **Memory Utilization**   Non-linear data structures can be more memory-efficient for certain types of data. For example, sparse graphs can be represented with adjacency lists rather than matrices, saving space when connections are limited.   * 1. **Varied Performance Characteristics**   The performance of operations (like search, insertion, and deletion) can vary widely depending on the specific non-linear structure used. For example, binary search trees provide logarithmic time complexity for search operations, while graphs may require more complex algorithms with varying time complexities.   * 1. **Support for Complex Data Relationships**   Non-linear data structures are ideal for representing complex relationships and multi-dimensional data, such as social networks, web page links, or geographical maps, where interactions between elements are not strictly linear.   1. **Common operations performed on non-linear data structures:**   Common operations performed on non-linear data structures, such as trees and graphs, include:  **4.1. Traversal**: Navigating through the structure, like in-order, pre-order, post-order traversal for trees, and depth-first search (DFS) or breadth-first search (BFS) for graphs.  **4.2. Insertion**: Adding new elements or nodes to the structure at a specific position, like inserting a child node in a tree or adding an edge in a graph.  **4.3. Deletion**: Removing elements or nodes, such as deleting a node in a tree or removing an edge from a graph.  **4.4. Searching**: Finding a specific node or value, using algorithms like DFS or BFS in graphs, or search algorithms in trees like binary search in Binary Search Trees (BST).  **4.5. Updating**: Modifying the value or data of a node or element in the structure.  **4.6. Merging/Combining**: Combining multiple trees or graphs into a single structure.  **5. Application of Non-Linear data structure**  Non-linear data structures are essential in various applications due to their ability to represent complex relationships. Here are some common use cases:   * 1. **Hierarchical Data Representation**   Example: File systems, organizational charts, and taxonomies, where data is organized in a parent-child relationship.   * 1. **Network Representation**   Example: Social networks, transportation systems, and communication networks, where entities (nodes) are connected by relationships (edges).   * 1. **Database Indexing**   Example: B-trees and other tree structures are used in databases to efficiently manage and access large amounts of data.   * 1. **Artificial Intelligence**   Example: Decision trees, game trees, and state spaces in algorithms for search and optimization, such as minimax for games.   * 1. **Routing Algorithms**   Example: Graphs are used in algorithms like Dijkstra's and A\* for finding the shortest path in navigation and network routing.   * 1. **Recommendation Systems**   Example: Graph-based models are used to represent user-item interactions in collaborative filtering and recommendation engines.   * 1. **Compilers**   Example: Abstract syntax trees (ASTs) are used to represent the structure of source code for analysis and transformation during compilation.   * 1. **Genomic Data Analysis**   Example: Trees and graphs can represent relationships in biological data, such as evolutionary trees or gene networks.   * 1. **Game Development**   Example: Trees and graphs are used to manage game levels, AI decision-making, and interactions among game entities.   * 1. **Web Structure Representation**   Example: The World Wide Web can be represented as a graph, with web pages as nodes and hyperlinks as edges, facilitating search algorithms.  These applications highlight the versatility of non-linear data structures in representing complex data relationships across different fields and industries. |



**Practical Activity 3.1.2: Applying non-linear data structure**

**Task:**

1: Referring to the key reading (3.1.2) you are requested to perform the following task about non-linear data structure:

As future technician in Computer System and Architecture you are requested to implement non-linear data structure basic operations using algorithm and C program.

2: Use instructions to apply non-linear data structure using algorithm

3: Present out the steps to apply non-linear data structure using algorithm

4: Referring to the presented steps in the task 3, apply non-linear data structure basic operations.

5: Present your work to the trainer and whole class. Ask for clarification where necessary

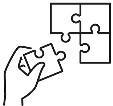
6: Read key reading 3.1.2 in trainee manual for more clarifications

7: Perform task provided in the application of learning 3.1

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 3.1.2.: Applying non-linear data structure**  **1. Binary Tree Operations**  **1.1. Insertion**  **Steps:**   1. Create a new node with the given value. 2. If the tree is empty, set the new node as the root. 3. Otherwise, use a queue to perform a level-order traversal:  * Enqueue the root node. * While the queue is not empty: * Dequeue a node. * If the left child is empty, assign the new node to the left child and return. * Otherwise, enqueue the left child. * If the right child is empty, assign the new node to the right child and return. * Otherwise, enqueue the right child.   **1.2. Deletion**  **Steps:**   1. If the tree is empty, return. 2. If the node to be deleted is the root:  * Replace it with the deepest node (last node in level-order).  1. Use a queue for level-order traversal:  * Enqueue the root node. * While the queue is not empty:   + Dequeue a node.   + If the left child is the node to be deleted:     - Replace it with the deepest node and return.   + Otherwise, enqueue the left child.   + If the right child is the node to be deleted:     - Replace it with the deepest node and return.   + Otherwise, enqueue the right child.   **1.3. Traversal**  **1.3.1. In-Order Traversal (Left, Root, Right)**  **steps**  1.If the current node is not null:   * Recursively call in-order traversal on the left child. * Visit the current node (print its value). * Recursively call in-order traversal on the right child.   **1.3.2. Pre-Order Traversal (Root, Left, Right)**  1.If the current node is not null:   * Visit the current node (print its value). * Recursively call pre-order traversal on the left child. * Recursively call pre-order traversal on the right child.   **1.3.3. Post-Order Traversal (Left, Right, Root)**  1.If the current node is not null:   * Recursively call post-order traversal on the left child. * Recursively call post-order traversal on the right child. * Visit the current node (print its value).   **2. Graph Operations**  **2.1. Depth-First Search (DFS)**  **Steps:**  1. Create a stack and a visited set.  2.Push the starting vertex onto the stack.  3.While the stack is not empty:   * Pop a vertex from the stack. * If it has not been visited:   + - Mark it as visited (print its value).     - Push all its unvisited neighbors onto the stack.   **2.2. Breadth-First Search (BFS)**  Steps:   1. Create a queue and a visited set. 2. Enqueue the starting vertex and mark it as visited. 3. While the queue is not empty:  * Dequeue a vertex from the queue (print its value). * For each neighbor of the dequeued vertex:   + - If the neighbor has not been visited:   + Mark it as visited.   + Enqueue the neighbor.   **Summary of Steps**  **Binary Tree:**  **Insert:** Create a new node, place it in the first available position using a queue.  **Delete**: Find the node to delete, replace it with the deepest node using a queue.  **In-Order**: Recursively visit left, root, right.  **Pre-Order**: Recursively visit root, left, right.  **Post-Order**: Recursively visit left, right, root.  Graph:  **DFS**: Use a stack to explore as far as possible along a branch before backtracking.  **BFS:** Use a queue to explore all neighbors at the present depth before moving on to nodes at the next depth level. |

