

DATA STRUCTURE & ALGORITHMS

INTRODUCTION

Q-1: What is Abstract Data? Define data structure and types of Data Structure with example. (Linear and Non-Linear Data Structure).

Answer:

Abstract data is a way of representing data that hides the implementation details from the user. This allows the user to focus on the data itself, without worrying about how it is stored or manipulated.

A data structure is a way of organizing data so that it can be stored and accessed efficiently. There are two main types of data structures: linear data structures and non-linear data structures.

- Linear data structures are data structures in which the data elements are arranged in a sequence. Examples of linear data structures include arrays, linked lists, and queues.
- Non-linear data structures are data structures in which the data elements are not arranged in a sequence. Examples of non-linear data structures include trees, graphs, and heaps.

Q-2: Briefly describe complexity and space time trade-off of algorithms.

Answer:

The complexity of an algorithm is a measure of how much time and space it takes to run. The space complexity of an algorithm is the amount of memory it needs to store its data, while the time complexity is the amount of time it takes to execute its instructions.

The space-time trade-off is the relationship between the space and time complexity of an algorithm. In general, algorithms with lower space complexity tend to have higher time complexity, and vice versa.

For example, a bubble sort algorithm has a time complexity of $O(n^2)$, while a quicksort algorithm has a time complexity of $O(n \log n)$. However, the bubble sort algorithm has a space complexity of $O(1)$, while the quicksort algorithm has a space complexity of $O(\log n)$.

Q-3: Describe preliminary idea of algorithms runtime complexity.

The runtime complexity of an algorithm is a measure of how long it takes to run. It is typically expressed as a function of the size of the input.

There are three main types of runtime complexity:

- **Constant time:** The algorithm takes the same amount of time to run, regardless of the size of the input.
- **Linear time:** The algorithm's running time is proportional to the size of the input.
- **Exponential time:** The algorithm's running time grows exponentially with the size of the input.

Q-4: How to represent linear array in memory. Traversing Algorithm for Linear Array.

Answer

A linear array can be represented in memory as a contiguous block of memory. The elements of the array are stored in consecutive memory locations.

For example, consider an integer array [10, 20, 30, 40]. In memory, it may be represented as follows:

Address	Value
1000	10
1004	20
1008	30
1012	40

Traversing Algorithm:

To traverse a linear array, we can use a for loop. The for loop will iterate through the array, one element at a time.

To traverse a linear array, you can simply iterate through the array from the first element to the last element, accessing each element in turn. The traversal algorithm might look something like this:'

```
for (i = 0; i < n; i++) {  
    // Access element i  
}
```

where **n** is the length of the array.

Q-5: Why algorithms analysis is important?

Answer:

Algorithms analysis is important because it allows us to choose the most efficient algorithm for a given problem. By understanding the runtime complexity of different algorithms, we can select the algorithm that will take the least amount of time to run.

- **Efficiency:** By analyzing algorithms, we can determine their efficiency in terms of time and space usage.
- **Performance Comparison:** Algorithm analysis allows us to compare different algorithms and choose the most suitable one for a specific task.
- **Scalability:** As the input size increases, the performance of an algorithm becomes crucial. Algorithm analysis helps us understand how an algorithm's runtime or space requirements grow as the input grows. It ensures that the chosen algorithm can handle larger inputs efficiently.
- **Optimization Opportunities:** Analyzing algorithms can reveal areas where improvements can be made.
- **Problem Solving:** Understanding algorithm analysis provides a foundation for problem-solving in computer science.

Q-6: What is asymptotic analysis of an algorithms? What are asymptotic notation?

Answer:

Asymptotic analysis is a way of analysing the runtime complexity of algorithms. It is used to measure the running time of an algorithm as the input size grows infinitely large.

Asymptotic notation is a way of writing functions that ignores constants and lower-order terms. The most common asymptotic notations are:

- **$O(n)$:** The algorithm's running time is proportional to the size of the input.
- **$\Omega(n)$:** The algorithm's running time is at least proportional to the size of the input.
- **$\Theta(n)$:** The algorithm's running time is asymptotically equal to the size of the input.

Q-7: Definition and Classification of Array. How to 2D array represent in memory.

Answer:

An array is a data structure that can store a collection of elements of the same type. The elements of an array are stored in contiguous memory locations.

Arrays can be classified as either one-dimensional or two-dimensional. A one-dimensional array is an array that stores elements in a single row. A two-dimensional array is an array that stores elements in a grid.

2D array represent in memory:

It represents a tabular structure with rows and columns. Elements are accessed using two indices: one for the row and another for the column. For example, a 2D array can store a table of numbers like:

For example, a 2D array can store a table of numbers like:

1	2	3
4	5	6
7	8	9

To represent a 2D array in memory, the elements are typically stored sequentially. Depending on the programming language and memory layout, the elements can be stored row-wise or column-wise. For the above example, a row-wise representation may look like:

Address	Value
1000	1
1004	2
1008	3
1012	4
1016	5
1020	6
1024	7
1028	8
1032	9

Q-8: Distinguish between Linear and Non-Linear Data structure.

Answer:

- ✚ Linear data structures are data structures in which the data elements are arranged in a sequence. This means that each data element has a **single predecessor and a single successor**. Examples of linear data structures include arrays, linked lists, and queues.
- ✚ Non-linear data structures are data structures in which the data elements are not arranged in a sequence. This means that each data element may have **multiple predecessors and multiple successors**. Examples of non-linear data structures include trees, graphs, and heaps.

Here is a table that summarizes the key differences between linear and non-linear data structures:

Feature	Linear Data Structures	Non-Linear Data Structures
Data arrangement	Data elements are arranged in a sequence	Data elements are not arranged in a sequence
Predecessor and successor	Each data element has a single predecessor and a single successor	Each data element may have multiple predecessors and multiple successors
Examples	Arrays, linked lists, queues	Trees, graphs, heaps

Q-9: Define overflow, underflow and garbage collection?

Answer:

- ✚ **Overflow occurs when the data in a data structure exceeds its capacity.** For example, if an array is only capable of storing 10 elements, and we try to add an 11th element, an overflow will occur.
- ✚ **Underflow occurs when the data in a data structure falls below its minimum value.** For example, if a stack is only capable of storing 10 elements, and we try to pop the 11th element, an underflow will occur.
- ✚ **Garbage collection is a process that automatically reclaims memory that is no longer being used by an application.** This is done by identifying objects that are no longer reachable from any live references, and then freeing the memory that they occupy.

Q-10: What is a complexity? Calculating the time and space complexity.

```
function addMatrices(A, B, n):  
  for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++) {  
      C[i][j] = A[i][j] + B[i][j]  
    }  
  }  
  
  return C
```

Answer:

The complexity of an algorithm is a measure of how much time and space it takes to run. The space complexity of an algorithm is the amount of memory it needs to store its data, while the time complexity is the amount of time it takes to execute its instructions.

Time Complexity:

The time complexity of this algorithm is $O(n^2)$. It has two nested loops, both iterating from 0 to $n-1$, to access each element in the $n \times n$ matrices A and B. Since the loops are nested, the total number of iterations is $n * n = n^2$.

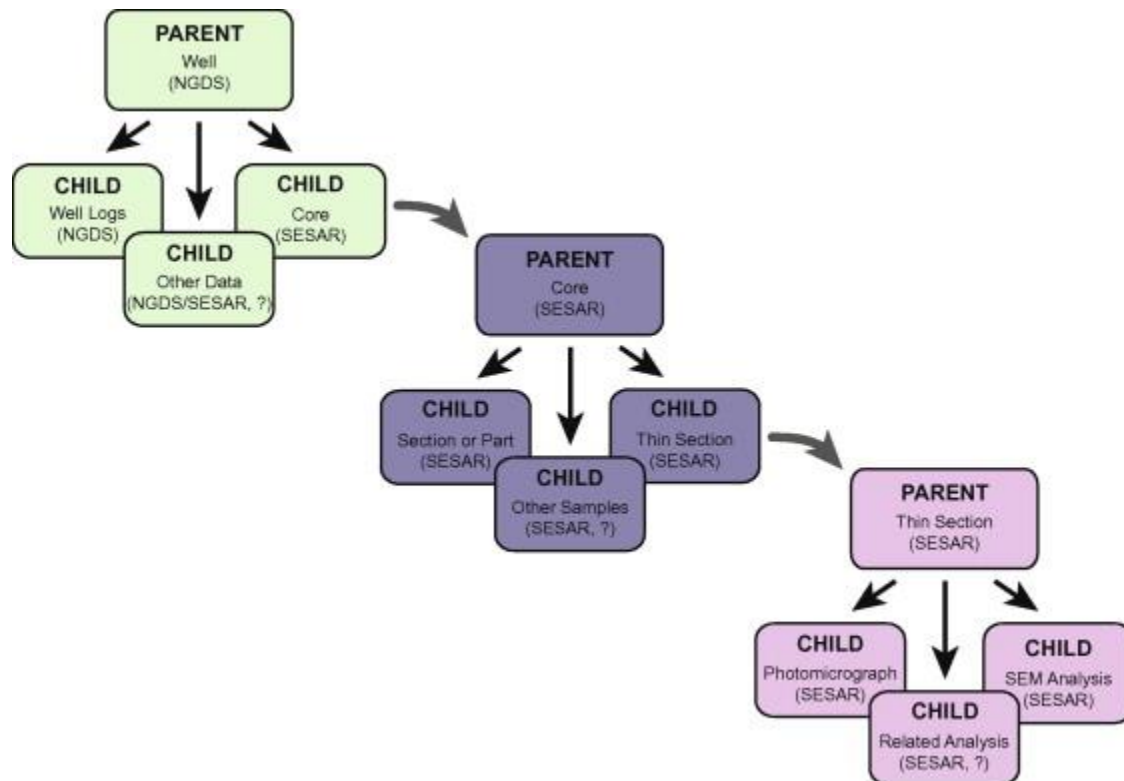
Space Complexity:

The space complexity of this algorithm is also $O(n^2)$. It creates a new 2D array C of size $n \times n$ to store the result. The size of the array C directly depends on the input size 'n', so the space complexity grows quadratically with 'n'.

In summary, the provided algorithm for matrix addition has a time complexity of $O(n^2)$ and a space complexity of $O(n^2)$.

S Q: What is a "parent-child relationship" in the context of data structures

Answer:



In the context of data structures, a "parent-child relationship" refers to a hierarchical relationship between elements or nodes in a structure. It is commonly associated with tree-based data structures such as binary trees, general trees, and graphs.

In this relationship, each node (except for the root node) has a parent node, and zero or more child nodes. The parent node is the immediate predecessor or ancestor of a given node, while the child nodes are the immediate successors or descendants of that node.

The parent-child relationship allows for efficient traversal and manipulation of the data structure. enables operations such as searching for a specific node, inserting new nodes, deleting nodes, and navigating through the structure by moving from parent to child or vice versa.

LINK LIST:

Q-1: What is link list? Linked List Classification with example. Explain different type of linked list with pictorial view.

Answer:

A linked list is a data structure that consists of a collection of nodes, where each node contains data and a pointer to the next node in the list. The first node in the list is called the head, and the last node is called the tail. Linked lists are a dynamic data structure, which means that they can grow and shrink as needed. This makes them a good choice for applications where the size of the data set is not known in advance, or where the data set is frequently updated.

There are three main types of linked lists:

Singly linked list: Each node in a singly linked list contains a pointer to **the next node in the list**.

```
Head -> [Node1] -> [Node2] -> [Node3] -> null
```

Doubly linked list: Each node in a doubly linked list contains *pointers to the previous and next nodes in the list*.

```
null <- [Node1] <-> [Node2] <-> [Node3] -> null
```

Circular linked list: A circular linked list is a linked list *where the last node points back to the first node*.

```
Head -> [Node1] -> [Node2] -> [Node3] -|
      ^-----|
```


Q-2: Write different between link list and array.

Answer:

Here is a table that summarizes the key differences between linked lists and arrays:

- **Memory allocation** - Arrays are allocated contiguous blocks of memory, while linked lists use non-contiguous memory.
- **Size** - The size of an array is fixed, while a linked list can grow dynamically.
- **Indexing** - Arrays can be indexed using integers, while linked lists require traversal to access a specific node.
- **Insertion and deletion** - Array insertion and deletion can be expensive, whereas linked lists provide efficient insertion and deletion operations.

Feature	Linked list	Array
Data structure	Dynamic	Static
Memory usage	More space-efficient	Less space-efficient
Insertion and deletion	Easy	Difficult
Searching	Slow	Fast
Accessing elements	Sequential access only	Random access

Q-3: Write an algorithm to perform the following operation:

- i) Searching a node from a link list.**
- ii) Create a circular linked list**
- iii) To insert an element after the given element of the list.**
- iv) To delete an element from the end of the list.**
- v) To traverse & Searching (Sorted and Unsorted) a linked list.**

Answer:

Searching a node from a link list.

```
Algorithm Search(head, value):  
    current = head  
    while current is not null:  
        if current.value is equal to value:  
            return current  
        current = current.next  
    return null
```

Create a circular linked list.

```
Algorithm CreateCircularLinkedList(values):  
    head = null  
    tail = null  
    for each value in values:  
        node = createNode(value)  
        if head is null:  
            head = node  
        else:  
            tail.next = node  
        tail = node  
    tail.next = head  
    return head
```

End function

To insert an element after the given element of the list.

```
Algorithm InsertAfter(node, value):  
    if node is null:  
        return  
    newNode = createNode(value)  
    newNode.next = node.next  
    node.next = newNode
```

To delete an element from the end of the list.

```
Algorithm DeleteEnd(head):  
    if head is null or head.next is null:  
        return null  
    current = head  
    while current.next.next is not null:  
        current = current.next  
    current.next = null  
    return head
```

To traverse & Searching (Sorted and Unsorted) a linked list.

Traverse:

```
Algorithm Traverse(head):  
    current = head  
    while current is not null:  
        // Process current node  
        current = current.next
```

Searching (Sorted)

```
Algorithm SearchSorted(head, value):  
    current = head  
    while current is not null and current.value is less than or equal to val:  
        if current.value is equal to value:  
            return current  
        current = current.next  
    return null
```

Searching (Unsorted)

```
Algorithm SearchUnsorted(head, value):  
    current = head  
    while current is not null:  
        if current.value is equal to value:  
            return current  
        current = current.next  
    return null
```

Q-4: State some applications of linked lists. Representing Linked List in memory. Or Pictorial view.

Answer:

Linked lists have various applications in different domains. Here are some common applications:

- i. **Dynamic data structures:** Linked lists allow efficient insertion and deletion operations, making them suitable for dynamic data structures like stacks, queues, and hash tables.
- ii. **Implementing graphs:** Linked lists can be used to represent adjacency lists in graph data structures, where each node represents a vertex, and its linked list contains the adjacent vertices.
- iii. **File systems:** Linked lists are used to manage the hierarchical structure of files in a file system, where each node represents a file or directory and its linked list contains the child files or subdirectories.
- iv. **Music and video playlists:** Linked lists can be used to create playlists, where each node represents a song or video and its linked list contains the references to the next and previous songs or videos.

