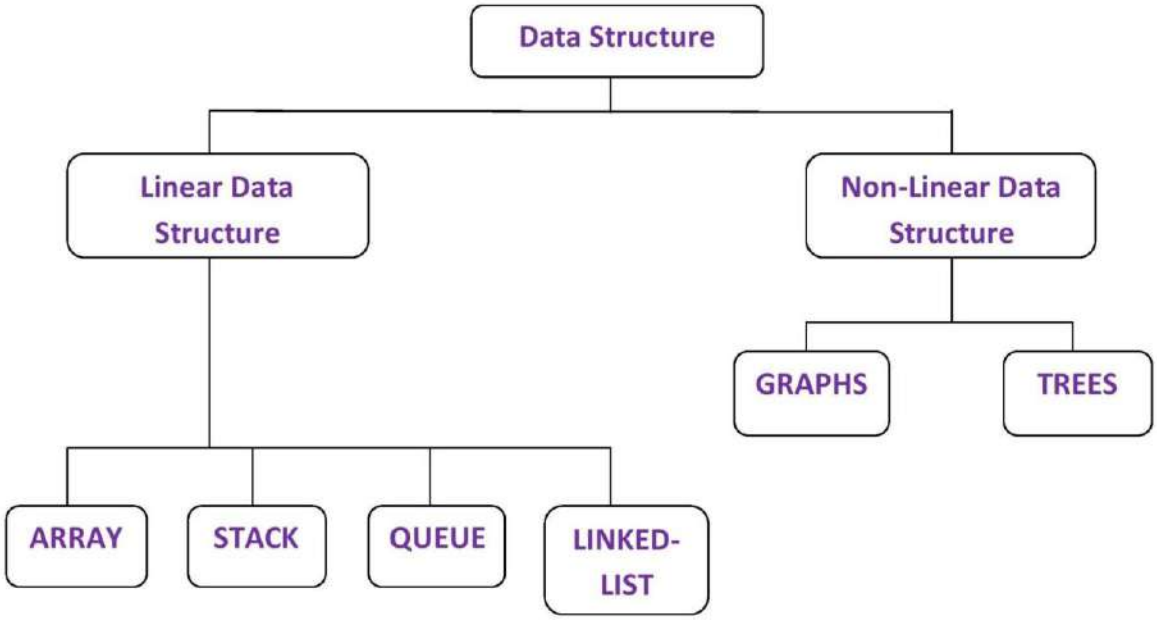
SYLLABUS

1. Introduction to Linear Data Structures:
2. Definition and importance of linear data structures,
3. Abstract data types (ADTs) and their implementation,
4. Overview of time and space complexity analysis for linear data structures.
5. Searching Techniques:
6. Linear & Binary Search,
7. Sorting Techniques:
8. Bubble sort,
9. Selection sort,
8. Insertion Sort.

1.) Definition of Data Structures

A **data structure** is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It defines the relationship between the data and the operations that can be performed on the data.



Importance of Data Structures

Data structures play a crucial role in computer science. They provide:

- **Efficient Data Management**: Reducing processing time and improving performance.
- **Data Organization**: Logical arrangement for easier understanding and access.
- **Data Abstraction**: Hiding implementation details for focused data manipulation.
- **Reusability**: Common structures can be reused across applications.
- **Algorithm Optimization**: Efficient algorithms depend on appropriate data structures.

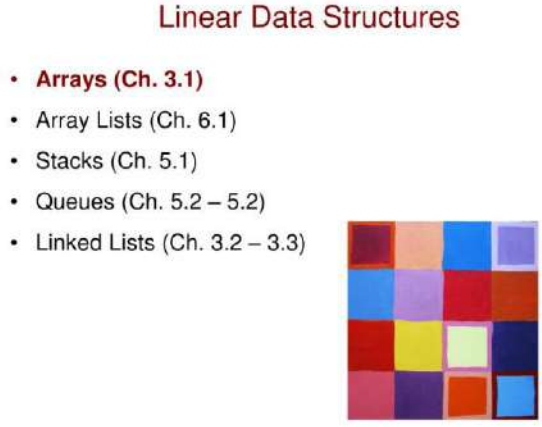
Classification of Data Structures

1. **Linear Data Structures**:
    - **Array**: Contiguous memory storage for homogeneous elements.
    - **Linked List**: Dynamic allocation with nodes linked by pointers.
    - **Queue**: **FIFO** structure for tasks like scheduling.
    - **Stack**: **LIFO** structure for managing function calls.
2. **Non-Linear Data Structures**:
    - **Tree**: Hierarchical structure with nodes having multiple child nodes.
    - **Graph**: Nodes connected by edges, representing relationships.