** Points to Remember**

* While describing non-linear data structures concepts remember the following:
* Non-linear data structures are types of data structures in which data elements are not arranged sequentially or linearly.
* Types of Non-Linear Data Structures are: graph data structure, trees data structure and hash table data structure
* Characteristics of non-linear data structures are: hierarchical relationship, flexible connections, non-sequential access, dynamic size, complexity of implementation, memory utilization, varied performance characteristics and support for complex data relationships
* Common operations performed on non-linear data structures, such as trees and graphs, include: traversal, insertion, deletion, searching, updating and merging/combining
* Application of Non-Linear data structure: hierarchical data representation, network representation, database indexing, artificial intelligence, routing algorithms, recommendation systems, compilers, genomic data analysis, game development and web structure representation
* While Applying non-linear data structure, take into consideration the following:
* The insertion is done in the following way:
* Trees: Insert nodes based on value (e.g., binary search tree).
* Graphs: Add edges connecting vertices; consider directed vs. undirected.
* Deletion is done in the following way:
* Trees: Remove nodes carefully, maintaining
* Graphs: Remove edges and vertices; update connections.
* Searching is done in the following way:
* Trees: Use traversal methods (in-order, pre-order, post-order) for binary trees; search based on node values.
* Graphs: Implement search algorithms (DFS, BFS) to find nodes or paths.
* Traversal is done in the following way:
* Trees: Different methods for traversing (e.g., depth-first or breadth-first).
* Graphs: Use DFS or BFS to explore all vertices and edges
* Updating is done in the following way:
* Trees: Modify node values or structure (e.g., rebalancing in AVL trees).
* Graphs: Update edge weights or properties, especially in weighted graphs.

**Application of learning 3.1.**

In a secondary school, managing student course enrolment can be challenging due to the variety of subjects, prerequisites, and elective options. Visit your school computer lab, and use a non-linear data structure to organize this information hierarchically, making it easier for students and administrators to navigate. By utilizing trees and other non-linear structures, the school can efficiently represent relationships between core subjects, elective subjects, and prerequisites.

## Indicative content 3.2: Applying Tree Data Structure



****

**Duration: 10 hrs**



**Theoretical Activity 3.2.1: Description of tree data structure**

**Tasks:**

1: Answer to the following questions related to tree data structure:

i. Describe tree data structure

ii. Explain Tree representation

iii. Clearly explain Huffman’s tree

2: Provide your answers on flipcharts/papers.

3: Present the findings/answers to the whole class

4: Ask questions where necessary

5: Read the key readings 3.2.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 3.2.1.: Description of tree data structure**   1. **Definition of Key Terms**    1. **A Tree** is a widely-used abstract data type in computer science, which is primarily used to represent hierarchical data. It consists of nodes connected by edges, forming a structure that resembles an upside-down tree, where the root is at the top, and the branches lead down to the leaves.    2. **Node**: The fundamental part of a tree structure. A node contains data and may have references to its child nodes.    3. **Root**: The topmost node in a tree. It is the starting point from which all other nodes descend. A tree can only have one root.    4. **Edge**: The connection between two nodes in a tree.    5. **Child**: A node that descends from another node. In a tree, each node (except the root) has one parent and may have multiple children.    6. **Parent**: The node that has one or more child nodes.    7. **Leaf**: A node that has no children. It is at the bottom of the tree structure.    8. **Subtree**: A tree consisting of a node and its descendants, forming a smaller tree within the larger structure.    9. **Height**: The length of the longest path from the root to a leaf node. It determines the number of levels in a tree.    10. **Depth**: The number of edges from the root to a specific node. It indicates how far down the tree a node is.    11. **Sibling**: Nodes that share the same parent.    12. **Degree**: The number of children a node has. 2. **Basic Terminologies**    1. **Binary Tree**: A tree data structure where each node can have at most two children (referred to as left child and right child).    2. **Balanced Tree**: A tree where the height difference between the left and right subtrees of any node is minimal (commonly used in search trees).    3. **Complete Tree**: A binary tree in which all levels are completely filled except possibly the last level, which is filled from left to right.    4. **Full Tree**: A tree where every node other than the leaves has exactly two children.    5. **Traversal**: The process of visiting all the nodes in a tree in a specific order (e.g., in-order, pre-order, post-order). 3. **Types of Trees**    1. **General Tree**: A tree where any node can have any number of children.    2. **Binary Tree**:   A tree where each node has at most two children, referred to as the left child and the right child.   * 1. **Full Binary Tree**: Every node has 0 or 2 children.   2. **Complete Binary Tree**: All levels are completely filled, except possibly the last, which is filled from left to right.   3. **Perfect Binary Tree**: A binary tree where all interior nodes have two children, and all leaf nodes are at the same level.   4. **Binary Search Tree (BST)**:   A binary tree where the left child contains nodes with values smaller than the parent, and the right child contains nodes with values greater than the parent.   * 1. **AVL Tree**:   A self-balancing binary search tree where the difference between the heights of the left and right subtrees (balance factor) of any node is at most one.   * 1. **Red-Black Tree**:   A self-balancing binary search tree where each node has an additional attribute: a color (either red or black). These trees maintain balance through specific rules for adding and deleting nodes.   * 1. **Heap Tree**:   A complete binary tree where the value of the parent node is either greater than (max-heap) or smaller than (min-heap) the values of its children.   * 1. **Trie**:   Also known as a prefix tree, this tree structure is used for storing strings in a way that facilitates fast prefix lookups, commonly used in dictionaries and search engines.   * 1. **B-Tree**:   A self-balancing search tree commonly used in databases and file systems. B-trees store multiple keys in each node and allow for more than two child nodes.   * 1. **N-ary Tree**:   A generalization of a binary tree where nodes can have up to **N** children.   1. **Tree Representation**   A tree is a hierarchical data structure used to represent data in a parent-child relationship, with one root node and several levels of child nodes. Unlike arrays or linked lists, trees allow for faster searching, insertion, and deletion in many cases. Trees are used in applications such as databases, file systems, and hierarchical data storage.    **Components of a Tree:**   * + **Node**: A basic unit of a tree that stores data and may point to other nodes (children).   + **Root**: The topmost node in a tree, without a parent.   + **Child**: A node directly connected to another node (parent) through an edge.   + **Parent**: A node that has at least one child.   + **Leaf**: A node that has no children.   + **Subtree**: A section of a tree that is a tree itself, consisting of a node and its descendants.   + **Edge**: A connection between two nodes, showing the relationship between them.   + **Depth**: The number of edges from the root to a node.   + **Height**: The number of edges on the longest path from the root to a leaf.   1. **Linked Representation**:   + Each node contains the following:     - Data     - Pointer to the left child (if any)     - Pointer to the right child (if any)   struct Node {  int data;  struct Node\* left;  struct Node\* right;  };   * 1. **Array Representation (Used for complete binary trees):**   The tree is stored as an array where:  The root node is at index 0.  For any node at index i, the left child is at 2\*i + 1, and the right child is at 2\*i + 2.  **Types of Trees:**   * **Binary Tree:** Each node has at most two children (left and right). * Binary Search Tree (BST): A binary tree where the left child is smaller than the parent and the right child is larger. * **Balanced Tree:** A tree where the height of the left and right subtrees of every node differs by at most one. * **AVL Tree:** A self-balancing binary search tree. * **B-Tree:** A generalization of a binary search tree used in databases and file systems.  1. **Huffman’s Tree**   A Huffman Tree is a binary tree used in the Huffman coding algorithm for data compression. It assigns variable-length codes to characters based on their frequency in the input data. More frequent characters get shorter codes, while less frequent characters get longer codes. Huffman coding is widely used in file compression algorithms like ZIP, GZIP, and media formats such as MP3.  **Huffman Tree Construction Process:**  **Frequency Counting:** Determine the frequency of each character in the data to be encoded.  **Priority Queue:** Insert each character into a priority queue (or min-heap) where the priority is determined by the frequency of the character.  **Tree Building:**  Remove the two nodes with the smallest frequencies from the queue.  Create a new internal node with these two nodes as children, and the frequency of this new node is the sum of the two smallest frequencies.  Insert the new node back into the queue.  Repeat until there is only one node left, which becomes the root of the Huffman Tree.  Assign Codes: Traverse the tree to assign binary codes to each character. Left edges are assigned 0, and right edges are assigned 1.  **Example:**  Consider the following character frequencies:  A: 5  B: 9  C: 12  D: 13  E: 16  F: 45  The Huffman tree construction follows these steps:  Build a min-heap from the characters.  Combine the two smallest nodes (A: 5 and B: 9) into a new node (14).  Continue combining nodes until the tree is built.  Huffman Tree Example:  \*  / \  \* F (45)  / \  E (16) \*  / \  D (13) C (12)  / \  A (5) B (9)   * Huffman Codes:   + F: 1   + E: 01   + D: 001   + C: 000   + B: 0001   + A: 0000   The characters are now represented with variable-length binary codes based on their frequency, making the encoding efficient for compression.  **Advantages of Huffman Tree:**   * **Efficient Compression**: Reduces the total number of bits required to encode a message. * **Prefix Property**: No code is a prefix of any other, preventing ambiguity during decoding.   **Applications:**   * **File Compression**: ZIP, GZIP, and other formats. * **Multimedia Encoding**: MP3, JPEG, PNG. |



