ANSWER 01)-

Part a):

Data structures are essential components in computer science used to organize, manage, and store data efficiently. Here's a detailed overview of different types of data structures:

1. Arrays : A collection of elements, identified by index or key. Arrays have a fixed size, and each element is of the same data type. They provide fast access (O(1) time complexity) to elements by index but are costly for insertions and deletions (O(n) time complexity).
2. Linked lists : A sequence of elements where each element points to the next. There are several types:

a. Single : each node contains data and a reference to the next node.
b. Double : Each node has references to both the next and the previous node.
c. Circular : The last node points back to the first node. Linked lists allow for efficient insertions and deletions (O(1) time complexity) but slower access times (O(n)).

3. Stacks : collection of elements that follows the Last In, First Out (LIFO) principle. Operations include push (insert) and pop (remove). Stacks are used in function call management and undo mechanisms.
4. Hash tables : Store key-value pairs for efficient data retrieval. They use a hash function to compute an index into an array of buckets from which the desired value can be found. Hash tables provide average-case O(1) time complexity for search, insert, and delete operations.
   Each data structure is suited for specific tasks, and understanding their properties helps in selecting the right one for the given problem.

Part b): -

The complexity of an algorithm refers to the measure of the resources (such as time and space) required by the algorithm to solve a problem as a function of the size of the input. It is a crucial aspect of algorithm design and analysis, helping to predict and compare the efficiency of different algorithms. The two main types of complexity are time complexity and space complexity.

TIME COMPLEXITY

Time complexity refers to the amount of time an algorithm takes to complete as a function of the size of the input (often denoted as nn). It is usually expressed using Big O notation, which describes the upper bound of the algorithm's running time, providing an estimate of the worst-case scenario.

Common time complexity:

1. Constant time: The running time is constant and does not change with the input size. Example: Accessing an element in an array.
2. Logarithmic time: The running time grows logarithmically with the input size. Example: Binary search in a sorted array.

3. Linear time : The running time grows linearly with the input size. Example: Iterating through an array.

   SPACE COMPLEXITY
   refers to the amount of memory an algorithm uses in relation to the size of the input. This includes both the memory needed to store the input and the additional memory used by the algorithm to process the input.
   Common space complexity:
   1. Constant space : The algorithm uses a fixed amount of space regardless of the input size. Example: Simple arithmetic operations.
   2. Linear space : The memory required grows linearly with the input size. Example: Storing input in an array.

ANSWER 02):-

Linked list is a data structure consisting of a sequence of elements, where each element (called a node) contains data and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists do not have a fixed size and allow for efficient insertion and deletion of elements, making them a versatile tool for various applications in computer science.

Structure of a linked list

1. Data : The value or data held by the node.
2. Next : A reference (or pointer) to sequence.

   Types of linked lists:

1. Singly linked lists singly linked list is the simplest type of linked list. Each node contains data and a reference to the next node. The last node has a reference pointing to **null**, indicating the end of the list.
   Advantages: a. simple to understand.

   b. efficient for insertion and deletion operations at the beginning of the list (O(1) time complexity).

2. Doubly linked lists :
   doubly linked list contains nodes with references to both the next and the previous node. This allows traversal in both directions. Advantages : Bidirectional traversal is possible.
   efficient insertions and deletions at both ends of the list.
   Disadvantages : memory is required per node due to the additional reference. slightly more complex to implement compared to.

3. Dummy linked lists: a sentinel linked list, a dummy node (or sentinel node) is used at the beginning or end (or both) of the list to simplify boundary conditions and eliminate the need for special cases in insertion and deletion operations.

Advantages: simplifies insertion and deletion operations by eliminating edge cases. improves code readability and maintainability.
Disadvantages: extra memory is required for the sentinel nodes.
additional overhead in some operations due to the presence of dummy nodes.

Use case and applications
Linked lists are widely used in various applications due to their dynamic nature and efficient operations. Some common use cases include:
1. Implementing dynamic data structures like stacks, queues.
2. Graph adjanceny representations in graph algorithms.
3. Handling collision in hash tables in separate changings.

Operations on linked lists
1. Insertion: Adding a new node to the list. This can be done at the beginning, end, or any position within the list.
2. Deletion: removing a node from the list. This can involve updating the references of neighboring nodes.
3. Transversal: accessing each node in the list, typically for searching or modifying data.
4. Reversal: changing the direction of the links, effectively reversing the order of nodes.

Conclusion
Linked lists are fundamental data structures that provide a flexible way to store and manage data. Understanding the various types of linked lists and their operations is essential for effective algorithm design and problemsolving in computer science. Each type of linked list has its own advantages and trade-offs, making it suitable for different scenarios and applications.

ANSWER 03):-

Part a):

stack is a fundamental data structure that operates on a Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. The two primary operations associated with a stack are push and pop.

1. Check for overflow: before adding a new element, it's essential to check if there is enough space in the stack. In the case of an array-based stack, this means checking if the stack is full. If it is, an overflow condition occurs, and the operation cannot proceed.

