UNIT – I

1. Define data structure. Explain different types of data structure

Definition of data structures:

Data structures are specialized formats for organizing, processing, retrieving, and storing data. They enable efficient access and modification of data. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.

Classification or types of Data Structures:

Data structures can be classified in several ways based on their characteristics and the type of operations they support. Here are the primary classifications:

1. **Primitive vs. Non-Primitive Data Structures**
   - **Primitive Data Structures:** These are basic structures that directly operate upon the machine instructions. Examples include integers, floats, characters, and pointers.
   - **Non-Primitive Data Structures:** These are more complex structures that use primitive data structures. They can be further divided into:
     - Linear Data Structures
     - Non-Linear Data Structures
2. **Linear vs. Non-Linear Data Structures:**
   - **Linear Data Structures:** Elements are arranged in a sequential manner. Each element is connected to its previous and next element, making traversal straightforward. Examples include:
     - Arrays
     - Linked Lists
     - Stacks
     - Queues
   - **Non-Linear Data Structures:** Elements are not arranged in a sequential manner. Each element can connect to multiple elements, forming a hierarchical relationship. Examples include:
     - Trees
     - Graphs
3. **Static vs. Dynamic Data Structures**
   - Static Data Structures: These have a fixed size, and memory allocation is determined at compile-time. An example is an array.

- Dynamic Data Structures: These can grow or shrink in size as needed, with memory allocation done at run-time. Examples include linked lists, stacks, and queues implemented using linked lists.

4. **Homogeneous vs. Heterogeneous Data Structures**
   - Homogeneous Data Structures: All elements are of the same type. Examples include arrays and linked lists where all elements are of the same data type.
   - Heterogeneous Data Structures: Elements can be of different types. An example is a record (or struct in C/C++), where different fields can have different data types.

5. **Persistent vs. Ephemeral Data Structures**
   - Persistent Data Structures: Data structures that preserve previous versions of themselves when modified, allowing access to historical versions. These are common in functional programming.
   - Ephemeral Data Structures: Data structures that do not preserve previous versions when modified. Most traditional data structures fall into this category.

2. What is Abstract Data Type (ADT)? How does an Abstract Data Type (ADT) simplify the use of data structures, and why is it important in software development?

Ans: **An Abstract Data Type (ADT):**

An Abstract Data Type (ADT) is a theoretical concept used in computer science to describe a data type solely by its behavior (operations and properties) rather than by its implementation. An ADT specifies what operations are available and what they do, without specifying how these operations are implemented.

Key Characteristics of ADTs:

1. **Abstraction**: ADTs provide a layer of abstraction between the data structures and the operations performed on them. Users of an ADT need to know what operations are available and what they do, but not how the operations are implemented.
2. **Encapsulation**: The details of the data representation and the implementation of the operations are hidden from the user. This means that the internal structure of the data cannot be accessed directly.

3. **Interface**: ADTs are defined by a set of operations (interface) that can be performed on the data. For example, the interface of a stack ADT might include operations like push, pop, and isEmpty.

How ADTs Simplify the Use of Data Structures:

1. **Modularity**: ADTs allow developers to divide the program into modules. Each module can be developed, tested, and maintained independently.
2. **Reusability**: Once an ADT is defined, it can be reused in different parts of a program or in different programs without modification.
3. **Maintainability**: If the implementation of an ADT needs to be changed, the interface remains the same. This means that the rest of the program that uses the ADT does not need to change.
4. **Clarity**: By focusing on the operations and their behavior rather than the implementation details, ADTs help in making the code more readable and understandable.

Importance of ADTs in Software Development:

1. **Separation of Concerns**: ADTs help in separating the 'what' from the 'how'. This separation allows developers to focus on the high-level design of the software without worrying about low-level implementation details.
2. **Improved Collaboration**: In a team setting, ADTs allow different team members to work on different parts of a program independently. One team member can work on the implementation of an ADT while others use the ADT in their parts of the program.
3. **Flexibility and Adaptability**: If a better algorithm or data structure is discovered, it can be implemented behind the ADT interface without affecting the rest of the program.
4. **Testing and Debugging**: ADTs make it easier to test and debug code since the implementation of the data structure can be tested independently from the code that uses it.