**Practical Activity 3.2.2: Applying tree data structure**

**Task:**

1: Referring to the activities (3.2.2) you are requested to perform the following task about applying tree data structure:

As a trainee in computer system and architecture, visit your school’s computer lab and Apply tree basic operations algorithms and Implement tree operations in C

2: Apply instructions to apply tree basic operations algorithms

3: Present out the steps to implement tree operations in C

4: Referring to the presented steps in the task 3, implement tree operations in C

5: Present your work to the trainer and whole class.

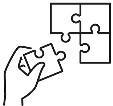
6: Ask for clarification where necessary

7: Perform task provided in the application of learning 3.2

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 3.2.2.: Applying tree data structure**   * 1. **Introduction**   Tree data structures have a wide range of applications. Let's walk through basic operations on trees, such as insertion, traversal, and searching, using algorithms and then implement them in C.   * 1. **Tree Insertion Algorithm**   Inserting a node into a binary tree typically involves finding the correct location for the new node based on the tree's rules (e.g., for a Binary Search Tree).   * + 1. **Binary Tree Insertion Algorithm**: * If the tree is empty, the new node becomes the root. * If the tree is not empty, recursively insert the new node in either the left or right subtree depending on the comparison with the root.   **Algorithm**:  function insertNode(root, value):  if root is null:  root = new Node(value)  return root  if value < root.data:  root.left = insertNode(root.left, value)  else:  root.right = insertNode(root.right, value)  return root   * 1. **Tree Traversal Algorithms**      1. **In-order Traversal** (Left, Root, Right):   Traverse the left subtree.  Visit the root node.  Traverse the right subtree.  **Algorithm**:  function inOrderTraversal(root):  if root is not null:  inOrderTraversal(root.left)  print(root.data)  inOrderTraversal(root.right)   * + 1. **Pre-order Traversal** (Root, Left, Right):   Visit the root node.  Traverse the left subtree.  Traverse the right subtree.  **Algorithm**:  function preOrderTraversal(root):  if root is not null:  print(root.data)  preOrderTraversal(root.left)  preOrderTraversal(root.right)   * + 1. **Post-order Traversal** (Left, Right, Root):   Traverse the left subtree.  Traverse the right subtree.  Visit the root node.  **Algorithm**:  function postOrderTraversal(root):  if root is not null:  postOrderTraversal(root.left)  postOrderTraversal(root.right)  print(root.data)   * 1. **Search Algorithm in a Binary Search Tree**   Searching for a value in a binary search tree (BST) involves comparing the value with the root and traversing left or right based on the value.  **Algorithm**:  function searchNode(root, key):  if root is null or root.data == key:  return root  if key < root.data:  return searchNode(root.left, key)  return searchNode(root.right, key)   1. **Implementing Tree Operations in C**   Below is a C implementation of the basic tree operations, including insertion, traversal, and searching.   * 1. **Tree Node Structure in C**   #include <stdio.h>  #include <stdlib.h>  // Define structure for tree node  struct Node {  int data;  struct Node\* left;  struct Node\* right;  };  // Create a new tree node  struct Node\* createNode(int value) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = value;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }   * 1. **Insertion Function**   // Insert a node into the binary search tree  struct Node\* insertNode(struct Node\* root, int value) {  if (root == NULL) {  return createNode(value); // If tree is empty, create a new node  }  if (value < root->data) {  root->left = insertNode(root->left, value); // Insert in the left subtree  } else {  root->right = insertNode(root->right, value); // Insert in the right subtree  }  return root;  }   * 1. **Tree Traversal Functions**      1. **In-order Traversal** (Left, Root, Right):   void inOrderTraversal(struct Node\* root) {  if (root != NULL) {  inOrderTraversal(root->left);  printf("%d ", root->data);  inOrderTraversal(root->right);  }  }   * + 1. **Pre-order Traversal** (Root, Left, Right):   void preOrderTraversal(struct Node\* root) {  if (root != NULL) {  printf("%d ", root->data);  preOrderTraversal(root->left);  preOrderTraversal(root->right);  }  }   * + 1. **Post-order Traversal** (Left, Right, Root):   void postOrderTraversal(struct Node\* root) {  if (root != NULL) {  postOrderTraversal(root->left);  postOrderTraversal(root->right);  printf("%d ", root->data);  }  }   * 1. **Search Function**   // Search for a node with a specific key in the binary search tree  struct Node\* searchNode(struct Node\* root, int key) {  if (root == NULL || root->data == key) {  return root;  }  if (key < root->data) {  return searchNode(root->left, key);  }  return searchNode(root->right, key);  }   * 1. **Main Function to Test Operations**   int main() {  struct Node\* root = NULL;  // Insert nodes into the tree  root = insertNode(root, 50);  root = insertNode(root, 30);  root = insertNode(root, 70);  root = insertNode(root, 20);  root = insertNode(root, 40);  root = insertNode(root, 60);  root = insertNode(root, 80);  // Perform different traversals  printf("In-order traversal: ");  inOrderTraversal(root);  printf("\n");  printf("Pre-order traversal: ");  preOrderTraversal(root);  printf("\n");  printf("Post-order traversal: ");  postOrderTraversal(root);  printf("\n");  // Search for a value  int key = 40;  struct Node\* foundNode = searchNode(root, key);  if (foundNode != NULL) {  printf("Node with value %d found in the tree.\n", key);  } else {  printf("Node with value %d not found in the tree.\n", key);  }  return 0;  }  **Explanation:**   1. **Insertion**: The insertNode function adds a new node in the appropriate position based on the binary search tree properties. 2. **Traversal**:    * In-order: Prints the nodes in ascending order.    * Pre-order: Visits the root before its subtrees.    * Post-order: Visits the root after its subtrees. 3. **Search**: The searchNode function looks for a specific value in the tree, returning the node if found, or NULL otherwise. |