Representing Linked List in memory:

In memory, linked lists are represented by allocating memory dynamically for each node and using pointers to establish the connections between nodes. Each node contains the data value and a reference (pointer) to the next node. The last node's reference is typically null, indicating the end of the list.

Fig-01:

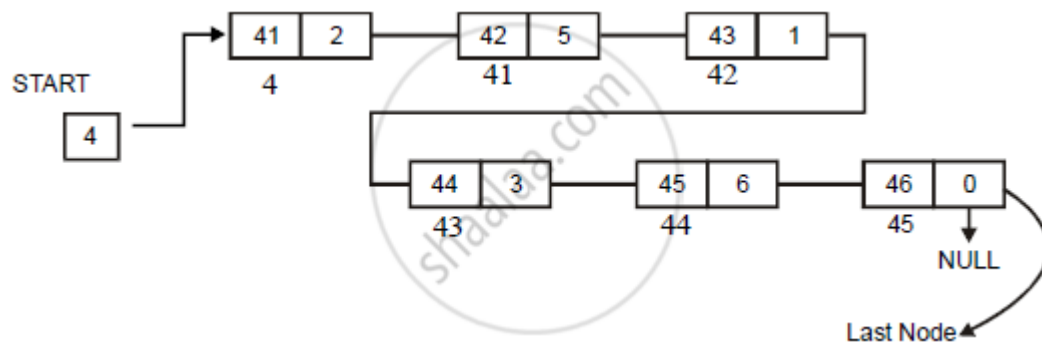
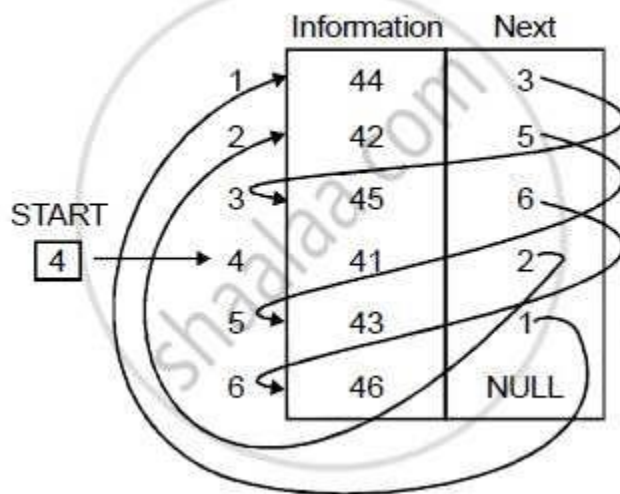
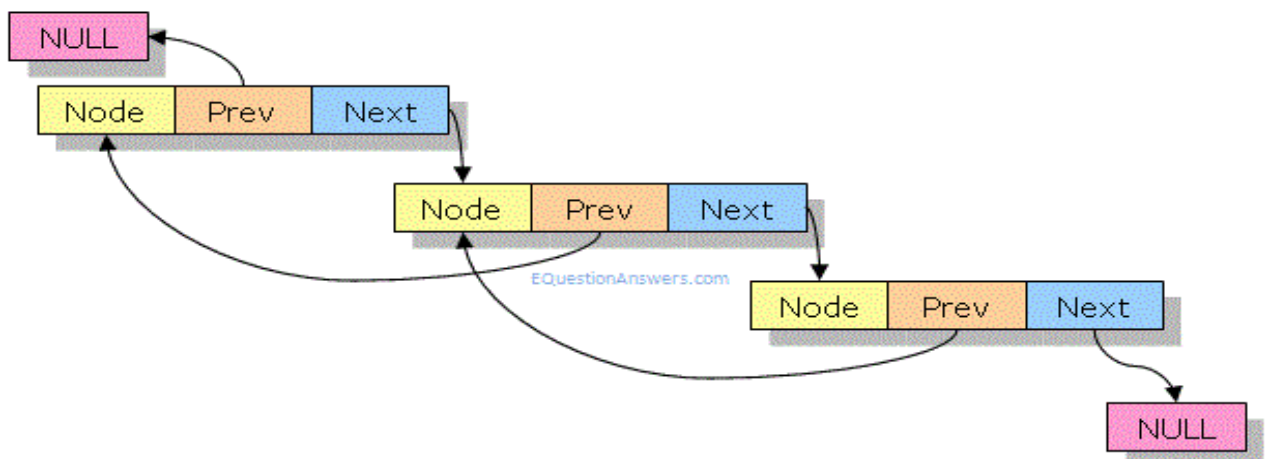


Fig-02:



Pictorial view:



Q-5: How does a linked list overcome the limitation of an array? Write down short note: Single, double & circular link list

Answer:

Linked lists overcome several limitations of arrays by providing the following advantages:

- **Dynamic Size:** Linked lists can grow or shrink dynamically, unlike arrays, which have fixed sizes. Therefore, linked lists can accommodate data items that may not be known at compile time.
- **Insertion and Deletion:** Insertion and deletion of elements in a linked list can be done without moving other elements, making it more efficient than arrays.
- **Memory Allocation:** Linked lists do not require contiguous memory allocation, unlike arrays, which need a fixed block of memory to store all their elements.
- **Flexibility:** Linked lists support different types of traversal, such as forward and backward traversal (in the case of doubly linked lists), circular traversal, and skipping nodes during traversal.

Short note: Single, double & circular link list:

Q-7: Suppose there is a Linked List in memory. Write down the advantages and disadvantages of an array over linked list.

Answer:

Arrays have several advantages over linked lists, including:

- **Random access:** Arrays allow for random access to any element by its index. This means that accessing elements in an array is very fast and efficient.
- **Memory usage:** Arrays use less memory than linked lists, as they do not require pointers to store next and previous nodes.
- **Processor cache:** Arrays are better suited to processor caching, meaning that they can be loaded into memory more efficiently.

However, arrays also have some disadvantages compared to linked lists, including:

- **Fixed size:** Arrays are of fixed size, meaning that space must be allocated for all elements beforehand, even if they are not needed. This can lead to wasted memory.
- **Insertion and Deletion:** Inserting or deleting elements in an array can be expensive, as it requires the movement of other elements.
- **Memory allocation:** Arrays require contiguous memory allocation, which can make them difficult to resize or move.

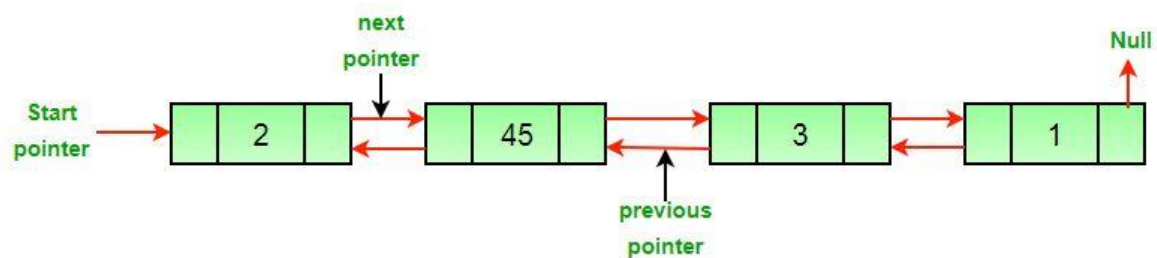
ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Q-8: Explain with appropriate figure how a node N between Node A and Node B can be deleted form a Linked List?

Answer:

To delete a node N between Node A and Node B, we can use the following algorithm:

1. Find the nodes A and B.
2. Set a pointer to the node N.
3. Set the next pointer of the node before N to the next pointer of the node N.
4. Delete the node N.



STACK AND QUEUE:

Q-1: All stacks are lists but all lists are not stacks"-Explain this statement with example

Answer:

A stack is a data structure that follows the Last In First Out (LIFO) principle.

This means that the last element added to the stack is the first element to be removed. A list is a data structure that allows for the insertion and deletion of elements at any position.

All stacks are lists because they are both data structures that store elements in a sequence. However, not all lists are stacks because not all lists follow the LIFO principle. For example, a linked list is a list that allows for the insertion and deletion of elements at any position, but it does not follow the LIFO principle.

Here is an example of how a stack can be used:


```
// Create a stack
Stack<Integer> stack = new Stack<>();

// Push some elements onto the stack
stack.push(1);
stack.push(2);
stack.push(3);

// Pop the elements off the stack
int x = stack.pop();
int y = stack.pop();
int z = stack.pop();

// x, y, and z will all be equal to 1, 2, and 3, respectively
```

Here is an example of how a linked list can be used:

```
// Create a linked list
LinkedList<Integer> list = new LinkedList<>();

// Add some elements to the linked list
list.add(1);
list.add(2);
list.add(3);

// Remove the first element from the linked list
int x = list.removeFirst();

// x will be equal to 1
```

Q-2: Show all the steps to evaluate the following postfix expression using postfix expression evaluating algorithm: (stack)

ABC+ * CBA-+*; assume A=1, B=2 and C=3

Answer:

Here are the steps to evaluate the postfix expression ABC+ * CBA-+*; assuming A=1, B=2 and C=3:

- Start by pushing the values of A, B and C onto the stack.
- Next, pop the top two values off the stack and perform the addition operation. The result of the addition operation is pushed back onto the stack.
- Repeat step 2 until there are no more values left on the stack.
- The final value on the stack is the result of the expression.

Here is the detailed steps:

1. Push A, B and C onto the stack.

stack = [1, 2, 3]

2. Pop the top two values off the stack and perform the addition operation. The result of the addition operation is pushed back onto the stack.

stack = [1, 5]

3. Repeat step 2 until there are no more values left on the stack.

stack = [10]

4. The final value on the stack is the result of the expression.

result = 10

ABC+*CBA-+* assum A=1, B=2, C=3

⇒ 1 2 3 + * 3 2 1 - + *

Symbol	STACK	Operation
1	1	
2	1, 2, 3	
3	1, 2, 3	
+	1, 5	[2+3]
*	5	[1*5]
3	5, 3	
2	5, 3, 2	
1	5, 3, 2, 1	
-	5, 3, 1	[2+1]
+	5, 4	[3+1]
*	20	[5*4]

Q-3: Convert the following infix expression into its equivalent postfix expression:(stack)

$$(A+B) * C - (D-E) ^{(F+G)}$$

$$A+B *(C-D)/(P-R)$$

Answer:

i) $(A+B)*C-(D-E)^{(F+G)}$

Symbol	STACK	Postfix Expression
((
A	(A	A
+	(A+	A
B	(A+B	AB
)		AB+
*	(AB+	AB+
C	(AB+C	AB+C
-	(AB+C-	AB+C*
((AB+C- (AB+C*
D	(AB+C- (D	AB+C*D
-	(AB+C- (D-	AB+C*D
E	(AB+C- (D-E	AB+C*DE
)	(AB+C- (D-E-	AB+C*DE-
^	(AB+C- (D-E-^	AB+C*DE-
((AB+C- (D-E-^(AB+C*DE-
F	(AB+C- (D-E-^(F	AB+C*DE-F
+	(AB+C- (D-E-^(F+	AB+C*DE-F
G	(AB+C- (D-E-^(F+G	AB+C*DE-FG
)	(AB+C- (D-E-^(F+G+	AB+C*DE-FG+
-	(AB+C- (D-E-^(F+G+)-	AB+C*DE-FG+^
)		AB+C*DE-FG+^

ii) $A+B*(C-D)/(P-R)$

Symbol	STACK	Postfix Expression
A		A
+	+	A
B	+	AB
*	++	AB
(++C	AB
C	++C	ABC
-	++C-	ABC
D	++C-	ABCD
)	++	ABCD-
/	+/	ABCD-*
(+/C	ABCD-*
P	+/C	ABCD-*P
-	+/C-	ABCD-*P
R	+/C-	ABCD-*PR
)	+/	ABCD-*PR-
NULL		ABCD-*PR-/+

$AB+C*-DE^FG+.$

AND

$AB*C-D/P-R.$

Q-4: Write algorithm(s) to perform PUSH and POP operations for a stack implemented as an array-based structure.

Answer:

Algorithm for PUSH operation (Insertion):

```
Algorithm PUSH(stack, item):  
    if stack is full:  
        return "Stack Overflow"  
    increment the top pointer  
    stack[top] = item
```

Or,

1. Check if the stack is full or not.
2. If the stack is full, display an error message and terminate the program.
3. If the stack is not full, increment the top pointer by 1 and insert the new element at that position.

Algorithm for POP operation (Deletion):

```
Algorithm POP(stack):  
    if stack is empty:  
        return "Stack Underflow"  
    item = stack[top]  
    decrement the top pointer  
    return item
```

Or,

1. Check if the stack is empty or not.
2. If the stack is empty, display an error message and terminate the program.
3. If the stack is not empty, retrieve the element at the top of the stack and decrement the top pointer by 1.

Illustration of stack POP and PUSH operations:

Let's consider a stack implemented using an array:

Stack: [5, 8, 3, 2]

Initial Stack:

| 5 |

| 8 |

| 3 |

| 2 |

| |

PUSH (7) operation:

After PUSH (7):

| 5 |

| 8 |

| 3 |

| 2 |

| 7 |

| |

POP operation:

After POP:

| 5 |

| 8 |

| 3 |

| 2 |

| |

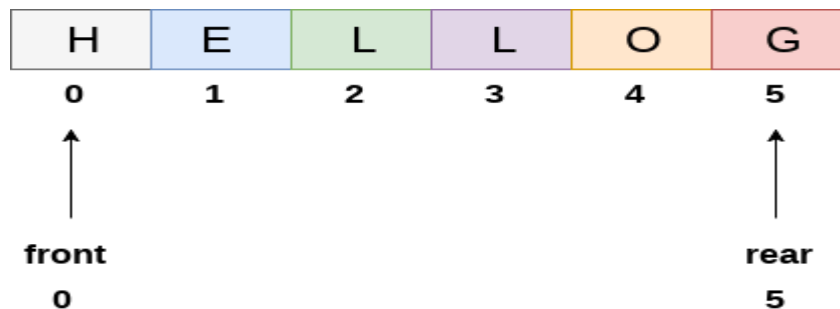
The top element (7) is removed from the stack.

Q-5: Explain Drawback of array implementation of queue and discuss about its solution.