Examples of Common ADTs:

1. **List**: A sequence of elements with operations like add, remove, and get.
2. **Stack**: A collection with last-in, first-out (LIFO) semantics. Operations include push, pop, and peek.

3. **Queue**: A collection with first-in, first-out (FIFO) semantics. Operations include enqueue and dequeue.
4. **Map (or Dictionary)**: A collection of key-value pairs with operations like put, get, and remove.
5. **Set**: A collection of unique elements with operations like add, remove, and contains.

By using ADTs, developers can create robust, maintainable, and scalable software systems.

3. Can you provide a detailed explanation of the quick sort algorithm?

Ans: Quick Sort is an efficient, in-place, comparison-based sorting algorithm. It is widely used due to its average-case time complexity of **O(nlogn),** though its worst-case performance is **$O(n^2)$.** However, with good pivot selection strategies, it performs well in most scenarios.

1. Basic Concept of Quick Sort

Quick Sort follows the **divide-and-conquer** paradigm:

- **Divide**: It selects a pivot element from the array and partitions the other elements into two sub-arrays: one with elements smaller than the pivot and one with elements greater than the pivot.
- **Conquer**: The sub-arrays are then sorted recursively using the same approach.
- **Combine**: Since the array is sorted in place, no explicit combination step is required.

2. Steps Involved in Quick Sort

1. **Choose a Pivot Element**:
    - The pivot element is used to partition the array into two parts. Common strategies for selecting the pivot include:
        - First element
        - Last element
        - Random element
        - Median-of-three (median of the first, middle, and last element)
2. **Partitioning**:
    - Partitioning means rearranging the array so that:
        - Elements less than the pivot are on the left.
        - Elements greater than the pivot are on the right.

- The pivot element is placed in its correct sorted position.

3. **Recursive Sorting**:
    - After partitioning, the pivot element is in its correct place. The algorithm then recursively applies the same process to the left and right sub-arrays.

3. Pseudo code for Quick Sort

```
function quickSort(arr, low, high):
    if low < high:
        # Partition the array and get the pivot index
        pi = partition(arr, low, high)
        # Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
function partition(arr, low, high):
    # Choose the pivot element (e.g., the last element)
    pivot = arr[high]
    i = low - 1  # Index of the smaller element
        for j = low to high - 1:
        # If the current element is smaller than the pivot
        if arr[j] < pivot:
            i = i + 1
            swap arr[i] with arr[j]
        # Swap the pivot element with the element at index i+1
    swap arr[i + 1] with arr[high]
        return i + 1  # Return the partitioning index
```

4. Example of Quick Sort

Consider sorting the array:

arr=[9,7,5,11,12,2,14,3,10,6]

Step 1 (Initial Array):

[9,7,5,11,12,2,14,3,10,6]

1. **Step 2 (Choose Pivot)**:

   Let's choose the last element, 6, as the pivot.

2. **Step 3 (Partitioning)**:

   After partitioning with pivot 6, the array becomes:

   [5,2,3,6,12,14,11,9,10,7]

   The pivot 6 is now in its correct position (index 3).

3. **Step 4 (Recursive Quick Sort)**:

   Now, quick sort is recursively applied to the sub-arrays to the left and right of 6:

   - Left sub-array: [5,2,3]
   - Right sub-array: [12,14,11,9,10,7]

4. **Step 5 (Sort Left Sub-array)**:

   Choose pivot 3 for the left sub-array [5,2,3]. After partitioning:

   [2,3,5]

   3 is in its correct position, and no further recursion is needed for this sub-array.

5. **Step 6 (Sort Right Sub-array)**:

   Choose pivot 7 for the right sub-array [12,14,11,9,10,7. After partitioning:

   [6,9,7,10,12,14,11]

   Repeat the process until the array is fully sorted.