** Points to Remember**

* While describing tree data structure, remember the following:
* A Tree is a widely-used abstract data type in computer science, which is primarily used to represent hierarchical data.
* Tree is represented in a hierarchical data structure used to represent data in a parent-child relationship, with one root node and several levels of child nodes.
* A Huffman Tree is a binary tree used in the Huffman coding algorithm for data compression. Huffman coding is widely used in file compression algorithms like ZIP, GZIP, and media formats such as MP3.
* Height shows the length of the longest path from the root to a leaf.
* Depth represents the length of the path from the root to a specific node.
* Subtree shows A tree formed from a node and its descendants.
* Each character is represented by a unique binary code, with more frequent characters having shorter codes.
* The tree is optimal for minimizing the total length of encoded data.
* While applying tree data structure, take into consideration the following:
* Insertion of nodes is done based on value (left for smaller, right for larger).
* While Searching do the following:
* Use a recursive or iterative approach to find a value.
* Compare the target with the current node and traverse left or right accordingly.

**Application of learning 3.2.**

Imagine you're developing a file management system for an operating system. In this system, files and directories are structured in a hierarchical tree format. The root directory has subdirectories, and each subdirectory can contain files or other subdirectories. Visit the computer lab and use a tree data structure to represent this hierarchical relationship, where each directory or file is a node, the root directory is the root node, and subdirectories or files are the child nodes.

## Indicative content 3.3: Applying graph data structure



****

**Duration: 10 hrs**



**Theoretical Activity 3.3.1: Description of graph data structure**

**Tasks:**

1: Answer to the following questions related to graph data structure:

* 1. What do you understand by graph data structure?
  2. What are types of graph data structure?

2: Provide your answers on flipcharts/papers.

3: Present the findings/answers to the whole class

5: Ask questions if any

6: Read the key readings 3.3.1. for more clarifications

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 3.3.1.: Description of graph data structure**   * 1. **Definition of Key terms**   2. **A graph data structure** is a collection of nodes (or vertices) connected by edges, used to represent relationships between entities. Here are some key points to understand about graph data structures:   3. **Vertices (Nodes):** The basic units of a graph, representing entities such as objects, people, or locations.   4. **Edges:** The connections between vertices, which can be directed (indicating a one-way relationship) or undirected (indicating a two-way relationship).   Edges can also have weights, representing costs or distances associated with traveling from one vertex to another.   1. **Types of Graphs:**    1. **Directed Graph**: Edges have a direction, showing a one-way relationship.   All the edges for a graph need to be directed to call it a directed graph or digraph. All the edges of a directed graph or digraph have a direction that will start from one vertex and end at another    Fig: Directed graph   * 1. **Undirected Graph**: Edges have no direction, indicating a mutual connection.   A graph is called a non-directed graph if all the edges present between any graph nodes are non-directed. By non-directed edges, we mean the edges of the graph that cannot be determined from the node it is starting and at which node it is ending. All the edges for a graph need to be non-directed to call it a non-directed graph. All the edges of a non-directed graph don't have any direction.    Fig:Undirected Graph   * 1. **Weighted Graph**: Edges have weights, useful for representing distances or costs.      * 1. **Unweighted Graph**: All edges are treated equally.   **Connected Graph:** There is a path between every pair of vertices.  For a graph to be labelled as a connected graph, there must be at least a single path between every pair of the graph's vertices. In other words, we can say that if we start from one vertex, we should be able to move to any of the vertices that are present in that particular graph, which means there exists at least one path between all the vertices of the graph.    Fig: connected graph   * 1. **Disconnected Graph:** Some vertices do not have paths connecting them.   A graph is said to be a disconnected graph where there does not exist any path between at least one pair of vertices. In other words, we can say that if we start from any one of the vertices of the graph and try to move to the remaining present vertices of the graph and there exists not even a single path to move to that vertex, then it is the case of the disconnected graph. If any one of such a pair of vertices doesn't have a path between them, it is called a disconnected graph.    Fig: Disconnected graph  **Tree:** A connected graph with no cycles.  **Directed Acyclic Graph (DAG):** A directed graph with no cycles, often used for task scheduling. |



**Practical Activity 3.3.2: Applying graph data structure**

**Task:**

1: Referring to the previous theoretical activities (3.3.1) you are requested to perform the following task about applying graph data structure:

As a trainee in Computer system and Architecture, you are requested to visit your school computer lab to Apply graph basic operations algorithms and Apply Shortest path algorithms using algorithm and C program

2: Apply instructions to apply graph and shortest path

3: Present out the steps to apply graph and Shortest path using algorithm and C program

4: Referring to the presented steps in the task 3, apply graph data structure using algorithm and C program.