Answer:

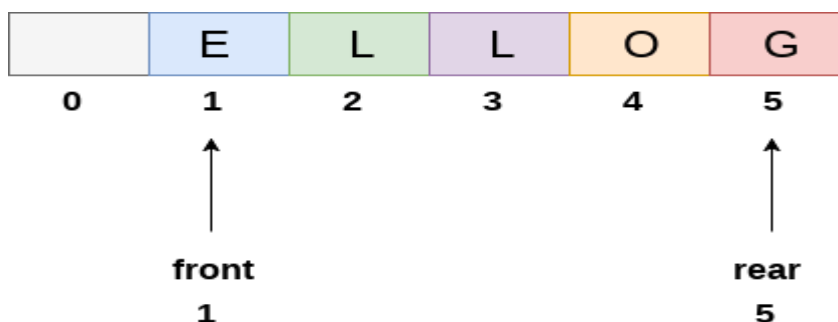
Here are some of the drawbacks of array implementation of queue:

- **Fixed size:** The size of the array must be declared in advance, which means that the queue can only store a fixed number of elements. If the queue is full, then new elements cannot be added until some elements are removed.
- **Wasted space:** If the queue is not full, then there will be wasted space in the array. This is because the elements in the queue are stored sequentially in the array, and if there are gaps between the elements, then this space cannot be used to store new element



Queue after inserting an element

- **Inefficient insertion and deletion:** Insertion and deletion of elements in an array-based queue can be inefficient, especially if the queue is full or nearly full. This is because the elements in the array must be shifted to make room for new elements or to fill in the gaps left by deleted elements.

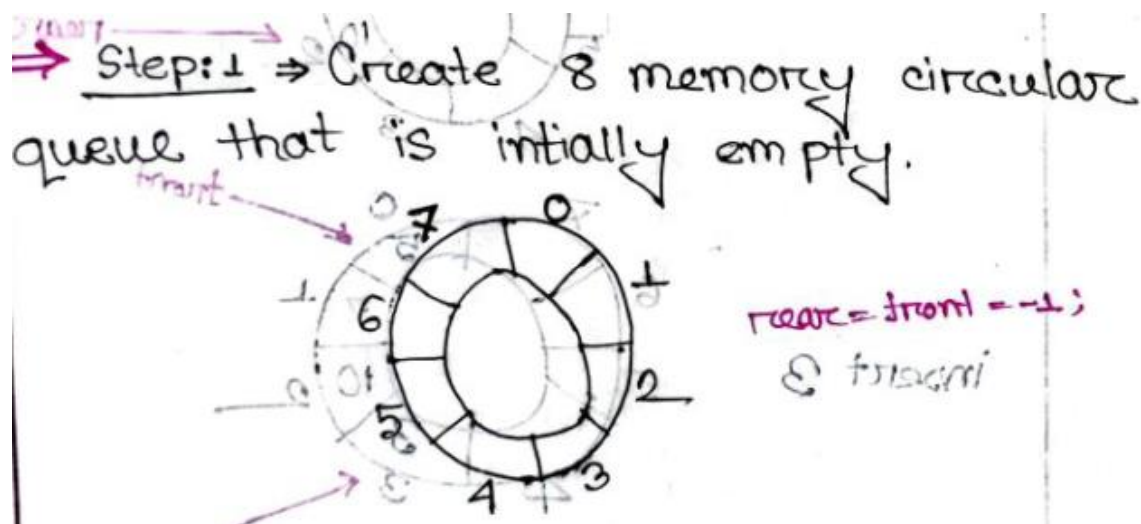


Queue after deleting an element

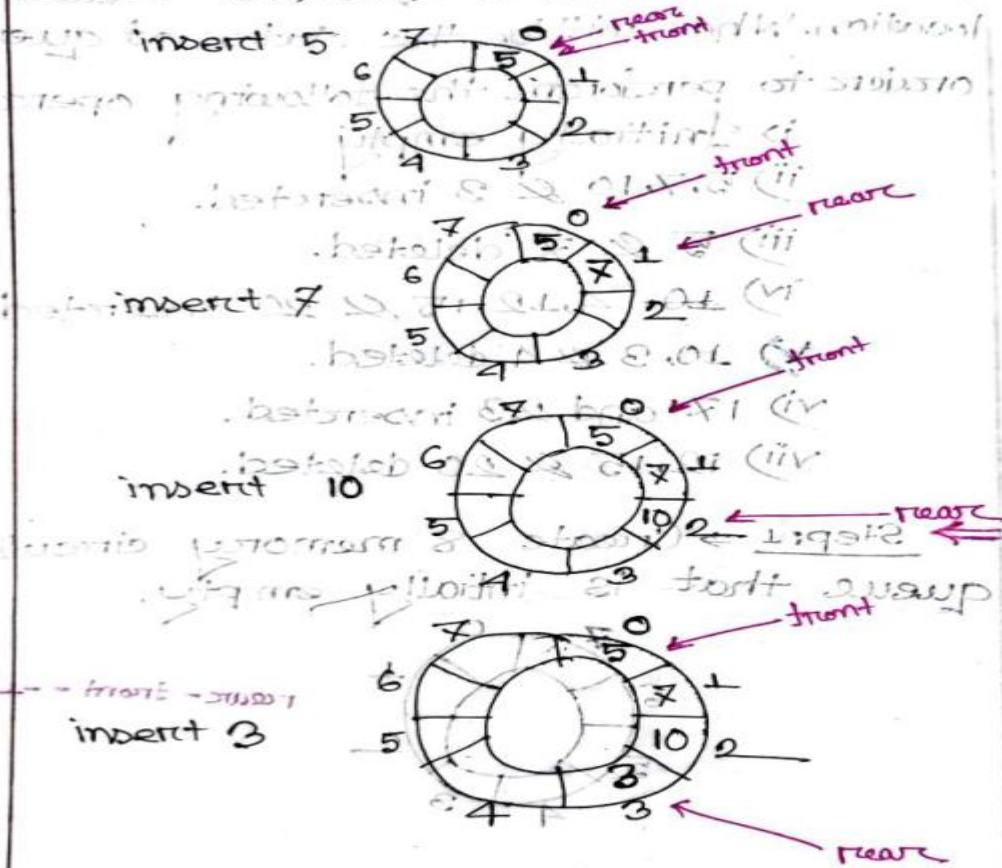
Q-6: Consider a circular queue with 8 memory locations. What will be the states of the queue in order to perform the following operations:

- i. Initially empty.**
- ii. 5, 7, 10, and 3 inserted.**
- iii. 5 and 7 deleted.**
- iv. 4, 12, 15, and 20 inserted.**
- v. 10, 3, and 4 deleted.**
- vi. 17 and 23 inserted.**
- vii. 12, 15, and 20 deleted.**

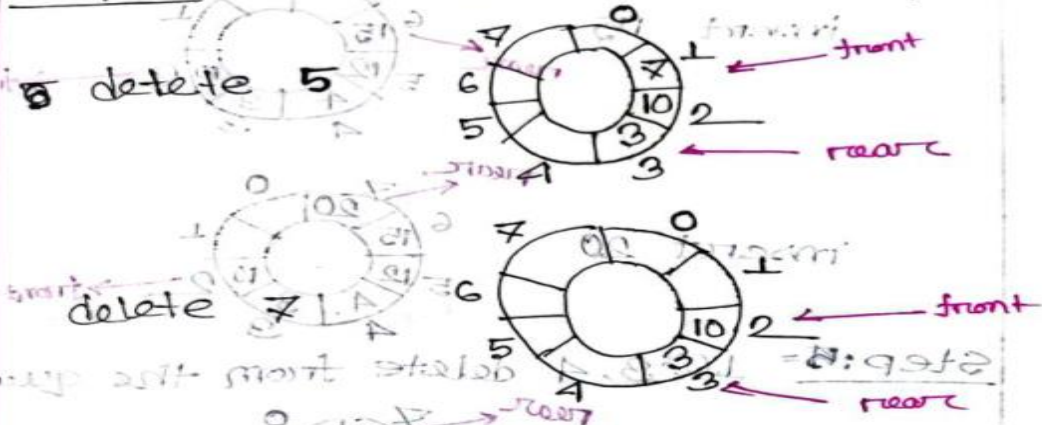
Answer:



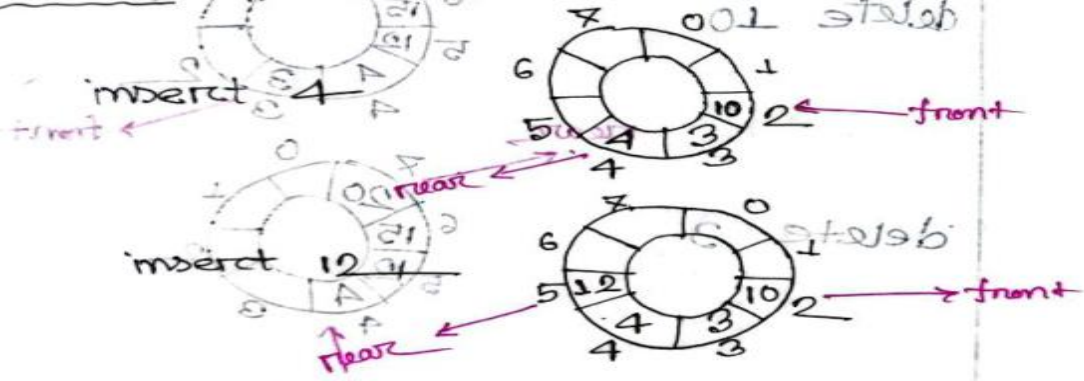
Step: 2 = 5, 7, 10 & 3 insert in the queue

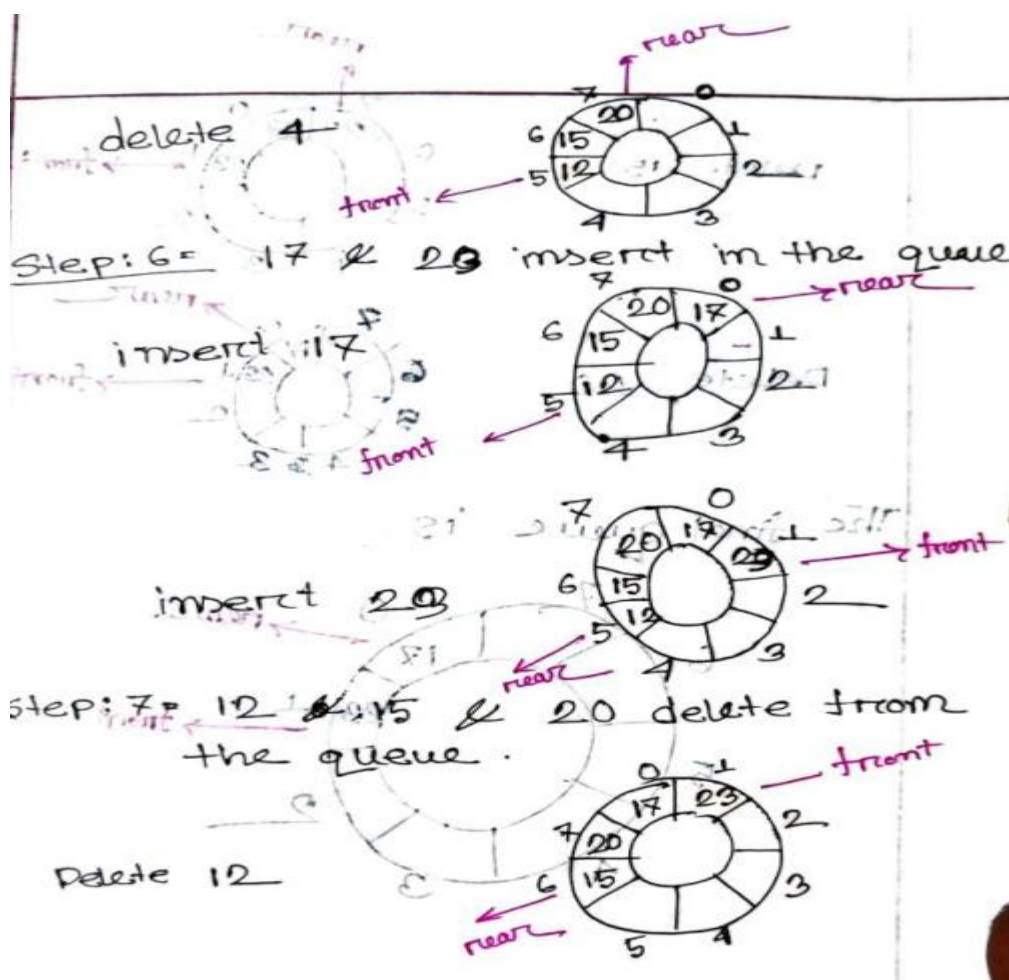
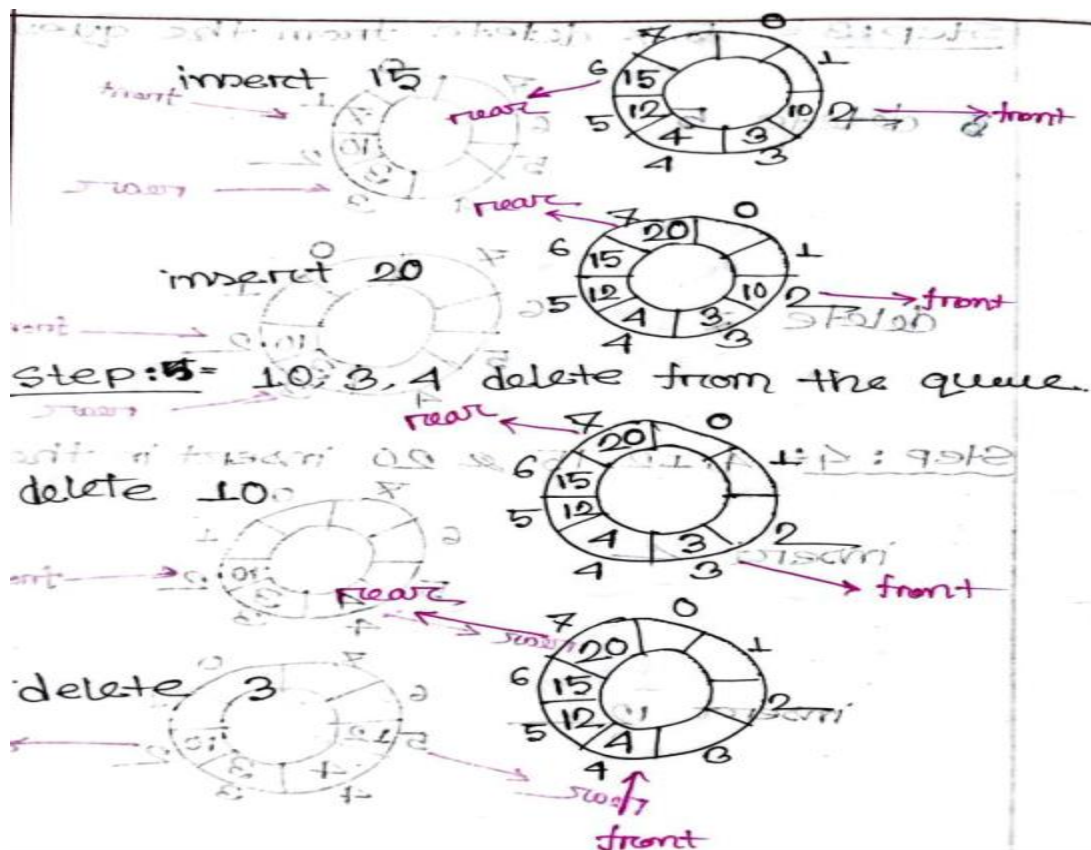


Step: 3 = 5, 7 delete from the queue.