Understanding data structures is fundamental for efficient algorithms and software performance. They are the building blocks for solving computational problems! □□

2.) Introduction To Linear Data Structures

Certainly! Let's delve into the world of **Linear Data Structures**. These structures play a crucial role in computer science, providing a systematic way to organize, manage, and manipulate data in a **sequential** or **linear** fashion. Here are the key points:



1. **Definition of Linear Data Structures**:
    - Linear Data Structures arrange data elements **sequentially** or **linearly**.
    - Each element has a **previous** and **next** adjacent element, except for the first and last elements.
    - These structures serve as the fundamental building blocks for data organization.
2. **Characteristics of Linear Data Structures**:
    - **Sequential Organization**:
        - Elements are arranged one after the other.
        - Each element has a unique predecessor (except the first) and a unique successor (except the last).
    - **Order Preservation**:
        - The order in which elements are added is preserved.
        - The first element added is the first to be accessed or removed, and the last element added is the last to be accessed or removed.
    - **Fixed or Dynamic Size**:
        - Linear data structures can have either fixed or dynamic sizes.
        - Arrays typically have a fixed size, while structures like linked lists, stacks, and queues can dynamically grow or shrink.
    - **Efficient Access**:
        - Accessing elements within a linear structure is typically efficient.
        - For example, arrays offer constant-time access using their index.
3. **Common Linear Data Structures**:
    - **Arrays**:
        - A collection of elements stored in contiguous memory locations.
        - Homogeneous elements (same data type).
        - Provides constant-time access based on index.
    - **Linked Lists**:

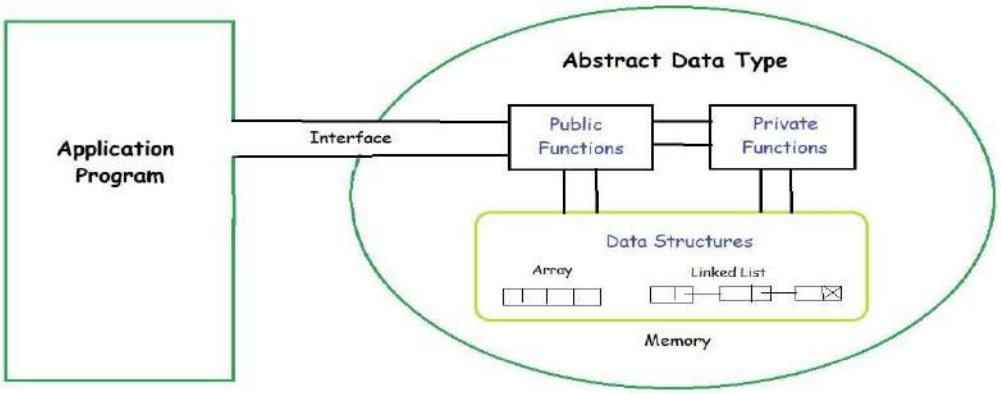
- A collection of nodes, each containing an element and a reference to the next node.
- Flexible size and dynamic allocation.

- **Stacks**:
  - Elements follow a **Last-In-First-Out (LIFO)** order.
  - Useful for managing function calls, undo operations, etc.
- **Queues**:
  - Elements follow a **First-In-First-Out (FIFO)** order.
  - Used for scheduling tasks, handling requests, etc.

Remember, linear data structures are essential tools for organizing and manipulating data in a **sequential** manner, making them indispensable in various computing tasks! □□

3.) Abstract Data Types (ADTs)

- ADTs are high-level data structures that define a set of operations without specifying their implementation details.
- They provide a **logical blueprint** for how data should behave, allowing programmers to focus on functionality rather than implementation.
- ADTs are essential for software design and modular programming.



Common Abstract Data Types

1. **Stack**:
    - **Operations**:
        - `push(item)`: Adds an item to the top of the stack.
        - `pop()`: Removes and returns the top item.
        - `peek()`: Returns the top item without removing it.
    - **Implementation**:
        - Can be implemented using arrays or linked lists.
        - Follows the **Last-In-First-Out (LIFO)** principle.
2. **Queue**:
    - **Operations**:
        - `enqueue(item)`: Adds an item to the end of the queue.
        - `dequeue()`: Removes and returns the front item.
        - `peek()`: Returns the front item without removing it.
    - **Implementation**:
        - Can be implemented using arrays or linked lists.
        - Follows the **First-In-First-Out (FIFO)** principle.
3. **Linked List**:
    - **Operations**:
        - `insert(item)`: Adds an item to the list.
        - `delete(item)`: Removes an item from the list.
        - `search(item)`: Finds an item in the list.
    - **Implementation**:
        - Consists of nodes linked together.
        - Allows dynamic allocation and flexible size.
4. **Tree**:
    - **Operations**:

- `insert(item)`: Adds an item to the tree.
- `delete(item)`: Removes an item from the tree.
- `search(item)`: Finds an item in the tree.