4. Can you interpret and explain the *merge sort algorithm*, including an example to illustrate its steps?

Ans: Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves; recursively sorts them, and then merges the sorted halves to produce the sorted array.

Merge Sort Steps:

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort each half.
3. **Combine**: Merge the two sorted halves into a single sorted array.

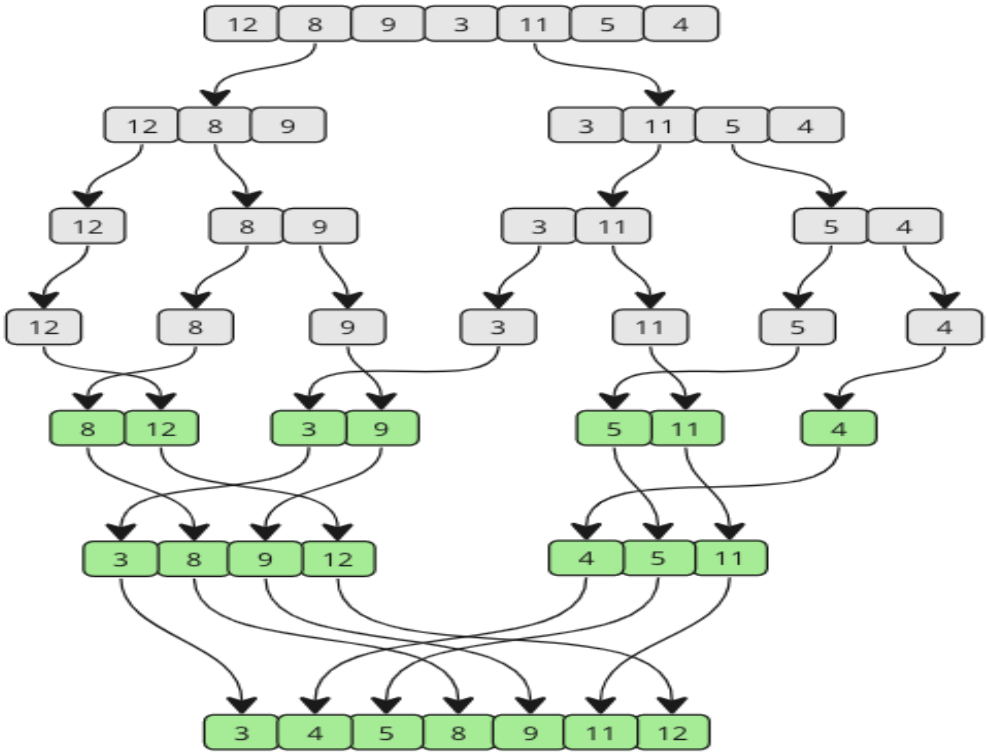
Merge Sort Algorithm (Pseudocode)

1. **MergeSort(arr[], l, r)**
   - If l < r
     - Find the middle point to divide the array into two halves:
       - m = (l + r) / 2
     - Call mergeSort for the first half:
       - mergeSort(arr, l, m)

- Call mergeSort for the second half:
  - mergeSort(arr, m + 1, r)
- Merge the two halves sorted in step 2 and 3:
  - merge(arr, l, m, r)

2. **Merge(arr[], l, m, r)**
   1. Create two pointers, one for each sorted half.
   2. Initialize an empty temporary array to hold the merged result.
   3. Compare the elements at the pointers of the two halves:
      - Copy the smaller element into the temporary array.
      - Move the pointer of the sublist with the smaller element forward.
   4. Repeat step 3 until one of the sublists is empty.
   5. Copy the remaining elements from the non-empty sublist to the temporary array.
   6. Copy the elements back from the temporary array to the original list.