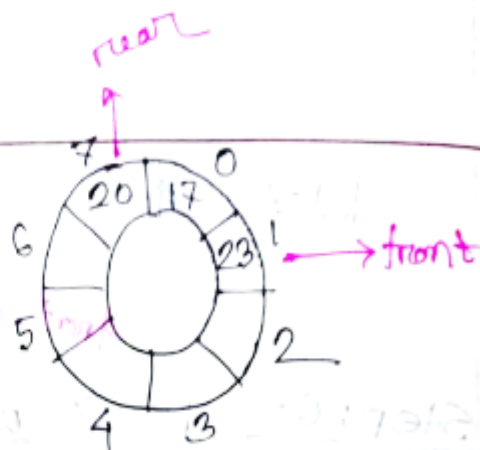


Step: 4 = 4, 12, 15 & 20 insert in the queue

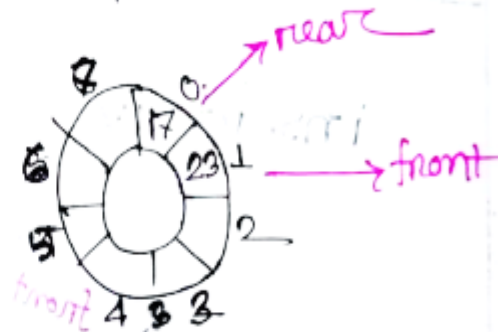




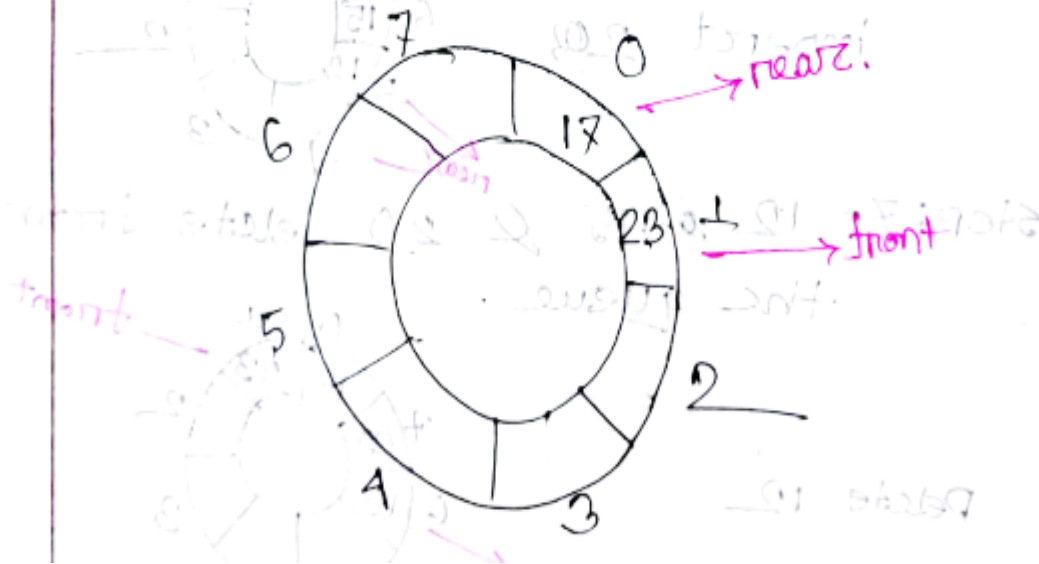
Delete 15



Delete 20



The final queue is



Q-7: Explain stack and queue operations with example. Algorithm for Stack and Queue operations.

Answer:

Stack: A stack is a linear data structure in which elements are inserted and removed from one end only. This end is called as top of the stack.

Stack operations:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove the element from the top of the stack.
- **Peek/POP:** Return the element at the top of the stack without removing it.
- **IsEmpty:** Check if the stack is empty or not.
- **IsFull:** Check if the stack is full or not.

Queue: A queue is a linear data structure in which elements are inserted at one end called rear and deleted from the other end called front.

Queue operations:

- **Enqueue:** Add an element to the rear end of the queue.
- **Dequeue:** Remove the element from the front end of the queue.
- **Peek:** Return the element at the front of the queue without removing it.
- **IsEmpty:** Check if the queue is empty or not.
- **IsFull:** Check if the queue is full or not.

Algorithm for Stack PUSH/ Pop operation:

Push(element):

1. If the stack is full, display an error message and terminate the program.
2. Else, increment the top pointer by 1.
3. Insert the new element at the position pointed to by the top pointer.
4. Algorithm for Stack POP operation:

Pop():

1. If the stack is empty, display an error message and terminate the program.
2. Else, retrieve the element at the position pointed to by the top pointer.
3. Decrement the top pointer by 1.

Algorithm for Queue ENQUEUE/ DEQUEUE operation:

Enqueue(element):

1. If the queue is full, display an error message and terminate the program.
2. Else, insert the new element at the rear end of the queue.
3. Increment the rear pointer by 1.

Dequeue ():

1. If the queue is empty, display an error message and terminate the program.
2. Else, retrieve the element at the front end of the queue.
3. Increment the front pointer by 1.

Q-8: Using algorithms translate to postfix expression and evaluate the following infix expression:

$$P=12/(7-3)+2*(1+5)$$

Answer: Follow sir note.....

the equivalent postfix expression is: 12 7 - 3 / 2 1 5 + * +.

Q-9: Consider the following stack of characters where STACK is allocated N=8 memories: STACK: A, C, D, F, K, _, _

- i. POP (STACK, ITEM).
- ii. POP (STACK (, ITEM).
- iii. PUSH (STACK, L).
- iv. POP (STACK, P).
- v. POP (STACK, ITEM).

Answer:

- i. POP (STACK, ITEM): Removes the top element from the stack and returns its value. In this case, the top element is "C", so the result of the operation is "C". The updated stack will be: STACK = A, D, F, K, _, _.
- ii. POP (STACK, ITEM): This code snippet is invalid syntax and will cause an error.
- iii. PUSH (STACK, L): Adds the element "L" to the top of the stack. The updated stack will be: STACK = A, C, D, F, K, _, _, L.
- iv. POP (STACK, P): Removes the top element from the stack and returns its value. In this case, the top element is "K", so the result of the operation is "K". The updated stack will be: STACK = A, C, D, F, _, _.
- v. POP (STACK, ITEM): Removes the top element from the stack and returns its value. In this case, the top element is "L", so the result of the operation is "L". The updated stack will be: STACK = A, C, D, F, _, _.

Q-10: Define Queue and Deques. Write down the two mandatory properties of a recursive function.

Answer:

A queue is a linear data structure that follows the First In First Out (FIFO) principle. It has two main operations, enqueue which adds an element to the rear end of the queue and dequeue which removes an element from the front end of the queue.

A deque (double-ended queue), on the other hand, is also a linear data structure that allows insertion and deletion at both ends. It can be thought of as a combination of stacks and queues as it allows for both LIFO (Last In First Out) and FIFO (First In First Out) operations.

The two mandatory properties of a recursive function are:

- **A base case:** The recursive function must have a terminating condition, which is known as the base case. Once the base case is reached, the recursion stops, and the function returns a value without any further recursive calls.
- **A recursive call:** The recursive function must call itself with a different set of arguments in each iteration until it reaches the base case. This allows the function to break down a complex problem into smaller sub-problems that can be solved using the same algorithm.

Q-11: Write down the disadvantages of stack and Queue.

Answer:

Here are some of the disadvantages of stacks and queues:

- Stacks:
 - LIFO order: Stacks follow the Last In First Out (LIFO) order, which means that the last element inserted into the stack is the first element that is removed. This can be inefficient for some applications, such as searching for an element in the stack.
 - Limited space: Stacks are limited in space, as they can only store a fixed number of elements. This can be a problem if the stack is used to store a large amount of data.
- Queues:
 - FIFO order: Queues follow the First In First Out (FIFO) order, which means that the first element inserted into the queue is the first element that is removed. This can be inefficient for some applications, such as real-time applications where elements need to be processed in a specific order.
 - Limited space: Queues are limited in space, as they can only store a fixed number of elements. This can be a problem if the queue is used to store a large amount of data.

Here are some additional disadvantages of stacks and queues:

- Stacks and queues are not efficient for random access: Stacks and queues are only efficient for sequential access, meaning that you can only access elements in the order in which they were inserted. This can be a problem for some applications that require random access to elements.
- Stacks and queues are not thread-safe: Stacks and queues are not thread-safe, meaning that they cannot be accessed by multiple threads at the same time without causing race conditions. This can be a problem for some applications that need to be accessed by multiple threads.

Q-12: Difference between Stack and Queue.

Answer:

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.

Feature	Stack	Queue
Order	LIFO (Last In First Out)	FIFO (First In First Out)
Operations	Push (add element to the top), Pop (remove element from the top)	Enqueue (add element to the rear), Dequeue (remove element from the front)
Applications	Undo/Redo, function calls, backtracking, expression evaluation	File processing, printing, simulation, resource management
Space complexity	$O(n)$	$O(n)$
Time complexity	$O(1)$	$O(1)$

Q-13: Describe about Circular Queue & Priority Queue.

Answer:

Here are some descriptions of circular queues and priority queues:

Circular Queue

A circular queue is a data structure that uses a single, fixed-size buffer to store elements. The elements in the queue are arranged in a circular fashion, so that the end of the queue is connected to the beginning of the queue. This allows the queue to be accessed in a FIFO (first-in, first-out) order, even if the queue is full.

Priority Queue

A priority queue is a data structure that stores elements in a sorted order, based on their priority. The element with the highest priority is always at the front of the queue. This allows elements to be processed in the order of their priority, which can be useful for applications where some elements need to be processed before others.

Here are some of the advantages of circular queues:

- **Efficient use of space:** A circular queue uses a single, fixed-size buffer, which can be more efficient than using multiple buffers.
- **Simple to implement:** Circular queues are relatively simple to implement, which can make them a good choice for applications where performance is not critical.

Here are some of the advantages of priority queues:

- **Efficient processing:** Priority queues can be used to efficiently process elements in the order of their priority.
- **Flexible:** Priority queues can be used to store elements of different types, as long as they can be compared.

Related problem:

Consider a circular queue with 8 memory locations. What will be the states of the queue in order to perform the following operations:

- i. Initially empty.
- ii. 5, 7, 10, and 3 inserted.
- iii. 5 and 7 deleted.
- iv. 4, 12, 15, and 20 inserted.
- v. 10, 3, and 4 deleted.
- vi. 17 and 23 inserted.
- vii. 12, 15, and 20 deleted.

Let's go through each operation step by step:

i. Initially empty:(1 to 7)

Queue: []

ii. 5, 7, 10, and 3 inserted:

Queue: [5, 7, 10, 3, _, _, _, _]

iii. 5 and 7 deleted:

Queue: [_, _, 10, 3, _, _, _, _]

iv. 4, 12, 15, and 20 inserted:

Queue: [_, _, 10, 3, 4, 12, 15, 20]

v. 10, 3, and 4 deleted:

Queue: [_, _, _, _, 12, 15, 20]

vi. 17 and 23 inserted:

Queue: [17, 23, _, _, 12, 15, 20]

vii. 12, 15, and 20 deleted:

Queue: [17, 23, _, _, _, _, _]

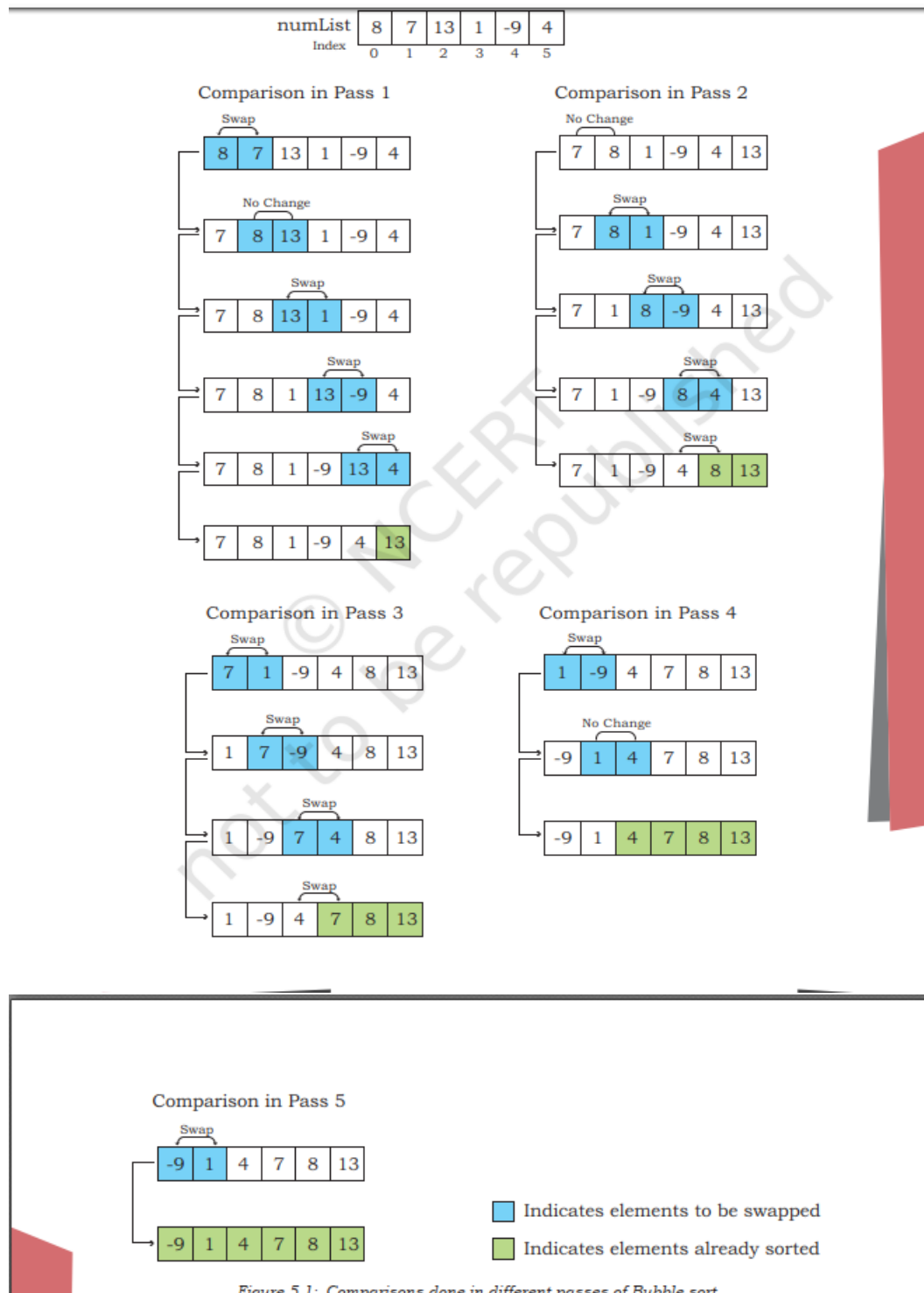
SORTING & SEARCHING

Q-1: Show the step in BUBBLE SORT algorithms for the following data-

39,52,29,81,67,22,15,59

Answer: Follow this Fig.