2. Increment the top pointer : stack typically maintains a pointer (or index) to the current top of the stack. Before adding the new element, this pointer is incremented to point to the next position where the new element will be placed.
3. Add the element : new element is placed at the position indicated by the top pointer.
4. Update the stack state : stack now includes the new element, and the top pointer accurately reflects the current top of the stack.

Push operations examples:

```
procedure PUSH(stack, element):
if stack is full:
    print("Stack Overflow")
    return
top = top + 1
stack[top] = element.
```

POP operation

operation removes and returns the element from the top of the stack. Here’s a detailed explanation of how the POP operation works:

1. Check for underflow: before removing an element, it is crucial to check if the stack is empty. If it is, an underflow condition occurs, meaning there are no elements to remove, and the operation cannot proceed.
2. Retrieve the element : element at the position indicated by the top pointer is retrieved. This is the element that will be removed from the stack.
3. Decrement the top pointer: retrieving the element, the top pointer is decremented to point to the new top of the stack (which is now the previous element).
4. Update the stack state: stack is updated to reflect that the top element has been removed.

Pseudocode for pop operation

```
procedure POP(stack):
if stack is empty:
print("Stack Underflow")
    return None
element = stack[top]
top = top - 1
return element.
```

Part B):-

AVL tree, named after its inventors Adelson-Velsky and Landis, is a type of selfbalancing binary search tree (BST). In an AVL tree, the heights of the two child subtrees of any node differ by at most one, which ensures that the tree remains balanced. This balance condition is crucial for maintaining efficient search, insertion, and deletion operations.

AVL tree is a self-balancing binary search tree that maintains a strict balance condition to ensure efficient performance of search, insertion, and deletion operations. Unlike a standard BST, which can become unbalanced and degrade in performance, an AVL tree uses rotations to maintain balance, making it a reliable choice for dynamic datasets where maintaining order and efficiency is crucial.

Difference between AVL and BST trees:

1. Balance condition:
   AVL = Maintains a strict balance condition where the heights of the child subtrees of any node differ by at most one. This ensures that the tree remains balanced, providing O(log n) time complexity for search, insertion, and deletion operations.
   BST = No specific balance condition is enforced. As a result, a BST can become skewed, leading to poor performance with operations degrading to O(n) in the worst case.
2. Performance :
   BST = Performance can degrade if the tree becomes unbalanced, particularly with skewed trees.
   AVL = ensures consistently good performance by maintaining balance, making operations faster on average compared to an unbalanced BST.
3. Use case :
   BST = Suitable for scenarios where data is inserted in a random manner and the tree is unlikely to become skewed.
   AVL = ideal for applications requiring frequent insertions and deletions, ensuring the tree remains balanced and operations stay efficient.

Answer 04):-

Searching is a fundamental operation in computer science, used to find the location of a specific element within a data structure. Two common searching algorithms are linear search and binary search. Here's a detailed explanation of their differences, with examples.

Linear search linear search, also known as sequential search, is the simplest search algorithm. It checks each element in the list sequentially until the target element is found or the list ends.
Characteristics:

1. Time complexity: where n is the number of elements in the list. This is because, in the worst case, the algorithm may have to check every element.
2. Space complexity : since no additional storage is needed.
3. Data requirement : works on both sorted and unsorted lists. Example : Suppose we have an unsorted list: [34, 7, 23, 32, 5, 62] and we want to find the element 23. procedure linearSearch(arr, target):   for each element in arr:
   ```
       if element == target:
   ```

```
        return index of element
return -1  // target not found.
```

Binary search

Binary search is a more efficient search algorithm but requires the list to be sorted. It works by repeatedly dividing the search interval in half. If the value of the target element is less than the middle element, the search continues in the lower half, otherwise, it continues in the upper half.

Charaterstics:

1. Time complexity : where n is the number of elements. This is because the search space is halved each time.
2. Space complexity : for iterative implementation, O(log n) for recursive implementation due to the call stack.
3. Data requirement: requires a sorted lists. Examples procedure binarySearch(arr, target):

difference between linear and binary search :-

1. Efficiency :
   a. Linear : time complexity. Best suited for small or unsorted datasets.
   b. Binary: (log n) time complexity. Much more efficient for large datasets but requires sorted data.
2. Data requirement :
   a. Linear : Can work on any list (sorted or unsorted).
   b. Binary : Requires the list to be sorted beforehand.
3. Use case :
   a. Linear : Used when the dataset is small, or sorting the data is impractical.
   b. Binary : Requires a bit more logic to handle the mid-point calculations and the halving of the search space.

Conclusion:

Both linear and binary search algorithms have their own strengths and use cases. Linear search is versatile and easy to implement but is less efficient for large datasets. Binary search, while requiring sorted data, offers significant performance improvements for large datasets due to its logarithmic time complexity. Understanding these differences is crucial for selecting the appropriate search algorithm based on the context of the problem and the characteristics of the data.

Answer 05):-

external sorting is a category of algorithms used for sorting data that cannot fit into a computer's main memory (RAM) and must be stored in external storage, such as hard drives or SSDs. These algorithms are essential when dealing with massive datasets, where traditional in-memory sorting algorithms are impractical due to memory constraints.