5: Present your work to the trainer and whole class. Ask for clarification where necessary

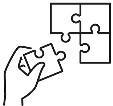
6: Read key reading 3.3.2 for more clarifications

7: Perform task provided in the application of learning 3.3

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 3.3.2.: Applying graph data structure**  Here’s a step-by-step algorithm to implement basic graph operations, including functions for adding and removing vertices and edges, as well as implementations of BFS and DFS algorithms.  **1. Algorithm Steps**  **Step 1**: Define the Graph Structure  Graph Representation:  Use an adjacency matrix or an adjacency list to represent the graph.  Define a structure to hold the number of vertices and the adjacency representation.  **Step 2:** Initialize the Graph  **Create Graph:**  Initialize the graph structure with zero vertices.  **Step 3:** Add Vertex Function  Function: addVertex(graph):  Check if the number of vertices is less than the maximum allowed.  Increment the vertex count.  **Step 4:** Add Edge Function  Function: addEdge(graph, vertex1, vertex2):  Ensure both vertices exist in the graph.  Update the adjacency representation to reflect the new edge.  **Step 5:** Remove Edge Function  Function: removeEdge(graph, vertex1, vertex2):  Check if the edge exists.  Update the adjacency representation to remove the edge.  **Step 6**: Remove Vertex Function  Function: removeVertex(graph, vertex):  Remove all edges connected to the vertex.  Decrease the vertex count.  **Step 7**: BFS Algorithm  Function: bfs(graph, startVertex):  Initialize a queue and a visited list.  Begin from the starting vertex, mark it as visited, and enqueue it.  While the queue is not empty:  Dequeue a vertex, print it, and enqueue all unvisited adjacent vertices.  **Step 8**: DFS Algorithm  Function: dfs(graph, startVertex, visited):  Mark the starting vertex as visited and print it.  For each adjacent vertex, if it hasn’t been visited, recursively call DFS on that vertex.  **Step 9:** Test the Implementation  **Main Function:**  Create a graph and add vertices (e.g., A, B, C, D, E).  Add edges (e.g., A-B, A-C, B-D, C-D, C-E).  Perform BFS and DFS starting from specific vertices.  Remove a vertex and/or edge and test the traversal methods again.  **Example Pseudocode**  1. InitializeGraph():  Create an empty graph structure  2. addVertex(graph):  if graph.numVertices < MAX\_VERTICES:  graph.numVertices += 1  3. addEdge(graph, vertex1, vertex2):  if vertex1 and vertex2 exist in graph:  graph.adjacency[vertex1][vertex2] = 1  graph.adjacency[vertex2][vertex1] = 1 // Undirected  4. removeEdge(graph, vertex1, vertex2):  if edge exists:  graph.adjacency[vertex1][vertex2] = 0  graph.adjacency[vertex2][vertex1] = 0  5. removeVertex(graph, vertex):  for each vertex in graph:  removeEdge(graph, vertex, adjacentVertex)  graph.numVertices -= 1  6. bfs(graph, startVertex):  Initialize queue and visited list  Enqueue startVertex, mark as visited  while queue is not empty:  Dequeue vertex, print it  for each adjacentVertex of vertex:  if not visited[adjacentVertex]:  Enqueue adjacentVertex, mark as visited  7. dfs(graph, startVertex, visited):  Mark startVertex as visited, print it  for each adjacentVertex of startVertex:  if not visited[adjacentVertex]:  dfs(graph, adjacentVertex, visited)  8. Main():  graph = InitializeGraph()  for each vertex in ['A', 'B', 'C', 'D', 'E']:  addVertex(graph)  addEdge(graph, 0, 1) // A-B  addEdge(graph, 0, 2) // A-C  addEdge(graph, 1, 3) // B-D  addEdge(graph, 2, 3) // C-D  addEdge(graph, 2, 4) // C-E  print("BFS from A:")  bfs(graph, 0)  print("DFS from B:")  visited = [false] \* MAX\_VERTICES  dfs(graph, 1, visited)   * 1. **Traversal Algorithms:**   **1.1.1. Depth First Search (DFS):** This algorithm traverses the graph by going as deep as possible before backtracking.  **1.1.2. Breadth First Search (BFS):** This algorithm explores the graph layer by layer starting from the root node.  **BFS and DFS for Graph Traversal in C**  // BFS Example  #include <stdio.h>  #include <stdlib.h>  #define MAX 100  int adj[MAX][MAX]; // Adjacency matrix for graph  int visited[MAX]; // Visited array to keep track of visited nodes  void BFS(int start, int n) {  int queue[MAX], front = 0, rear = -1, i;  visited[start] = 1;  queue[++rear] = start;  while (front <= rear) {  start = queue[front++];  printf("%d ", start);  for (i = 1; i <= n; i++) {  if (adj[start][i] == 1 && visited[i] == 0) {  queue[++rear] = i;  visited[i] = 1;  }  }  }  }  // DFS Example  void DFS(int v, int n) {  int i;  visited[v] = 1;  printf("%d ", v);  for (i = 1; i <= n; i++) {  if (adj[v][i] == 1 && visited[i] == 0)  DFS(i, n);  }  }   * 1. **Shortest Path Algorithms:**   These algorithms find the shortest path from one vertex to another in a graph.   * 1. **Dijkstra’s Algorithm:**   Finds the shortest path from a single source vertex to all other vertices in a graph with non-negative weights.  Example:  #include <stdio.h>  #include <limits.h>  #define V 9  int minDistance(int dist[], int sptSet[]) {  int min = INT\_MAX, min\_index;  for (int v = 0; v < V; v++)  if (sptSet[v] == 0 && dist[v] <= min)  min = dist[v], min\_index = v;  return min\_index;  }  void dijkstra(int graph[V][V], int src) {  int dist[V];  int sptSet[V];  for (int i = 0; i < V; i++)  dist[i] = INT\_MAX, sptSet[i] = 0;  dist[src] = 0;  for (int count = 0; count < V - 1; count++) {  int u = minDistance(dist, sptSet);  sptSet[u] = 1;  for (int v = 0; v < V; v++)  if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])  dist[v] = dist[u] + graph[u][v];  }  printf("Vertex \t Distance from Source\n");  for (int i = 0; i < V; i++)  printf("%d \t\t %d\n", i, dist[i]);  }   * 1. **Warshall’s Algorithm (All Pairs Shortest Path Algorithm):**   Finds the shortest path between all pairs of vertices using dynamic programming.  **Example**:  #include <stdio.h>  #define V 4  #define INF 99999  void floydWarshall(int graph[V][V]) {  int dist[V][V], i, j, k;  for (i = 0; i < V; i++)  for (j = 0; j < V; j++)  dist[i][j] = graph[i][j];  for (k = 0; k < V; k++) {  for (i = 0; i < V; i++) {  for (j = 0; j < V; j++) {  if (dist[i][j] > dist[i][k] + dist[k][j])  dist[i][j] = dist[i][k] + dist[k][j];  }  }  }  printf("Shortest distances between every pair of vertices:\n");  for (i = 0; i < V; i++) {  for (j = 0; j < V; j++) {  if (dist[i][j] == INF)  printf("%7s", "INF");  else  printf("%7d", dist[i][j]);  }  printf("\n");  }  }   * 1. **Minimum Spanning Tree (MST) Algorithms:**   **Prim’s Algorithm** and **Kruskal’s Algorithm** are commonly used to find the minimum spanning tree of a graph.  **Prim’s Algorithm:**  Finds the minimum spanning tree by growing a single tree one edge at a time.  **Example**:  #define V 5  int minKey(int key[], int mstSet[]) {  int min = INT\_MAX, min\_index;  for (int v = 0; v < V; v++)  if (mstSet[v] == 0 && key[v] < min)  min = key[v], min\_index = v;  return min\_index;  }  void primMST(int graph[V][V]) {  int parent[V];  int key[V];  int mstSet[V];  for (int i = 0; i < V; i++)  key[i] = INT\_MAX, mstSet[i] = 0;  key[0] = 0;  parent[0] = -1;  for (int count = 0; count < V - 1; count++) {  int u = minKey(key, mstSet);  mstSet[u] = 1;  for (int v = 0; v < V; v++)  if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])  parent[v] = u, key[v] = graph[u][v];  }  printf("Edge \tWeight\n");  for (int i = 1; i < V; i++)  printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);  }   * 1. **Kruskal’s Algorithm:**   Finds the minimum spanning tree by sorting all edges and adding them one by one to the MST.  **Example:**  #include <stdio.h>  int find(int parent[], int i) {  if (parent[i] == i)  return i;  return find(parent, parent[i]);  }  void unionFind(int parent[], int x, int y) {  int xroot = find(parent, x);  int yroot = find(parent, y);  parent[xroot] = yroot;  }  void kruskalMST(int graph[][3], int V, int E) {  int parent[V];  for (int i = 0; i < V; i++)  parent[i] = i;  for (int i = 0; i < E; i++) {  int u = graph[i][0], v = graph[i][1], w = graph[i][2];  int x = find(parent, u);  int y = find(parent, v);  if (x != y) {  printf("%d - %d: %d\n", u, v, w);  unionFind(parent, x, y);  }  }  } |