Comparison for Bobble sort:



Algorithm for Bobble Sort

Activity 5.1

Algorithm 5.1 sorts a list in ascending order. Write a bubble sort algorithm to sort a list in descending order?



Algorithm 5.1: Bubble Sort

BUBBLESORT(numList, n)

Step 1: SET $i = 0$

Step 2: WHILE $i < n$ REPEAT STEPS 3 to 8

Step 3: SET $j = 0$

Step 4: WHILE $j < n - i - 1$, REPEAT STEPS 5 to 7

Step 5: IF numList[j] > numList[j+1] THEN

Step 6: swap(numList[j], numList[j+1])

Step 7: SET $j = j + 1$

Step 8: SET $i = i + 1$

Q-2: Write Selection sort & Insertion algorithms and explain it, with example.

Answer:

Algorithm for Inserting Sort

Activity 5.4

Consider a list of 10 elements:
Array =
[7,11,3,10,17,23,1,4,21,5]
Determine the partially sorted list after three complete passes of insertion sort.



Algorithm 5.3: Insertion Sort

INSERTIONSORT(numList, n)

Step 1: SET $i = 1$

Step 2: WHILE $i < n$ REPEAT STEPS 3 to 9

Step 3: temp = numList[i]

Step 4: SET $j = i - 1$

Step 5: WHILE $j > 0$ and numList[j] > temp, REPEAT STEPS 6 to 7

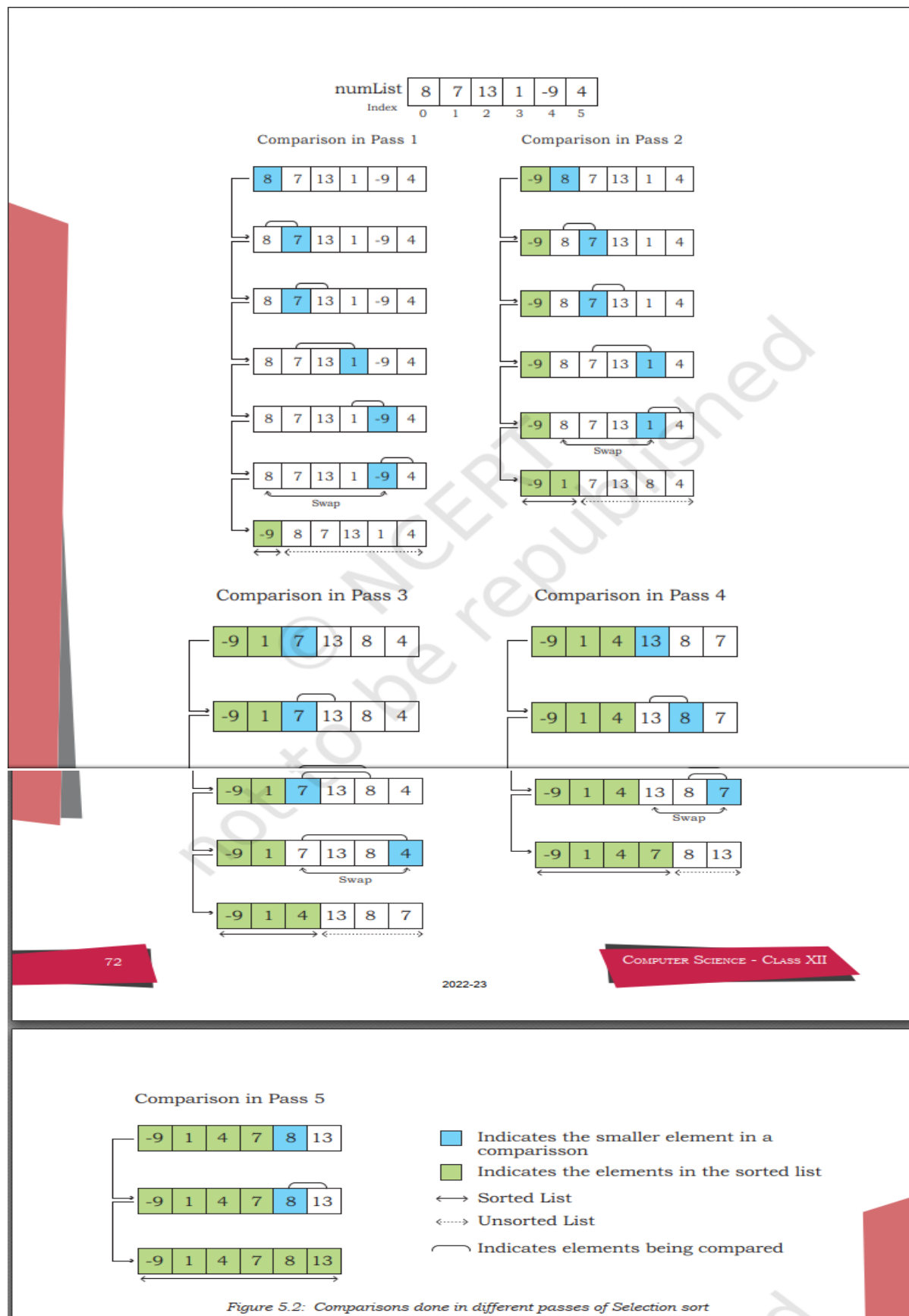
Step 6: numList[j+1] = numList[j]

Step 7: SET $j = j - 1$

Step 8: numList[j+1] = temp #insert temp at position j

Step 9: set $i = i + 1$

Comparison for Insertion sort:



Example: Let's sort the array [8, 3, 5, 2, 4] using Insertion sort.

- Pass 1: [3, 8, 5, 2, 4]
- Pass 2: [3, 5, 8, 2, 4]
- Pass 3: [2, 3, 5, 8, 4]
- Pass 4: [2, 3, 4, 5, 8]

Algorithm for Selection Sort

Algorithm 5.2: Selection Sort

SELECTIONSORT(numList, n)

Step 1: SET $i=0$

Step 2: WHILE $i < n$ REPEAT STEPS 3 to 11

Step 3: SET $\text{min} = i$, $\text{flag} = 0$

Step 4: SET $j = i+1$

Step 5: WHILE $j < n$, REPEAT STEPS 6 to 10

Step 6: IF $\text{numList}[j] < \text{numList}[\text{min}]$ THEN

Step 7: $\text{min} = j$

Step 8: $\text{flag} = 1$

Step 9: IF $\text{flag} = 1$ THEN

Step 10: $\text{swap}(\text{numList}[i], \text{numList}[\text{min}])$

Step 11: SET $i=i+1$

Activity 5.3

Consider a list of 10 elements:
 $\text{randList} = [7, 11, 3, 10, 17, 23, 1, 4, 21, 5]$.
Determine the partially sorted list after four complete passes of selection sort.

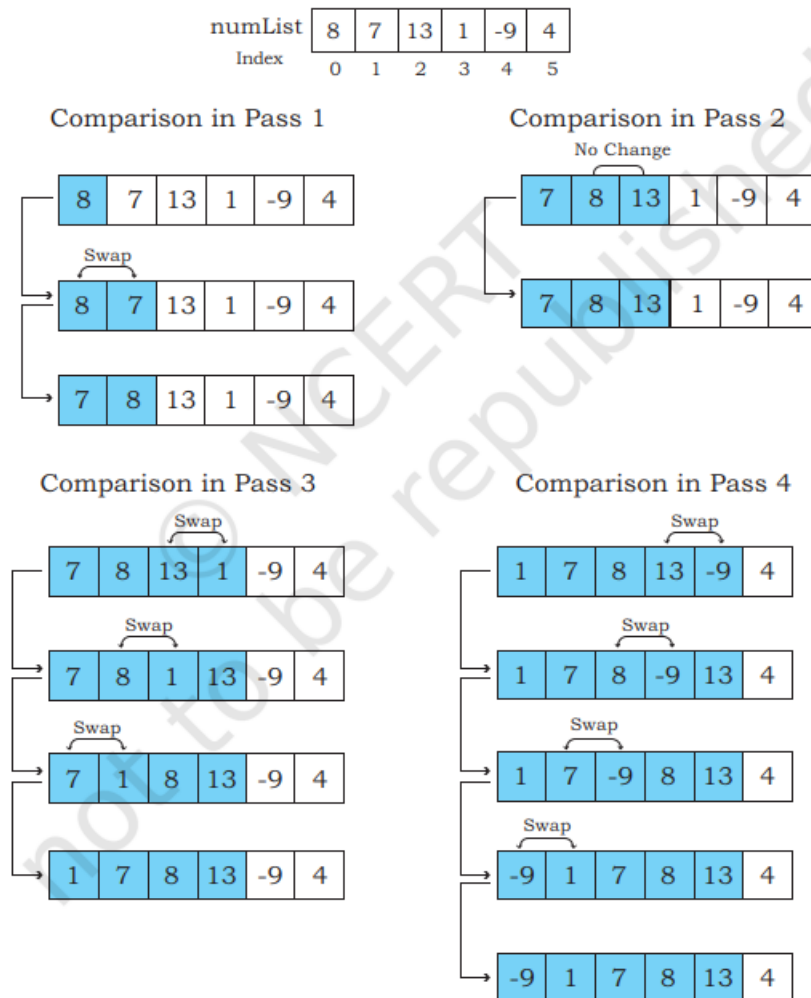


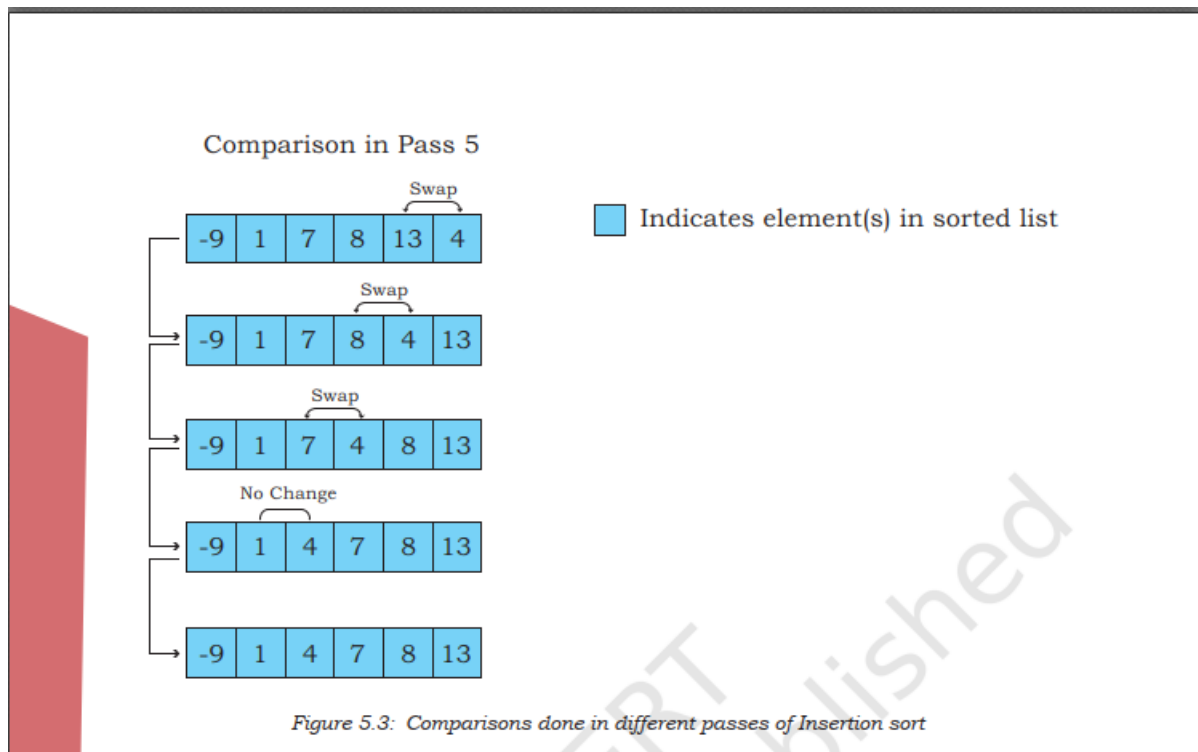
Example:

Example: Let's sort the array [8, 3, 5, 2, 4] using Selection sort.

- Pass 1: [2, 3, 5, 8, 4]
- Pass 2: [2, 3, 5, 8, 4]
- Pass 3: [2, 3, 4, 8, 5]
- Pass 4: [2, 3, 4, 5, 8]

Comparison for Selection sort:





Q-3: How many numbers of swapping needed to sort the characters "PEOPLE" / 8,22,7,9,31,19,5,13 in ascending/descending order, using BUBBLI SORT algorithm.

Answer:

To sort the characters "PEOPLE" in ascending order using Bubble sort algorithm, we need 9 swaps. The steps are as follows:

Pass 1: [E, P, L, O, E]

Pass 2: [E, L, P, E, O]

Pass 3: [E, L, E, P, O]

Pass 4: [E, E, L, O, P]

Pass 5: [E, E, L, O, P]

To sort the characters "PEOPLE" in descending order using Bubble sort algorithm, we need 10 swaps. The steps are as follows:

Pass 1: [P, E, O, L, E]

Pass 2: [P, O, L, E, E]

Pass 3: [P, L, O, E, E]

Pass 4: [P, L, O, E, E]

Pass 5: [P, L, O, E, E]

Something wrong 😊

Q-4: Explain the technique for Merge Sort Algorithms with Example.