Example: The following diagram explains how merge sort splits an array into smaller and smaller pieces until each sub-array contains only one element. As the merging process begins, values from each sub-array are compared, and the lowest value comes first. This ensures that each merge results in a sorted sub-array, and the array is fully sorted by the time all merges are completed



Short questions

1. Explain the difference between linear and non-linear data structures with examples.

Ans: The difference between **linear** and **non-linear** data structures lies in how the data elements are organized and accessed.

Linear Data Structures:

In linear data structures, the elements are arranged sequentially, one after the other, in a single level. Each element has a unique predecessor (except the first) and a unique successor (except the last), allowing for traversal in a single direction (or both, in some cases).

- **Characteristics:**
    - Easy to implement and access.
    - Elements are stored consecutively, making traversal straightforward.
    - Memory utilization is simple as it occupies a contiguous memory location (in some cases).
- **Examples:**
    - **Array:** A collection of elements stored in contiguous memory locations. Accessing an element requires an index. Example: int arr[5] = {1, 2, 3, 4, 5};
    - **Linked List:** A series of nodes where each node points to the next. Unlike arrays, linked lists don't require contiguous memory. Example: A singly linked list where each node points to the next node in the list.
    - **Stack:** A last-in, first-out (LIFO) structure. Elements are inserted and removed from the top of the stack. Example: A stack of books, where the last book added is the first one to be removed.
    - **Queue:** A first-in, first-out (FIFO) structure. Elements are inserted at the rear and removed from the front. Example: A line of people where the first person in line is the first one served.
- **Traversal:** In a linear data structure, you can traverse elements sequentially, one after another.

Non-linear Data Structures:

In non-linear data structures, the elements are not arranged in a sequential manner. Instead, they can form hierarchies or complex relationships where one element can be connected to multiple elements.

- **Characteristics:**
  - Data is organized in a hierarchical or interconnected way.
  - Traversal is not straightforward as elements may have multiple relationships.
  - More complex to implement and requires advanced memory management.
- **Examples:**
  - **Tree:** A hierarchical structure with a root node and child nodes forming branches. A binary tree, for example, has at most two children per node. Example: A family tree where each person can have multiple descendants.
  - **Graph:** A collection of nodes (vertices) connected by edges. It can be directed or undirected, and nodes can be interconnected in various ways. Example: A social network where users (nodes) are connected to each other (edges) in a web-like manner.
  - **Heap:** A specialized tree-based structure that satisfies the heap property, where the parent node is either greater than or equal to (in max heap) or less than or equal to (in min heap) its children. Example: A priority queue implemented using a heap.
- **Traversal:** Non-linear data structures require specialized algorithms to traverse, such as depth-first search (DFS) or breadth-first search (BFS) for trees and graphs.

Summary of Differences:

Feature	Linear Data Structures	Non-linear Data Structures
Organization	Sequential	Hierarchical or interconnected
Traversal	Single-level (simple)	Multi-level (complex)
Memory usage	Contiguous or linked	Dynamic and complex
Examples	Array, Linked List, Stack, Queue	Tree, Graph, Heap

2. Describe the linear search algorithm.

Ans: Linear search is a simple search algorithm used to find the position of a target value within a list or array. It works by sequentially checking each element of the list, one by one, until the desired element (target) is found or the entire list has been searched.

Algorithm Steps:

1. Start from the first element of the list (index 0).
2. Compare the current element with the target value.
3. If the current element matches the target, return the index of the element.
4. If the current element does not match the target, move to the next element.
5. Repeat the process until:
    - The target value is found (return its index).
    - The end of the list is reached (if no match is found, return -1 to indicate failure).

Pseudocode:

```
function linearSearch(arr, target):
  for i from 0 to length(arr) - 1:
    if arr[i] == target:
      return i  // Target found, return index
  return -1  // Target not found in the list
```

Example:

Consider an array `arr = [3, 8, 4, 9, 7]` and a target value of `9`.