** Points to Remember**

* While describing graph data structure, remember the following:
* A graph data structure is a collection of nodes (or vertices) connected by edges, used to represent relationships between entities
* Types of Graphs: directed graph ,undirected graph, weighted graph , unweighted graph , disconnected graph
* In Unweighted Graph, all edges are treated equally (no weights).
* Cyclic Graph Contains at least one cycle (a path that starts and ends at the same vertex).
* Acyclic Graph Contains no cycles.
* While applying graph data structure, take into consideration the following:
* Binary Tree Operations are Insert, Delete, Traversals
* Traversals is performed in different styles such as: In-Order, Pre-Order and Post-Order
* Depth-First Search (DFS) Use a stack (can be implemented using recursion) as Data Structure
* Breadth-First Search (BFS) Use a queue as Data Structure

 **Application of learning 3.3.**

As the future technician in Computer System and Architecture, you are requested to attend the Computer Lab to Create a simple graph using an adjacency list and implement the algorithm and a C program to find the shortest route from one intersection to another. Calculate the shortest path from intersection A to E in a predefined graph.

## Indicative content 3.4: Applying Hash Table Data Structure



****

**Duration: 10 hrs**



**Theoretical Activity 3.4.1: Description of hash table data structure**

**Tasks:**

1: Answer to the following questions related to the description of hash table data structure:

1. Define the following key terms:
   * + 1. Hash tables
       2. Bucket
       3. Hash functions
       4. Load factor
       5. Collision resolution

2: Write your answers on flipcharts/papers.

3: Present the findings/answers to the trainer and whole class

4: Ask questions where necessary.

5: Read the key readings 3.4.1. in trainee manual

|  |
| --- |
| Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStock**Key readings 3.4.1.: Description of hash table data structure**  **1. Definition of Key Terms**   * 1. **Table**: In data structures, a table is a data organization system used to store and retrieve data efficiently. It's commonly implemented as a **hash table** or **hash map**, which allows fast access to data using a key-value pair.   2. **Hash Tables**   **Hash Table**: A **hash table** (or **hash map**) is a data structure that stores key-value pairs. Each key is mapped to a value by using a **hash function** that calculates an index in an array (table), allowing for quick access to the value associated with the key. Hash tables are efficient for searching, inserting, and deleting operations.  A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.  For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:  Hash(key)= index;  When we pass the key in the hash function, then it gives the index.  Hash(john) = 3;  The above example adds the john at the index 3.   * 1. **Bucket**   A **bucket** refers to a specific position in the hash table where one or more keys may be stored. When multiple keys hash to the same index (due to a **collision**), all the associated keys and values are grouped together in that bucket. The bucket can be implemented as a list, tree, or another data structure.   * 1. **Hash Functions**   A **hash function** is an algorithm that takes a key as input and produces a fixed-size integer, which corresponds to an index in the hash table. The goal of the hash function is to distribute the keys uniformly across the table to minimize collisions. A good hash function ensures that different keys produce different indices (minimizing **collisions**).  Example of a simple hash function:  hash(key) = key % table\_size  Here, the key is divided by the size of the table, and the remainder is the index.  **Drawback of Hash function**  A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.  **Hashing**  Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is O(n), and O(logn) in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.  In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.  The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:  **Index = hash(key)**     * 1. **Load Factor** * The **load factor** of a hash table is the ratio of the number of elements in the table to the number of available buckets. It is a measure of how full the hash table is and impacts performance. A higher load factor increases the chances of collisions, which slows down operations.   Formula:  Load Factor = Number of elements / Number of buckets   * Many hash table implementations resize (grow) the table when the load factor exceeds a certain threshold (e.g., 0.75) to maintain efficient performance.   1. **Collision Resolution** * A **collision** occurs when two different keys hash to the same index in the hash table. Since a hash table relies on unique indices to store key-value pairs, collisions must be handled efficiently. * **Collision Resolution**: There are several strategies to resolve collisions in a hash table:   + **Chaining**: In this method, each bucket contains a linked list of entries that hash to the same index. If multiple keys have the same hash value, they are stored in the same bucket but linked together. This method is simple to implement and works well in practice.   + **Open Addressing**: In open addressing, all elements are stored directly in the hash table (no external lists). If a collision occurs, the algorithm searches for the next available empty bucket using a probing sequence.     - **Linear Probing**: Move to the next available bucket sequentially.     - **Quadratic Probing**: Move based on a quadratic function (e.g., 1^2, 2^2, 3^2) to find the next bucket.     - **Double Hashing**: Use a second hash function to determine the probing sequence. |



**Practical Activity 3.4.2: Applying hash table data structure**

**Task:**

1: Refer to the key reading 3.4.2 and perform the following task:

As a Computer system and Architecture trainee, go to the computer lab and Apply hashing functions algorithms, collision resolution techniques and Implement tables data structure in C

2: Apply instructions about hash table data structure using algorithm and C programming language

3: Present out the steps to apply hash table data structure and collision resolution techniques using algorithm and C program

4: Present your work to the trainer and whole class.