Key concepts of external sorting :

1. Memory hierarchy: External sorting leverages the memory hierarchy, using both fast but limited main memory and slower but abundant external storage. The goal is to minimize data transfer between these memory levels, as disk I/O operations are significantly slower than in-memory operations.
2. Chunk processing : data is divided into manageable chunks that fit into main memory. Each chunk is loaded into memory, sorted using an efficient in-memory sorting algorithm (like quicksort or mergesort), and then written back to disk.
3. Merge phases: After sorting the chunks, the next step involves merging these sorted chunks to produce a single sorted dataset. This is typically done using multi-way merging, which is an extension of the two-way merge process used in mergesort.

Steps involved in external sorting:

1. Divide the data into chunks:
   dataset is divided into smaller chunks that can fit into the available main memory.
2. Sort each chunk: chunk is read into main memory, sorted using an efficient in-memory sorting algorithm, and then written back to disk.
3. Merge sorted chunks: sorted chunks are then merged to form a single sorted file. This can be done in multiple passes if the entire set of sorted chunks cannot be merged in one go due to memory limitations.

External merge sort :

1. External merge sort is one of the most commonly used external sorting algorithms. Here's a detailed example of how it works: Initinal sorting phase:
   a. assume we have a large dataset of size 100 GB, and our main memory can handle 1 GB at a time.
   b. dataset is divided into 100 chunks of 1 GB each.
   c. 1 GB chunk is loaded into memory, sorted using an in-memory algorithm, and then written back to disk, resulting in 100 sorted 1 GB files.
   Merge phase:
      a. the first merge pass, groups of k sorted chunks (where **k** is the number of chunks that can be merged in memory) are merged. Suppose our system can handle 10 files at a time.
      b. 100 sorted files are processed in batches of 10. Each batch of 10 files is merged into a single sorted file of size 10 GB.
      c. the first pass, we have 10 sorted files of 10 GB each
      d. the next merge pass, these 10 sorted 10 GB files are merged into a single 100 GB sorted file.

External sorting is a vital technique for managing and sorting large datasets that do not fit into main memory. By dividing the data into chunks, sorting each chunk in memory, and then efficiently merging the sorted chunks, external sorting algorithms like external merge sort provide scalable and efficient solutions for handling massive data volumes. Understanding these principles is crucial for working with big data and ensuring that systems can scale to meet the demands of large-scale data processing.

Answer 06):-

Collision resolution methods are techniques used in hashing to handle situations where multiple keys hash to the same index in a hash table. When such a collision occurs, these methods ensure that the hash table can store and retrieve all elements correctly. There are several common collision resolution techniques, each with its own advantages and tradeoffs. Here, we'll discuss the following methods in detail:

1. chaining: Chaining involves creating a linked list (or another secondary data structure like a binary tree) at each index of the hash table. When a collision occurs, the new element is simply added to the list at that index.
   Advantages:
   a. simple to understand.
   b. hash table size does not need to be adjusted dynamically.

   Disadvantages:

   a. performance degrades if the linked lists become too long (though less likely with a good hash function).
   Example:

Index	Linked List
0	[ ]
1	[15, 8]
2	[ ]
3	[ ]
4	[11]
5	[12]
6	[27]

2. open addressing : open addressing, all elements are stored within the hash table itself. When a collision occurs, a probing sequence is used to find another empty slot.
   Linear probing: probing handles collisions by checking the next slot in a sequential manner until an empty slot is found.
   Advantages: elements are stored in a contiguous block of memory, which can improve cache performance.
   Disadvantages: Consecutive occupied slots can cause longer search times. Collisions tend to create clusters, worsening performance.

Example:

Index	Value
0	[ ]
1	[15]
2	[ ]
3	[ ]
4	[11]
5	[12]
6	[27]
7	[8] (collision resolved by checking next available index)

3. perfect hashing:
   Perfect hashing aims to create a hash function that maps keys to unique slots, eliminating collisions. It is typically used in static environments where all keys are known in advance.
   Advantages: no collision hence constant time operation. Disadvantages: complex to find a perfect hash function.
4. Cuckoo hashing: cuckoo hashing uses two hash functions and two hash tables. If a collision occurs, the existing element is "kicked out" and reinserted using the other hash function.
   Advantages: constant time lookups.
   Eliminating clusters.
   Disadvantages: more complex to implement.
   Requires more memory.

5. Robin hood hashing: Robin Hood hashing equalizes the load by ensuring that elements with longer probe sequences "steal" slots from elements with shorter probe sequences.
   Advantages: reduces variance in probe sequence length.
   Helps in balancing load .
   Disadvanteges:
   careful management to avoid excessive movement.

   Conclusion:
   Collision resolution methods are crucial for maintaining efficient hash table operations. Chaining is simple and effective, particularly for dynamic datasets. Open addressing, with its various probing techniques, offers a space-efficient alternative, though careful management of clustering is necessary. Perfect hashing is ideal for static datasets, while cuckoo and Robin Hood hashing provide advanced strategies for reducing probe sequence length and clustering. Understanding these methods helps in selecting the appropriate technique based on the specific requirements and constraints of the application.