Answer:

Merge Sort Algorithms:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid= (left+right)/2

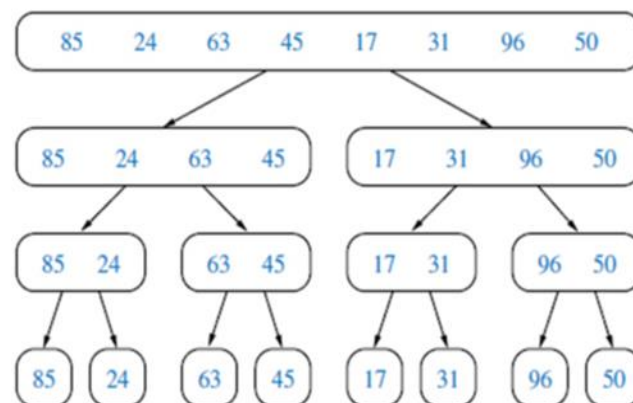
mergesort(array, left, mid)

mergesort(array, mid+1, right)

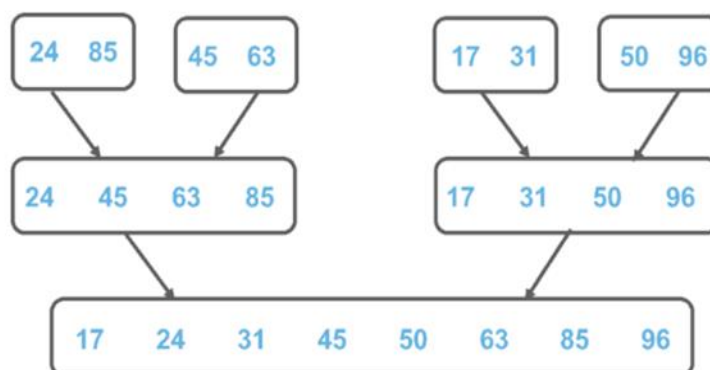
merge(array, left, mid, right)

step 4: Stop

Comparison for merge sort:



2. Recursively, merge sub-arrays to produce sorted sub-arrays until all the sub-array merges and only one array remains.



To sort an array using Merge sort, following is the process

Q-5: Suppose you have an array of following element:44,22,77,33,99,11,66,55.

Explain the technique for Selection Sort Algorithms from above element.

Answer: (Follow 2 no. question)

The array is now sorted: [11, 22, 33, 44, 55, 66, 77, 99]

Q-6: Comparison __ Selection sort & Insertion& All sort.

Answer:

Algorithm	Time Complexity	Space Complexity
Bubble sort	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n)$
Quick sort	$O(n \log n)$	$O(\log n)$

As you can see, all of the sorting algorithms have a worst-case time complexity of $O(n^2)$. However, the merge sort and quick sort algorithms have a better average-case time complexity of $O(n \log n)$.

The bubble sort, selection sort, and insertion sort algorithms are all simple to understand and implement. However, they are not very efficient for large datasets. The merge sort and quick sort algorithms are more efficient for large datasets, but they are more complex to understand and implement.

Q-7: Analyse the complexity of Linear search & Binary search Algorithm.

Answer:

Algorithm	Best Case	Worst Case	Average Case	Space Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$

As you can see, linear search has a best-case time complexity of $O(1)$, which means that it can find the element in the array in constant time. However, its worst-case time complexity is $O(n)$, which means that it can take up to n comparisons to find the element in the array. The average case time complexity of linear search is also $O(n)$.

Binary search has a best-case, worst-case, and average case time complexity of $O(\log n)$. This means that it can find the element in the array in logarithmic time, which is much faster than linear search. However, binary search can only be used to search for elements in sorted arrays.

Both linear search and binary search have a space complexity of $O(1)$. This means that they only require constant space to operate.

LINEAR SEARCH VERSUS BINARY SEARCH

LINEAR SEARCH	BINARY SEARCH
An algorithm to find an element in a list by sequentially checking the elements of the list until finding the matching element	An algorithm that finds the position of a target value within a sorted array
Also called sequential search	Also called half-interval search and logarithmic search
Time complexity is $O(N)$	Time complexity is $O(\log_2 N)$
Best case is to find the element in the first position	Best case is to find the element in the middle position
It is not required to sort the array before searching the element	It is necessary to sort the array before searching the element
Less efficient	More efficient
Less complex	More complex
	Visit www.PEDIAA.com

Q-8: Write the algorithms/pseudocode with comparison for Binary search & Linear search

Answer:

Binary Search Algorithm:

Step 1: Set low to the first index and high to the last index of the array

Step 2: Repeat Steps 3-5 while low doesn't exceed high

Step 3: Find the middle element of the array using the formula $\text{mid} = (\text{low} + \text{high}) / 2$

Step 4: If the middle element matches the target value, return its index

Step 5: If the middle element is greater than the target value, set high to $\text{mid} - 1$; if it's less than the target value, set low to $\text{mid} + 1$

Step 6: If the target value is not found, return "not found"

Linear Search Algorithm:

Step 1: Loop through the array starting from the first element

Step 2: Compare each element with the target value

Step 3: If the target value is found, return its index

Step 4: If the end of the array is reached and the target value is not found, return "not found"

Binary Search algorithm pseudocode:

Binary Search (L, N, KEY)

1) Set Loc: =0, Beg: = 1, End=N and $\text{mid} = (\text{Beg} + \text{End}) / 2$

2) Repeat steps 3 to 6 while $\text{beg} \leq \text{End}$

3) If $\text{key} < L[\text{mid}]$, then set $\text{End} = \text{mid} - 1$

4) Else if $\text{key} > L[\text{mid}]$ then set $\text{Beg} = \text{mid} + 1$

5) Else if $\text{key} = L[\text{mid}]$ then Loc: = mid, print: loc and exit

6) set $\text{mid} = (\text{Beg} + \text{End} / 2)$

7) if loc=0 write: item is not in list

08) Exit.

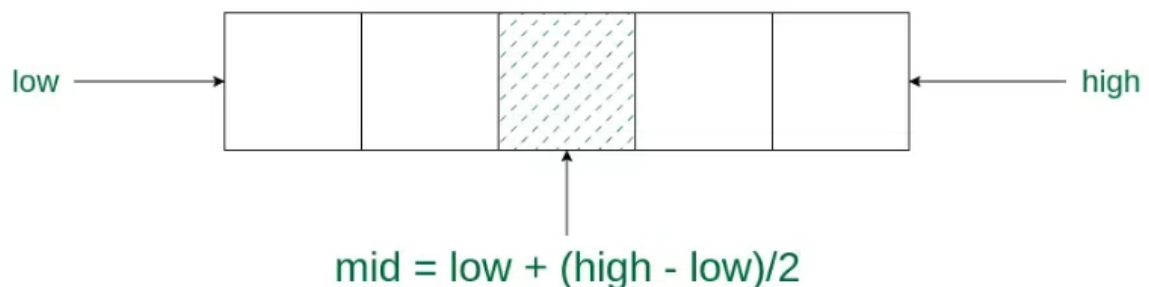
Linear Search algorithm pseudocode:

01. Set k=1 and Loc=0
02. Repeat steps 3 and 4 while K<=N.
03. If key = L[K] then set Loc: =K,
Print: Loc and exit
04. Set k: =K+1
05. If Loc= 0 Then write: them is not in list.
06. Exit. set

Q-9: Method/Technique of Binary Search.

Answer:

Binary Search is defined as a [searching algorithm](#) used in a sorted array by **repeatedly dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log N)$.

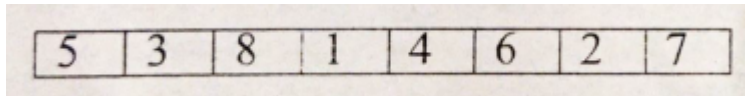


Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found

Q-10: Show the Successive step for finding 45 & 85 using binary search algorithms 17,19,28,30,45,55,58,61,63,67,72,76,80,89

Answer:

Q-11: Explain Quick sort for the following unsorted list:



5	3	8	1	4	6	2	7
---	---	---	---	---	---	---	---

Answer:

HEAP

Q-1: Define Heap. Write down the properties of max-heap and min-heap with an example.

Answer:

Heap is a specialized tree-based data structure that satisfies the heap property. A heap can be implemented as an array or a binary tree.

Properties of a max-heap:

- Max-Heap Property: In a max-heap, for any given node i , the value of node i is greater than or equal to the values of its children.
- Example: [90, 85, 70, 80, 75, 60, 45]
- Complete Binary Tree: A max-heap is a complete binary tree, meaning all levels of the tree are fully filled except possibly the last level, which is filled from left to right.

Properties of a min-heap:

- Min-Heap Property: In a min-heap, for any given node i , the value of node i is less than or equal to the values of its children.
- Example: [10, 15, 20, 30, 35, 40, 45]
- Complete Binary Tree: A min-heap is also a complete binary tree, similar to a max-heap.

Q-2: Construct the max heap from the given set of elements by showing each step of construction {26, 33, 19, 15, 7, 35, 17, 10}

Answer:

Insert {36} in the resultant Max-heap

Show each step of Heap sort over the resultant Max-heap.

Step 1: Start with the given set of elements: {26, 33, 19, 15, 7, 35, 17, 10}

Max-Heap after insertion of each element:

[26]

[33, 26]

[33, 26, 19]

[33, 26, 19, 15]

[33, 26, 19, 15, 7]

[35, 33, 19, 15, 7, 26]

[35, 33, 19, 15, 7, 26, 17]

[35, 33, 19, 15, 7, 26, 17, 10]

Step 2: Insert {36} into the resultant Max-Heap:

[36, 33, 35, 15, 7, 26, 17, 10, 19]

Step 3: Perform Heap Sort on the resultant Max-Heap:

Heap Sort steps:

1. Swap the root element (36) with the last element (19).

[19, 33, 35, 15, 7, 26, 17, 10, 36]

2. Heap the remaining elements from the root to maintain the max-heap property.

[35, 33, 26, 15, 7, 19, 17, 10, 36]

3. Repeat steps 1 and 2 for the remaining elements.

[33, 15, 26, 10, 7, 19, 17, 35, 36]

[26, 15, 19, 10, 7, 17, 33, 35, 36]

[19, 15, 17, 10, 7, 26, 33, 35, 36]

[17, 15, 7, 10, 19, 26, 33, 35, 36]

[15, 10, 7, 17, 19, 26, 33, 35, 36]

[10, 7, 15, 17, 19, 26, 33, 35, 36]

[7, 10, 15, 17, 19, 26, 33, 35, 36]

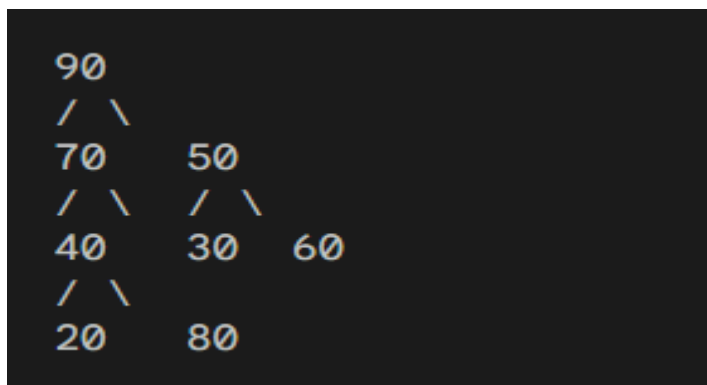
The final sorted array using Heap Sort is [7, 10, 15, 17, 19, 26, 33, 35, 36].

Q-4: Draw the Max Heap and Mean Heap Tree from the following element:

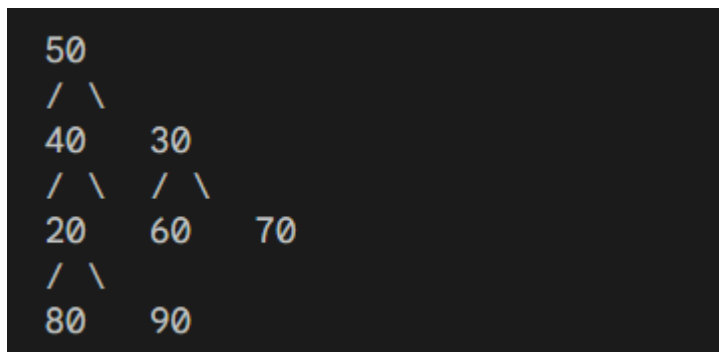
50,20,70,40,30,60,90,80

Answer: Follow sir note.....

Max Heap:



Mean Heap:



Q-5: Write an algorithm to Insert an element in the sorted Heap.

Write down the Algorithm of insert operation in the max heap.

Or, explain with example the process to insert elements into the heap.

Answer:

Algorithm to insert an element in the sorted heap:

- Append the new element to the end of the heap.
- Compare the value of the new element with its parent node.
- If the value of the new element is greater than the parent node, swap them.
- Repeat Step 3 until the heap property is satisfied.

For example:

Let's consider a min-heap with the following elements: [10, 20, 30, 40].

We want to insert the element 25 into the heap.

- ❖ Step 1: Insert 25 at the end of the heap: [10, 20, 30, 40, 25]
- ❖ Step 2: Compare 25 with its parent node (30).
 $25 < 30$, so swap 25 and 30.
Updated heap: [10, 20, 25, 40, 30]
- ❖ Step 3: Compare 25 with its parent node (20).
 $25 > 20$, no swap needed.
The element 25 is now inserted into the sorted min-heap: [10, 20, 25, 40, 30].

Q-6: Simulate values: the maxheap algorithm for the following 77, 39,95, 60, 23, 39, 91, 9,12.

And, Simulate the max heap algorithm for the following values: 67, 29, 90, 48, 12, 34, 90, 9 and 12.

Answer:

Simulation of the max heap algorithm for the following values: 77, 39, 95, 60, 23, 39, 91, 9, 12.

Step 1: Initial array: [77]

Step 2: Insert 39: [77, 39]

Step 3: Insert 95: [95, 39, 77]

Step 4: Insert 60: [95, 60, 77, 39]

Step 5: Insert 23: [95, 60, 77, 39, 23]

Step 6: Insert 39: [95, 60, 77, 39, 23, 39]

Step 7: Insert 91: [95, 60, 91, 39, 23, 39, 77]

Step 8: Insert 9: [95, 60, 91, 39, 23, 39, 77, 9]

Step 9: Insert 12: [95, 60, 91, 39, 23, 39, 77, 9, 12]

The resulting max-heap is [95, 60, 91, 39, 23, 39, 77, 9, 12].

Simulation of the max heap algorithm for the following values: 67, 29, 90, 48, 12, 34, 90, 9, and 12.

Step 1: Initial array: [67]

Step 2: Insert 29: [67, 29]

Step 3: Insert 90: [90, 29, 67]

Step 4: Insert 48: [90, 48, 67, 29]

Step 5: Insert 12: [90, 48, 67, 29, 12]

Step 6: Insert 34: [90, 48, 67, 29, 12, 34]

Step 7: Insert 90: [90, 48, 90, 29, 12, 34, 67]

Step 8: Insert 9: [90, 48, 90, 29, 12, 34, 67, 9]

Step 9: Insert 12: [90, 48, 90, 29, 12, 34, 67, 9, 12]

The resulting max-heap is [90, 48, 90, 29, 12, 34, 67, 9, 12].