5: Read key reading 3.4.2 and ask for clarification where necessary

6: Perform the task provided in application of learning 3.4.

|  |
| --- |
| **Refer Instruction Manual Booklet Black Iconvector Illustration Isolated On  White Background Label Eps10 Stock Illustration - Download Image Now -  iStockKey readings 3.4.2 : Applying hash table data structure**   * 1. **Hashing Functions Algorithm**   A **hashing function** is a function that takes an input (or "key") and returns an integer that is used as the index in a hash table. A good hash function should minimize collisions and distribute the keys uniformly across the hash table.  Here’s an algorithm for a basic hashing function using the modulo operator:  int hashFunction(int key, int tableSize) {  return key % tableSize; // Modulo operation to determine index  }  This hash function returns the remainder when dividing the key by the table size, ensuring the index falls within the bounds of the table.   * 1. **Collision Resolution Techniques**   Collisions occur when two keys hash to the same index. There are two common methods for resolving collisions: **chaining** and **open addressing**.  **2.1. Chaining:**   * In chaining, each index of the hash table points to a linked list, where multiple key-value pairs can be stored if they hash to the same index.   **Chaining Algorithm**:  #include <stdio.h>  #include <stdlib.h>  // Node for linked list  struct Node {  int key;  struct Node\* next;  };  // Hash table using chaining  struct Node\* hashTable[10];  // Hash function  int hashFunction(int key) {  return key % 10;  }  // Insert function with chaining collision resolution  void insert(int key) {  int index = hashFunction(key);  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->key = key;  newNode->next = hashTable[index];  hashTable[index] = newNode;  }  // Search function  int search(int key) {  int index = hashFunction(key);  struct Node\* current = hashTable[index];  while (current != NULL) {  if (current->key == key) {  return 1; // Key found  }  current = current->next;  }  return 0; // Key not found  }  // Display function to show hash table  void display() {  for (int i = 0; i < 10; i++) {  struct Node\* current = hashTable[i];  printf("Index %d: ", i);  while (current != NULL) {  printf("%d -> ", current->key);  current = current->next;  }  printf("NULL\n");  }  }  int main() {  // Insert values  insert(15);  insert(25);  insert(35);  insert(44);  insert(9);  // Display hash table  display();  // Search for values  printf("\nSearch 25: %d\n", search(25)); // Key found  printf("Search 50: %d\n", search(50)); // Key not found  return 0;  }  **Explanation**:   * **Insert**: Each new key is inserted into a linked list at the appropriate index based on the hash function. * **Search**: The algorithm traverses the linked list to find the key. * **Display**: Shows the current structure of the hash table and how the keys are chained.   **2.2. Open Addressing:**   * In open addressing, all elements are stored within the hash table itself, and when a collision occurs, the algorithm searches for the next available space using a probing technique.   **Example1:**  **Linear Probing Algorithm**:  #include <stdio.h>  #include <stdlib.h>  #define TABLE\_SIZE 10  int hashTable[TABLE\_SIZE];  // Hash function  int hashFunction(int key) {  return key % TABLE\_SIZE;  }  // Insert function using linear probing  void insert(int key) {  int index = hashFunction(key);  // Linear probing  while (hashTable[index] != -1) {  index = (index + 1) % TABLE\_SIZE;  }  hashTable[index] = key;  }  // Search function  int search(int key) {  int index = hashFunction(key);  // Linear probing  while (hashTable[index] != -1) {  if (hashTable[index] == key) {  return 1; // Key found  }  index = (index + 1) % TABLE\_SIZE;  }  return 0; // Key not found  }  // Display function to show hash table  void display() {  for (int i = 0; i < TABLE\_SIZE; i++) {  if (hashTable[i] == -1) {  printf("Index %d: Empty\n", i);  } else {  printf("Index %d: %d\n", i, hashTable[i]);  }  }  }  int main() {  // Initialize hash table to -1 (indicating empty slots)  for (int i = 0; i < TABLE\_SIZE; i++) {  hashTable[i] = -1;  }  // Insert values  insert(15);  insert(25);  insert(35);  insert(44);  insert(9);  // Display hash table  display();  // Search for values  printf("\nSearch 25: %d\n", search(25)); // Key found  printf("Search 50: %d\n", search(50)); // Key not found  return 0;  }  **Explanation**:   * **Linear Probing**: If a collision occurs (the index is already occupied), the next available slot is found by incrementing the index. * **Insert and Search**: Both use linear probing to handle collisions.   **Example2:**  // Simple example of a hash function and collision resolution using chaining in C  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 10  typedef struct Node {  char \*key;  int value;  struct Node \*next;  } Node;  Node \*hashTable[TABLE\_SIZE];  // Simple hash function using modulo  int hashFunction(char \*key) {  int sum = 0;  for (int i = 0; key[i] != '\0'; i++) {  sum += key[i];  }  return sum % TABLE\_SIZE;  }  // Insert key-value pair into the hash table  void insert(char \*key, int value) {  int index = hashFunction(key);  Node \*newNode = (Node \*)malloc(sizeof(Node));  newNode->key = strdup(key);  newNode->value = value;  newNode->next = hashTable[index];  hashTable[index] = newNode;  }  // Search for a key in the hash table  int search(char \*key) {  int index = hashFunction(key);  Node \*node = hashTable[index];  while (node != NULL) {  if (strcmp(node->key, key) == 0) {  return node->value;  }  node = node->next;  }  return -1; // Key not found  }  int main() {  insert("Alice", 20);  insert("Bob", 25);  insert("Charlie", 30);  printf("Value for Alice: %d\n", search("Alice"));  printf("Value for Bob: %d\n", search("Bob"));  return 0;  }  **In this example:**  • **Hash function** calculates an index by summing the ASCII values of characters in the key.  • **Collision resolution** is handled by chaining, where multiple key-value pairs can be stored in a linked list at the same index. ****Implementation of Hash Tables in C**** This combines the previously discussed techniques into a complete hash table implementation.  You can choose between **chaining** (storing multiple keys in a bucket via linked lists) or **open addressing** (storing all elements in the table and probing when collisions occur).  Both approaches provide a practical solution for implementing hash tables in **C**, offering fast search, insert, and delete operations.  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 10  typedef struct Node {  int key;  char value[256];  struct Node \*next;  } Node;  typedef struct HashTable {  Node \*buckets[TABLE\_SIZE];  } HashTable;  // Hash function  unsigned int hash(int key) {  return key % TABLE\_SIZE;  }  // Create a new hash table  HashTable \*create\_table() {  HashTable \*table = malloc(sizeof(HashTable));  for (int i = 0; i < TABLE\_SIZE; i++) {  table->buckets[i] = NULL;  }  return table;  }  // Insert key-value pair  void insert(HashTable \*table, int key, const char \*value) {  unsigned int index = hash(key);  Node \*new\_node = malloc(sizeof(Node));  new\_node->key = key;  strcpy(new\_node->value, value);  new\_node->next = table->buckets[index];  table->buckets[index] = new\_node;  }  // Search for a value by key  char \*search(HashTable \*table, int key) {  unsigned int index = hash(key);  Node \*current = table->buckets[index];  while (current != NULL) {  if (current->key == key) {  return current->value;  }  current = current->next;  }  return NULL; // Not found  }  // Free the hash table  void free\_table(HashTable \*table) {  for (int i = 0; i < TABLE\_SIZE; i++) {  Node \*current = table->buckets[i];  while (current != NULL) {  Node \*temp = current;  current = current->next;  free(temp);  }  }  free(table);  }  // Example usage  int main() {  HashTable \*table = create\_table();  insert(table, 1, "Value1");  insert(table, 11, "Value2"); // Collision with key 1  printf("Key 1: %s\n", search(table, 1));  printf("Key 11: %s\n", search(table, 11));  free\_table(table);  return 0;  } |