- **Implementation**:
  - Hierarchical structure with nodes and edges.
  - Examples: Binary trees, AVL trees, etc.

Implementation Details

- The actual implementation of ADTs depends on the programming language and specific requirements.
- Choose an appropriate data structure (e.g., arrays, linked lists) based on efficiency and ease of use.

Remember, ADTs provide a powerful abstraction layer, allowing developers to create robust and modular software systems! □□

4.) TIME COMPLEXITY & SPACE COMPLEXITY

Absolutely! Here's an overview of time and space complexity analysis for linear data structures with C code examples:

Time Complexity:

Time complexity refers to the amount of time it takes an algorithm using a specific data structure to execute, based on the input size (typically denoted by `n`). We express it using Big O notation, which represents the upper bound of an algorithm's execution time. Here are common time complexities for operations in linear data structures:

- **Constant Time (O(1))**: Operations that take a fixed amount of time regardless of input size (e.g., accessing an element by index in an array).
- **Linear Time (O(n))**: Operations where the execution time grows proportionally to the input size (e.g., iterating through an entire array or linked list).

Space Complexity:

Space complexity refers to the amount of additional memory an algorithm needs besides the input data itself. It's also expressed using Big O notation, considering the extra space required as the input size grows. Common space complexities for linear data structures:

- **Constant Space (O(1))**: Algorithms that use a fixed amount of extra space regardless of input size (e.g., searching an array).
- **Linear Space (O(n))**: Algorithms that require extra space proportional to the input size (e.g., creating a new array to store all elements during insertion in an array).

Examples:

1. Array:

- **Time Complexity:**
  - Access: O(1) (using index)
  - Traversal: O(n) (iterating through all elements)
  - Search (linear search): O(n) (worst case, may need to check all elements)
  - Insertion (at the end): O(1) (often) - amortized constant time, needs to shift elements if array is full

- Deletion (from the end): O(1) (similar to insertion)
- Insertion/Deletion (in the middle): O(n) (needs to shift elements)

- **Space Complexity:** O(n) (stores all elements contiguously in memory)

C

```c
#include <stdio.h>


int main() {
  int arr[5] = {1, 2, 3, 4, 5};


  // Access: O(1)
  printf("First element: %d\n", arr[0]);


  // Traversal: O(n)
  for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");


  return 0;
}
```

2. Linked List:

- **Time Complexity:**

- Access (by index): O(n) (need to traverse to find the element)

- Traversal: O(n) (follow pointers)
- Search: O(n) (similar to traversal)
- Insertion (at the beginning): O(1) (update head pointer)
- Deletion (at the beginning): O(1) (update head pointer)
- Insertion/Deletion (anywhere else): O(n) (need to traverse to find the position)

- **Space Complexity:** O(n) (each node stores data and a pointer to the next node)

C

```
#include <stdio.h>
#malloc.h>


struct Node {
  int data;
  struct Node* next;
};


int main() {
  struct Node* head = NULL; // Initially empty list


  // Insertion at the beginning: O(1)
  struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
  newNode->data = 10;
  newNode->next = head;
```

```
  head = newNode;


  // Traversal: O(n)
  struct Node* temp = head;
  while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
  }
  printf("\n");


  return 0;
}
```

By understanding time and space complexity, you can choose the appropriate data structure for your program's needs. Remember, arrays offer faster random access (O(1)) but can be slow for insertions/deletions in the middle (O(n)). Linked lists excel at insertions/deletions (O(1) at the beginning/end) but have slower random access (O(n)).

5.) Searching Techniques

1. Linear Search

- **Concept:** Linear search iterates through a list of items one by one, comparing each item with the target value you're searching for.

- **Time Complexity:** O(n) in the worst and average case. This means the search time grows linearly with the number of items (n) in the list.
- **Advantages:**
  - Simple to implement.
  - Works with any type of data (sorted or unsorted).
- **Disadvantages:**
  - Can be slow for large datasets.