Q-7: Reducing the size of the following message for the applying variable length Huffman coding_ BBBCAAADEEEEABBBACCD A.

Answer:

TREE

Q-1: What will be the depth of a complete binary tree which has 3000 nodes.

Answer:

The depth of a complete binary tree with 3000 nodes can be calculated using the formula:

$$\text{depth} = \log_2(n+1)$$

where n is the number of nodes in the complete binary tree.

Applying this formula, we get:

$$\text{depth} = \log_2(3000+1) = \log_2(3001) \approx 11.55$$

Therefore, the depth of the complete binary tree will be approximately 11 to 12 levels.

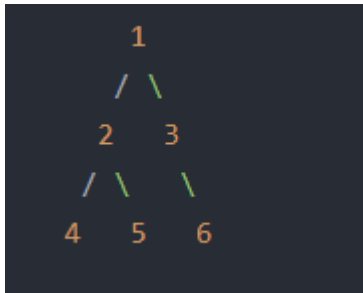
Q-2: How a binary tree can be store in computer memory using array? Explain with example.

Answer:

A binary tree can be stored in computer memory using an array by representing the binary tree as a complete binary tree and storing it in level-order traversal (also known as breadth-first traversal) in an array.

In this representation, the root of the binary tree is stored in the first element of the array, and for any node at index i, its left child is stored at index $2i+1$ and its right child is stored at index $2i+2$.

For example, consider the following binary tree:



We can represent this binary tree using an array in level-order traversal as follows:

[1, 2, 3, 4, 5, 6]

Here, the root node 1 is stored at index 0, its left child node 2 is stored at index $2 \times 0 + 1 = 1$ and its right child node 3 is stored at index $2 \times 0 + 2 = 2$. Similarly, the left child node 2 has its left child node 4 stored at index $2 \times 1 + 1 = 3$ and its right child node 5 stored at index $2 \times 1 + 2 = 4$. And the right child node 3 has its right child node 6 stored at index $2 \times 2 + 2 = 6$.

By using this representation, we can efficiently store a binary tree in computer memory using an array.

Q-3: Suppose the following sequences list the nodes of binary tree T in preorder and inorder respectively.

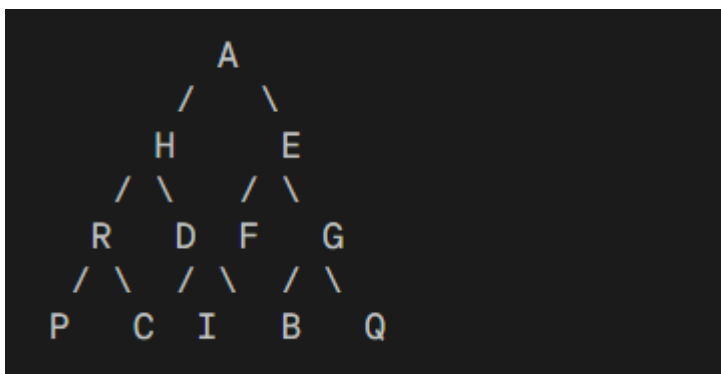
Preorder: A, H, E, F, G, B, D, C, I, R, P, Q.

Inorder: R, H, D, E, P, G, A, F, C, K, B, Q

Draw the diagram of the tree

Answer:

Diagram of the tree



The preorder traversal of a binary tree visits the nodes in the following order:

- The root node
- The left subtree of the root node
- The right subtree of the root node

The inorder traversal of a binary tree visits the nodes in the following order:

- The left subtree of the root node
- The root node
- The right subtree of the root node

Q-4: Consider the following tree

Answer: Follow sir note.....

Q-5: Define the following terms with an example:

i. Tree

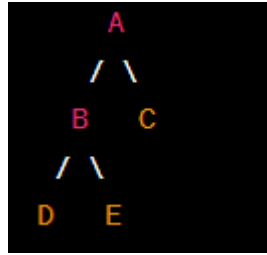
ii. Full Binary Tree

iii. Complete Binary Tree

Answer:

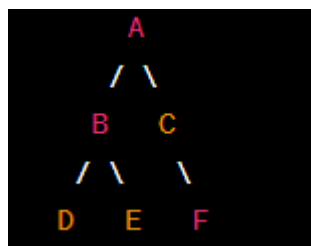
i. Tree: A tree is a hierarchical data structure consisting of nodes connected by edges. The nodes in a tree have parent-child relationships with each other, and there is only one node at the top of the hierarchy, called the root. Trees are commonly used to represent hierarchical relationships between data, such as file systems or organization charts.

Example: An example of a tree is a file system on a computer. The root node represents the main directory, and each child node represents a subdirectory or file within that directory.



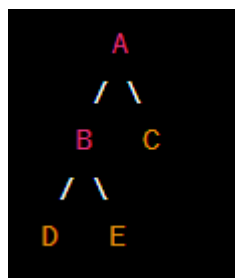
ii. Full Binary Tree: A full binary tree is a special type of binary tree where each node has either zero or two children. In other words, every node in a full binary tree has either two children or no children at all.

Example: An example of a full binary tree is a binary search tree where every node has exactly two child nodes.



iii. Complete Binary Tree: A complete binary tree is a binary tree where all levels are completely filled, except possibly for the last level, which may not be completely filled but must have all nodes as left as possible.

Example: An example of a complete binary tree is a binary heap, which is a tree-based data structure used for sorting.

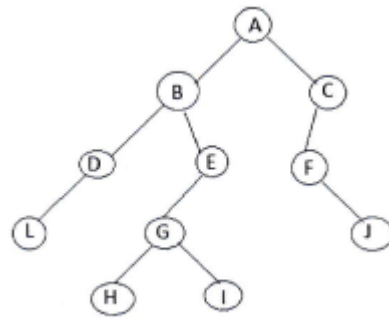


Q-6: From the following Binary Tree find the sequence of nodes When traversing:

i. Pre-Order

ii. In-Order

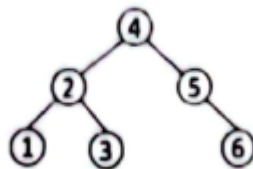
iii. post-Order



Answer:

Follow sir note.....

Q-7: Simulate the pre-order and in-order traversal of the following tree:



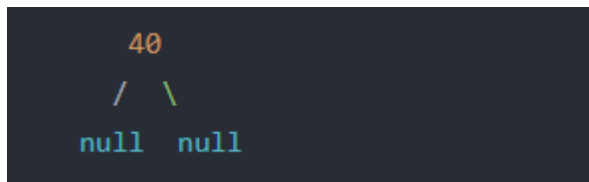
Answer:

Follow sir note.....

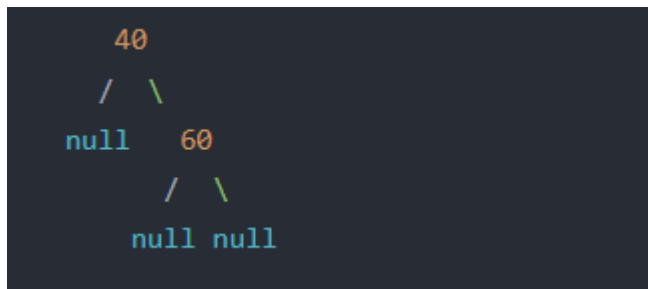
Q-8: Construct a binary search tree inserting the following numbers in order into an empty binary search tree: 40, 60, 20, 50, 55, 35.

Answer:

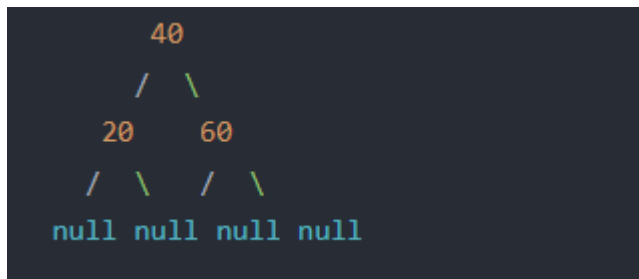
We start by creating an empty binary search tree. Then, we add the first number, 40, as the root node.



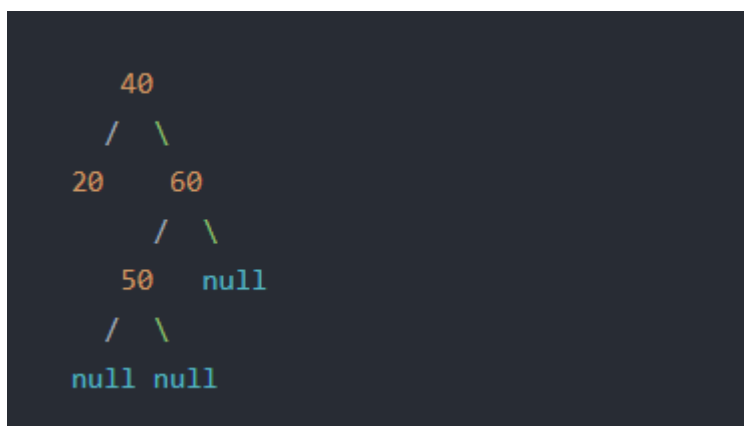
Next, we add 60 to the right of the root because it is greater than 40.



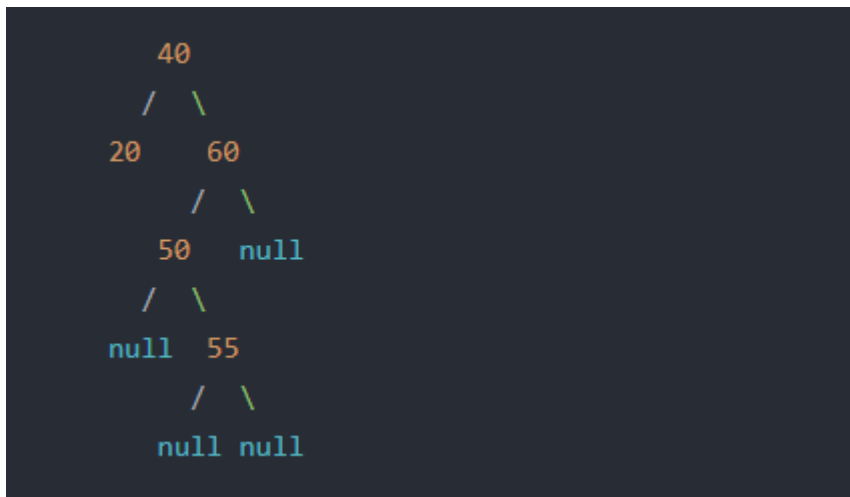
Then, we add 20 to the left of the root because it is less than 40.



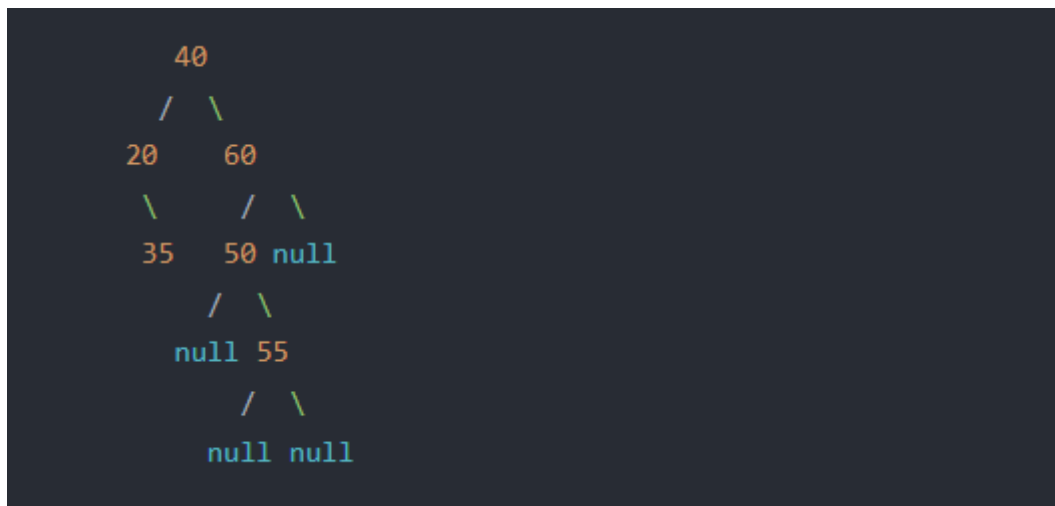
Next, we add 50 to the right of the node 60 because it is less than 60 but greater than 40.



After that, we insert 55 to the right of node 50 because it is greater than 50.



Finally, we insert 35 to the left of node 50 because it is less than 50.



And there you have it! This is the resulting binary search tree after inserting the given numbers in order.

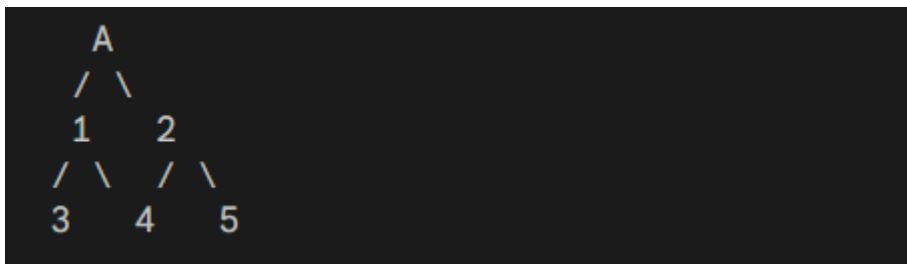
Q-9: Illustrate similar and copies of a tree with example.

Answer:

Similar trees: Two trees are similar if they have the same structure, but the nodes may have different values. For example, the following two trees are similar:



Copies of trees: Two trees are copies of each other if they have the same structure and the same values at the same nodes. For example, the following two trees are copies of each other:



Q-10: Construct a Binary tree for the expression $[(a+b*c)]-[(d*e+f)/(g-h)]$. What will be the outputs for Pre order and Post order of the expression.

Answer:

Follow sir note.....

Q-11: What is the condition for a complete Binary tree? Draw a complete Binary tree for the element of 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20.

Answer:

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Here are the conditions for a complete binary tree:

- Every node in the tree has either 0 or 2 children.
- All levels of the tree except the last one are completely filled.
- The nodes in the last level are as far left as possible.

A complete binary tree for the elements 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 is as follows:

Draw a complete Binary tree:



In this tree, all levels are completely filled except the last level, which is filled from left to right. Therefore, it satisfies the condition for a complete binary tree.

Q-12: Draw the tree diagram following expiration

$$E=[a+(b+c)]*[d-e]/(f+g)]$$

Traverse post order and preorder sequence of the expression.

Answer:

Q-13 :What is the relationship between parent and child in binary tree?