** Points to Remember**

* While describing hash table data structure, remember the following:
* Hash Table: A data structure that stores key-value pairs, providing fast access using a hash function.
* A bucket refers to a specific position in the hash table where one or more keys may
* be stored. Each index in the hash table is called a bucket. Multiple keys can hash to the same bucket due to collisions.
* Hash Function: is an algorithm that takes a key as input and produces a fixed-size
* integer, which corresponds to an index in the hash table. It Maps keys to indices in

the table. A good hash function minimizes collisions.

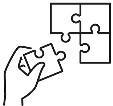
* Load Factor: A measure of how full the hash table is. High load factors increase

collisions

* Collision resolution in a hash table refers to the techniques used to handle situations when two or more keys produce the same index (or "hash")
* While applying hash table data structure, take into consideration the following:
* Hashing Functions: Converts a key into an index using a function like the module

operator (key % tableSize), ensuring uniform distribution of keys.

* Collision Resolution:
  + Chaining: Uses linked lists to store multiple elements at the same index.
  + Open Addressing: Uses probing techniques like linear probing to find the next available slot when a collision occurs.
* Hash Table Implementation: Combines hashing with collision resolution to createefficient data structures for fast insertion, searching, and deletion in C.

 **Application of learning 3.4.**

At school, we need to implement a simple student database system using a hash table. Each student will be identified by a unique student ID (an integer), and we will store the student's name and age. You are request to go to the computer lab and use hash table data structure in algorithm and C program to do the following : insertion, searching, and deletion of student records.

## Learning outcome 3 end assessment



**Written assessment**

* 1. **Read the following questions and answer true if it is correct and false if it is not correct**

1. In a non-linear data structure, elements are arranged sequentially.
2. In a graph, vertices are connected by edges.
3. Hash tables resolve collisions using methods like chaining and open addressing.
4. In a binary tree, each node can have up to two children.
5. A tree can have multiple roots.
   1. **Match the following terms with their correct descriptions:**

| **Terms** | **Descriptions** | **Answer** |
| --- | --- | --- |
| 1. Binary Tree | A. A collection of nodes connected by edges | **1. …….** |
| 2. Leaf Node | B. A tree where each node has at most two children | **2. ……..** |
| 3. Depth-First Search (DFS) | C. FIFO operation | **3. ……..** |
| 4. Hash Function | D. A node with no children | **4. ………** |
| 5.Weighted Graph | E. An algorithm to traverse a graph | **5………..** |
| 6. Collision | F. A graph where edges have associated weights | **6. ……….** |
| 7. Graph | G. Pictorial representation of algorithm | **7. ……….** |
| 8. Hash Table | H. Occurs when two keys hash to the same index | **8. ……….** |
|  | I. A data structure used for key-value pairs |  |
|  | J. A structure that maps keys to values |  |

* 1. **Choose the best answer among the following and cycle the letter corresponding to it:**
     + 1. In a tree data structure, the topmost node is known as:

1. Leaf
2. Root
3. Child
4. Branch
   * + 1. What does an adjacency matrix represent in a graph?
5. The degree of each vertex
6. The connections between vertices
7. The weights of edges
8. The paths in the graph
   * + 1. What is the primary use of a table data structure?
9. To store key-value pairs
10. To manage hierarchical relationships
11. To represent binary trees
12. To handle graph algorithms
    * + 1. What is a characteristic of non-linear data structures?
13. They follow a sequential order
14. They have a hierarchical relationship
15. They allow random access
16. They cannot store large amounts of data
    1. What are the primary differences between tree and graph data structures, and in what scenarios would you choose one over the other?

**Practical assessment**

You are developing a university course management system that utilizes various data structures and algorithms to manage student enrolments, course prerequisites, and instructor assignments. Use algorithm and C program to do the following :

* + - * 1. Create a hash table for student records with collision handling.
        2. Implement graph algorithms for course relationships and apply Dijkstra's for pathfinding.
        3. Manage course schedules and instructor assignments through structured tables.

**END**

**References**

### Books

**Sedgewick, R., & Wayne, K.** (2011). Algorithms (4th ed.). Addison-Wesley.

**Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). Introduction to algorithms (3rd ed.). MIT Press.

**Knuth, D. E.** (1997). The art of computer programming, Volume 1: Fundamental algorithms (3rd ed.). Addison-Wesley.

**Morrison, D. F.** (2004). Computer algorithms: Introduction to design and analysis (3rd ed.). Prentice Hall.

**Rosen, K. H.** (2012). Discrete mathematics and its applications (7th ed.). McGraw-Hill.

**Bhagat, A.** (2013). Mastering data structures through C. BPB Publications.

### Web Links

GeeksforGeeks. (n.d.). Non-linear Data Structures. Retrieved from https://www.geeksforgeeks.org/non-linear-data-structures/

TutorialsPoint. (n.d.). Trees in Data Structures. Retrieved from https://www.tutorialspoint.com/data\_structures\_algorithms/tree\_data\_structure.htm

Programiz. (n.d.). Data Structures and Algorithms in C. Retrieved from https://www.programiz.com/dsa

Khan Academy. (n.d.). Graph Theory and Algorithms. Retrieved from <https://www.khanacademy.org/computing/computer-science/algorithms>

W3Schools. (n.d.). C Programming Language - Learn C with Examples. Retrieved from https://www.w3schools.com/c/

**October 2024**