C code example:

C

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int target) {
  for (int i = 0; i < n; i++) {
    if (arr[i] == target) {
      return i; // Index of the target element
    }
  }
  return -1; // Target not found
}

int main() {
  int arr[] = {2, 3, 4, 10, 40};
  int target = 10;
```

```
  int index = linearSearch(arr, sizeof(arr) / sizeof(arr[0]),
target);

  if (index != -1) {
    printf("Element found at index %d\n", index);
  } else {
    printf("Element not found\n");
  }

  return 0;
}
```

2. Binary Search

- **Concept:** Binary search only works on **sorted** lists. It repeatedly divides the search area in half by comparing the target value with the middle element. This eliminates half of the list from further search in each iteration.
- **Time Complexity:** O(log n) in the best and average case. This means the search time grows logarithmically with the number of items, making it significantly faster for larger datasets.
- **Advantages:**
  - Much faster than linear search for sorted lists.
- **Disadvantages:**
  - Requires the list to be sorted beforehand (additional time complexity).

- Not suitable for unsorted lists.

C code example:

C

```c
#include <stdio.h>

int binarySearch(int arr[], int low, int high, int target) {
  if (low > high) {
    return -1; // Target not found
  }

  int mid = low + (high - low) / 2;

  if (arr[mid] == target) {
    return mid; // Target found at the middle element
  } else if (arr[mid] < target) {
    return binarySearch(arr, mid + 1, high, target); // Search in
the right half
  } else {
    return binarySearch(arr, low, mid - 1, target); // Search in
the left half
  }
}

int main() {
```

```
  int arr[] = {2, 3, 4, 10, 40};
  int n = sizeof(arr) / sizeof(arr[0]);
  int target = 10;

  // Sort the array before searching (assuming it's initially
unsorted)
  // You can use a sorting algorithm here (e.g., bubble sort,
insertion sort)
  // ...

  int index = binarySearch(arr, 0, n - 1, target);

  if (index != -1) {
    printf("Element found at index %d\n", index);
  } else {
    printf("Element not found\n");
  }

  return 0;
}
```

Sorting Techniques

1. Bubble Sort

- **Concept:** Bubble sort repeatedly iterates through the list, compares adjacent elements, and swaps them if they are in the wrong order. The largest element "bubbles" up to the end with each pass.
- **Time Complexity:** O(n^2) in the worst and average case. This means the sorting time grows quadratically with the number of items, making it inefficient for large datasets.
- **Advantages:**
  - Simple to understand and implement.
- **Disadvantages:**
  - Very slow for large datasets.

C code example:

C

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n - 1; ++i) {
      for (int j = 0; j < n - i - 1; ++j) {
         if (arr[j] > arr[j + 1]) {
            // Swap arr[j] and arr[j + 1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
```

```
            arr[j + 1] = temp;
          }
        }
    }
}
int main() {
    int myArray[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(myArray) / sizeof(myArray[0]);
    printf("Original array: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", myArray[i]);
    }
    printf("\n");
    bubbleSort(myArray, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", myArray[i]);
    }
    printf("\n");
    return 0;
}
```

2. Selection Sort

- **Concept:** Selection sort iterates through the list, finding the minimum (or maximum) element and swapping it with the first (or last) element. Then, it repeats this process for the remaining unsorted part of the list.
- **Time Complexity:** O(n^2) in the worst and average case, similar to bubble sort.
- **Advantages:**
  - Slightly faster than bubble sort in practice for some datasets.
  - Easier to implement than some other sorting algorithms.
- **Disadvantages:**
  - Still inefficient for large datasets due to the quadratic complexity.

C code example:

C

```c
#include <stdio.h>


void selectionSort(int arr[], int n) {
  for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < n; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
```

```
    }
    if (minIndex != i) {
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
    }
  }
}
```

3. Insertion Sort

- **Concept:** Insertion sort iterates through the list, inserting each element at its correct position in the already sorted sub-list. It builds the final sorted list one by one.
- **Time Complexity:**
  - O(n^2) in the worst case (similar to bubble and selection sort).
  - O(n) in the best case (already sorted list).
- **Advantages:**
  - More efficient than bubble and selection sort for partially sorted lists or small datasets.
  - Stable sorting algorithm (preserves the order of equal elements).
- **Disadvantages:**
  - Can still be slow for large, random datasets.

C code example:

C

```c
#include <stdio.h>


void insertionSort(int arr[], int n) {
  for (int i = 1; i < n; i++) {
    int key = arr[i];
    int j = i - 1;

    // Shift elements of the sorted sub-list to make space for
the key
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}
```

Choosing the right sorting technique depends on the specific needs of your program.

Consider factors like:

- **Size of the data:** Bubble sort, selection sort, and insertion sort are not ideal for very large datasets due to their quadratic complexity.

- **Sortedness of the data:** Insertion sort performs well for partially sorted data.
- **Stability:** If preserving the order of equal elements is important, use insertion sort.

For large datasets, consider more efficient sorting algorithms like quicksort, merge sort, or heap sort.