1. Start at the first element:
    - Compare `arr[0]` (3) with `9` → No match.
2. Move to the second element:
    - Compare `arr[1]` (8) with `9` → No match.
3. Move to the third element:
    - Compare `arr[2]` (4) with `9` → No match.
4. Move to the fourth element:
    - Compare `arr[3]` (9) with `9` → Match found.

Return the index `3`, because `9` is located at index 3 in the array.

3. Explain the binary search algorithm.

Ans: Binary search is an efficient algorithm for finding the position of a target value within a **sorted** list. It works by repeatedly dividing the search range in half, reducing the number of

elements to search by half at each step. This significantly improves the search efficiency compared to linear search.

Conditions:

- The list must be sorted in ascending or descending order for binary search to work.

Algorithm Steps:

1. Start by setting two pointers: one to the beginning of the list (low) and one to the end (high).
2. Calculate the middle index (mid) of the current range: mid = (low + high) / 2.
3. Compare the target value with the middle element of the list:
    - If the middle element is equal to the target, the search is successful; return the index.
    - If the target is less than the middle element, discard the right half of the list by adjusting the high pointer to mid - 1.
    - If the target is greater than the middle element, discard the left half of the list by adjusting the low pointer to mid + 1.
4. Repeat the process until the low pointer is greater than the high pointer (meaning the target is not present in the list).
5. If the target is not found, return -1.

Pseudo code:

```
function binarySearch(arr, target):
  low = 0
  high = length(arr) - 1
    while low <= high:
    mid = (low + high) // 2  // Calculate middle index
       if arr[mid] == target:
      return mid  // Target found at index mid
       if arr[mid] < target:
      low = mid + 1  // Target is in the right half
    else:
      high = mid - 1  // Target is in the left half
     return -1  // Target not found in the array
```

4. What is asymptotic notation? Describe the different types of asymptotic notations.

Ans: Asymptotic Notations:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e., the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e., the worst case.

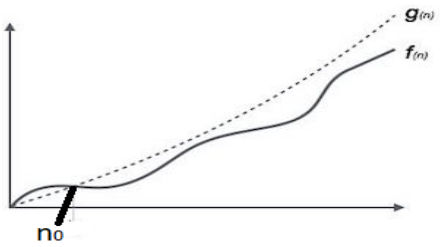
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.

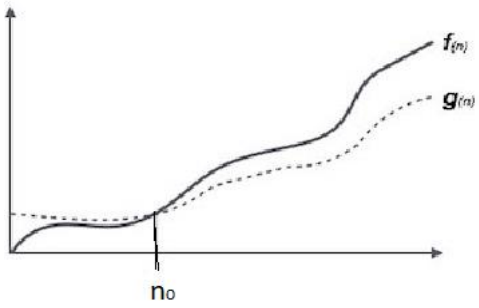


For example, for a function ***f*(n)**

O(f(n)) = { g(n) : there exists c > 0 and n0 such that f(n) ≤ c.g(n) for all n > n0. }

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best-case time complexity or the best amount of time an algorithm can possibly take to complete.

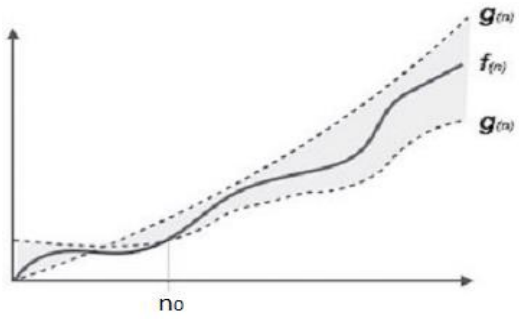


For example, for a function ***f*(n)**

Ω(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }

5. Describe the selection sort algorithm.

Ans: Selection sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on sorting order) element from the unsorted portion of the list and swapping it with the first unsorted element, thereby growing the sorted portion of the list step by step.

Algorithm Steps:

1. Start with the first element as the current minimum.
2. Iterate through the list to find the smallest element in the unsorted part.
3. Once the smallest element is found, swap it with the first unsorted element.
4. Move the boundary between the sorted and unsorted parts of the list.
5. Repeat the process for the remaining unsorted portion of the list.
6. Continue until the entire list is sorted.

Pseudocode:

```
function selectionSort(arr):
    n = length(arr)
    for i from 0 to n - 1:
        min_index = i  // Assume the current index holds the minimum element
       // Find the minimum element in the unsorted part
        for j from i + 1 to n - 1:
            if arr[j] < arr[min_index]:
                min_index = j
         // Swap the found minimum element with the element at index i
        if min_index != i:
            swap(arr[i], arr[min_index])
```

Example:

Consider an array `arr = [29, 10, 14, 37, 13]`.

1. **First pass** (`i = 0`):
   - Find the smallest element in the list `[29, 10, 14, 37, 13]` → The smallest is `10`.
   - Swap `10` with the first element (`29`): `[10, 29, 14, 37, 13]`.
2. **Second pass** (`i = 1`):
   - Find the smallest element in the unsorted list `[29, 14, 37, 13]` → The smallest is `13`.
   - Swap `13` with the element at index 1 (`29`): `[10, 13, 14, 37, 29]`.
3. **Third pass** (`i = 2`):
   - Find the smallest element in the unsorted list `[14, 37, 29]` → The smallest is `14`.
   - No swap needed as the element is already in the correct place.
4. **Fourth pass** (`i = 3`):
   - Find the smallest element in the unsorted list `[37, 29]` → The smallest is `29`.
   - Swap `29` with `37`: `[10, 13, 14, 29, 37]`.
5. **Final sorted list**: `[10, 13, 14, 29, 37]`.

6. Discuss the bubble sort algorithm. Why it is not preferred for large datasets?

Ans: Bubble sort is a simple comparison-based sorting algorithm that 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. The largest (or smallest, depending on sorting order) elements "bubble up" to their correct positions after each pass.

Algorithm Steps:

1. Start at the beginning of the list.
2. Compare the first element with the second:
   - If the first element is greater than the second, swap them.

- Otherwise, leave them in place.

3. Move to the next pair of adjacent elements and repeat the comparison and swapping process.
4. Continue until the end of the list. After the first pass, the largest element will be in its correct position.
5. Repeat the entire process for the remaining unsorted part of the list, excluding the last sorted element.
6. The algorithm stops when no swaps are made in a complete pass (indicating the list is sorted).

Pseudo code:

```
function bubbleSort(arr):
    n = length(arr)
  for i from 0 to n - 1:
      swapped = False
     // Last i elements are already sorted
      for j from 0 to n - i - 1:
         if arr[j] > arr[j + 1]:
            swap(arr[j], arr[j + 1])
            swapped = True
     // If no two elements were swapped in the inner loop, the list is sorted
      if not swapped:
         break
```

Why Bubble Sort is Not Preferred for Large Datasets:

1. **Inefficient Time Complexity:**
   - Bubble sort has a time complexity of $O(n^2)$ in both the average and worst cases, which makes it inefficient for large datasets. For a list of n elements, it requires around n^2 comparisons and swaps, leading to a significant performance slowdown as the size of the list increases.
2. **Too Many Comparisons and Swaps:**
   - The algorithm compares every pair of adjacent elements and potentially swaps them, even if the list is almost sorted. This results in a high number of unnecessary operations, which further reduces its efficiency.

3. **Doesn't Handle Large Inputs Well:**
    - Since its time complexity grows quadratically, it is impractical for sorting large datasets. More efficient algorithms like **Merge Sort (O(n log n))**, **Quick Sort (O(n log n))**, or **Heap Sort (O(n log n))** are better suited for large datasets.
4. **No Adaptability:**
    - Although bubble sort can terminate early if no swaps are made in a pass, this "optimization" doesn't help much in cases where only a few elements are out of order. More sophisticated algorithms like **Insertion Sort** or **Quick Sort** adapt better to the order of the input data.