Answer:

Answer:

Suppose, a node is stored in position (index) i of an array, then the position of its left child will be $2i$ and the position of its right child will be $2i+1$. Thus

the position of its parent node will be $\left\lfloor \frac{i}{2} \right\rfloor$. $\lfloor x \rfloor$ means floor of x to the previous integer. The position of root node is 1 and its children's positions are 2 and 3. If a node value stored in position 4, then the positions of its children will be 8 and 9 and so on. When a node is stored in position 13, then the position of its parent is-

$$\left\lfloor \frac{13}{2} \right\rfloor = \lfloor 6.5 \rfloor = 6$$

Q- 14: Define leaf, complete binary tree, full binary tree.

Answer:

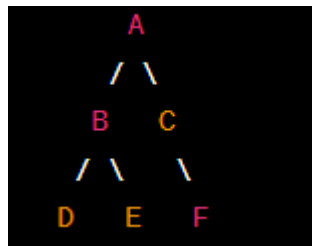
i.

- Leaf: A leaf is a node in a binary tree that has no children.
- The following is an example of a leaf:

A

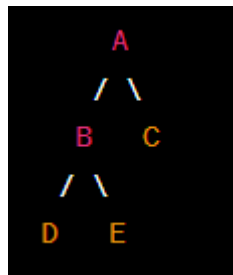
ii. Full Binary Tree: A full binary tree is a special type of binary tree where each node has either zero or two children. In other words, every node in a full binary tree has either two children or no children at all.

Example: An example of a full binary tree is a binary search tree where every node has exactly two child nodes.



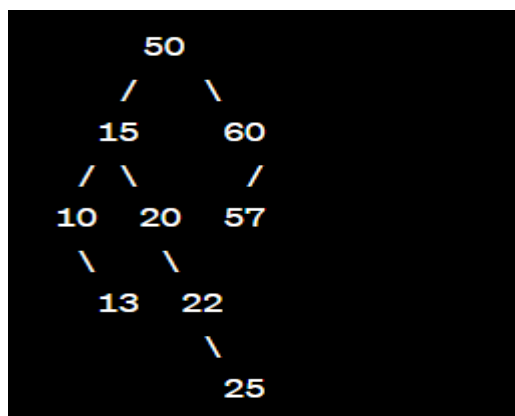
iii. Complete Binary Tree: A complete binary tree is a binary tree where all levels are completely filled, except possibly for the last level, which may not be completely filled but must have all nodes as left as possible.

Example: An example of a complete binary tree is a binary heap, which is a tree-based data structure used for sorting.



Q- 15: Construct a BST with the following list of values:
50,15,10,13,20,22,25,60,42,57

Answer:



Q-16 : Calculate the level of a tree when the number of node is 36.

Answer:

For a binary tree with n nodes, the maximum height (h) of the tree is $\log_2(n)$ if the tree is balanced. If the tree is not balanced, the height can be equal to n in the worst case (when the tree is a straight line).

In this case, we are given that the number of nodes (n) is 36. Let's calculate the level of the tree:

1. If the tree is balanced: The maximum height (h) of a balanced binary tree with 36 nodes is $\log_2(36) \approx 5.17$.

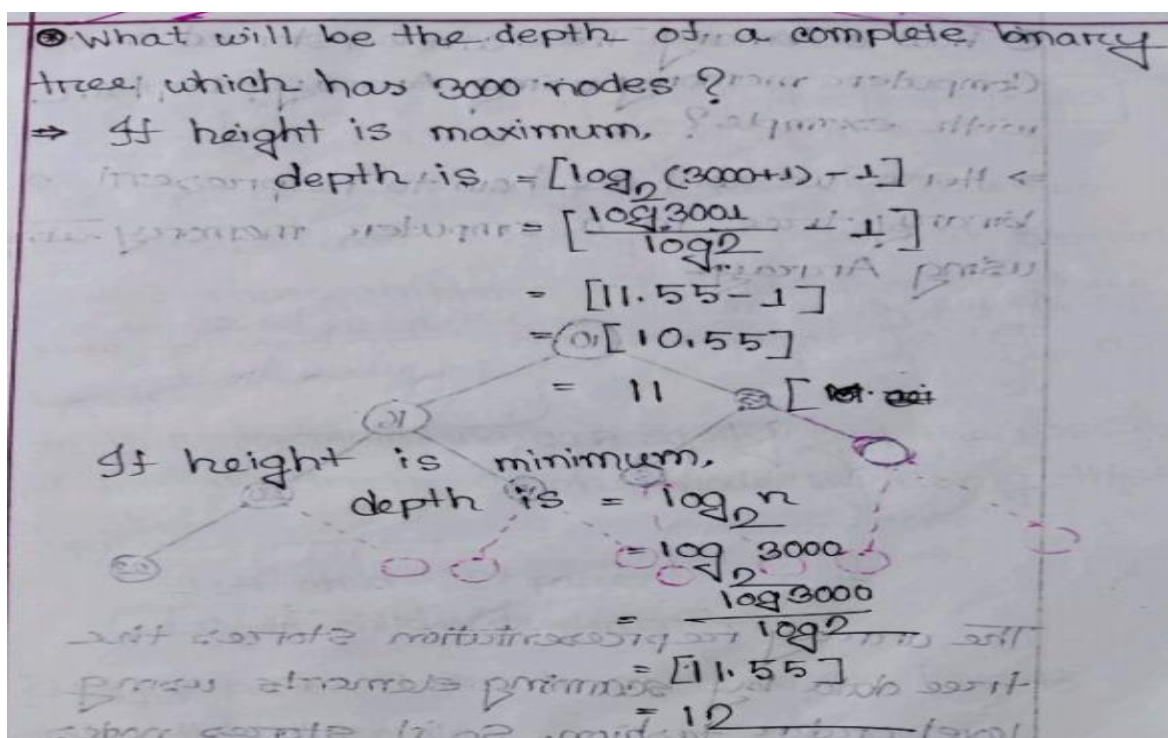
So, the level of the tree = height + 1 $\approx 5 + 1 = 6$.

2. If the tree is not balanced: The height can be equal to the number of nodes (n) in the worst case.

So, the level of the tree = height + 1 = 36 + 1 = 37.

Since the question does not specify whether the tree is balanced or not, we have two possible answers for the level of the tree:

- If the tree is balanced, the level is 6.
- If the tree is not balanced (worst case), the level is 37.

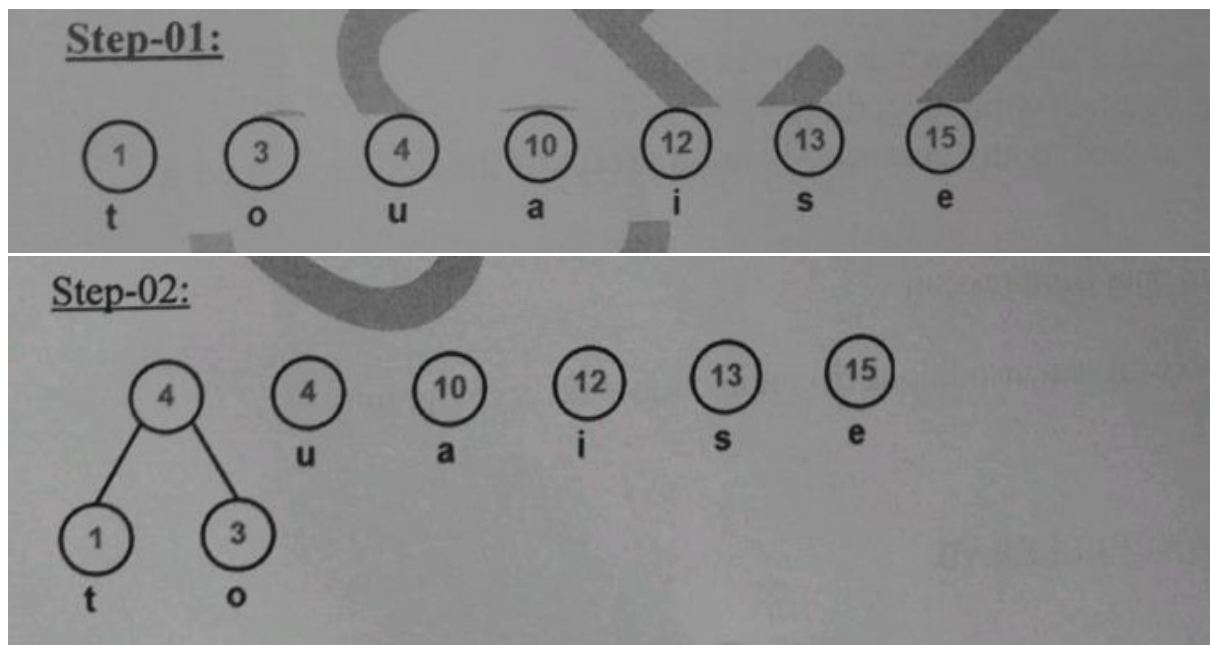


Q- 17: Build Huffman Tree from the following data and write the code-word for every character

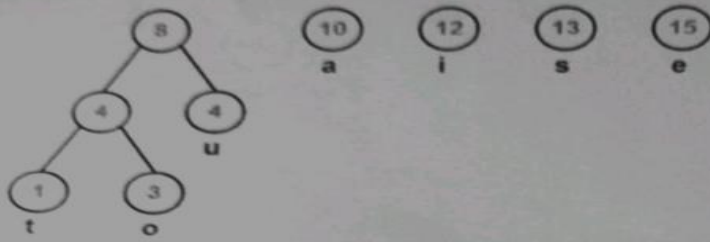
A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression. find the Huffman Code for each character.

Character	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

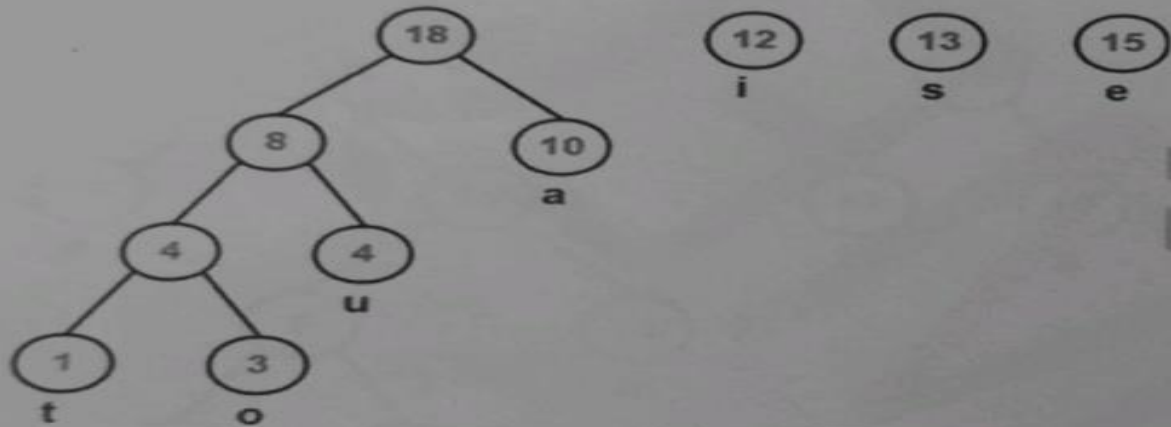
Answer:



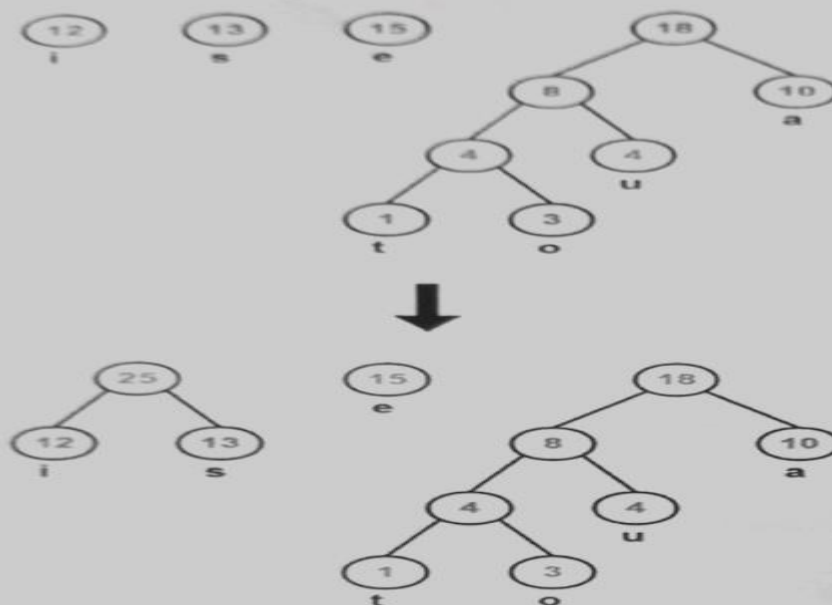
Step-03:



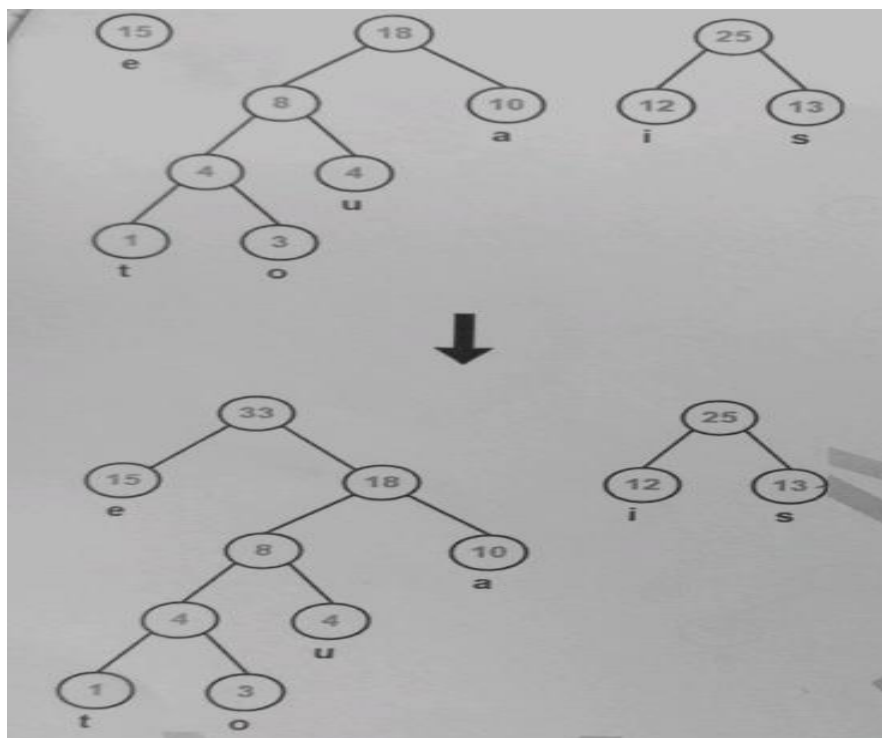
Step-04:



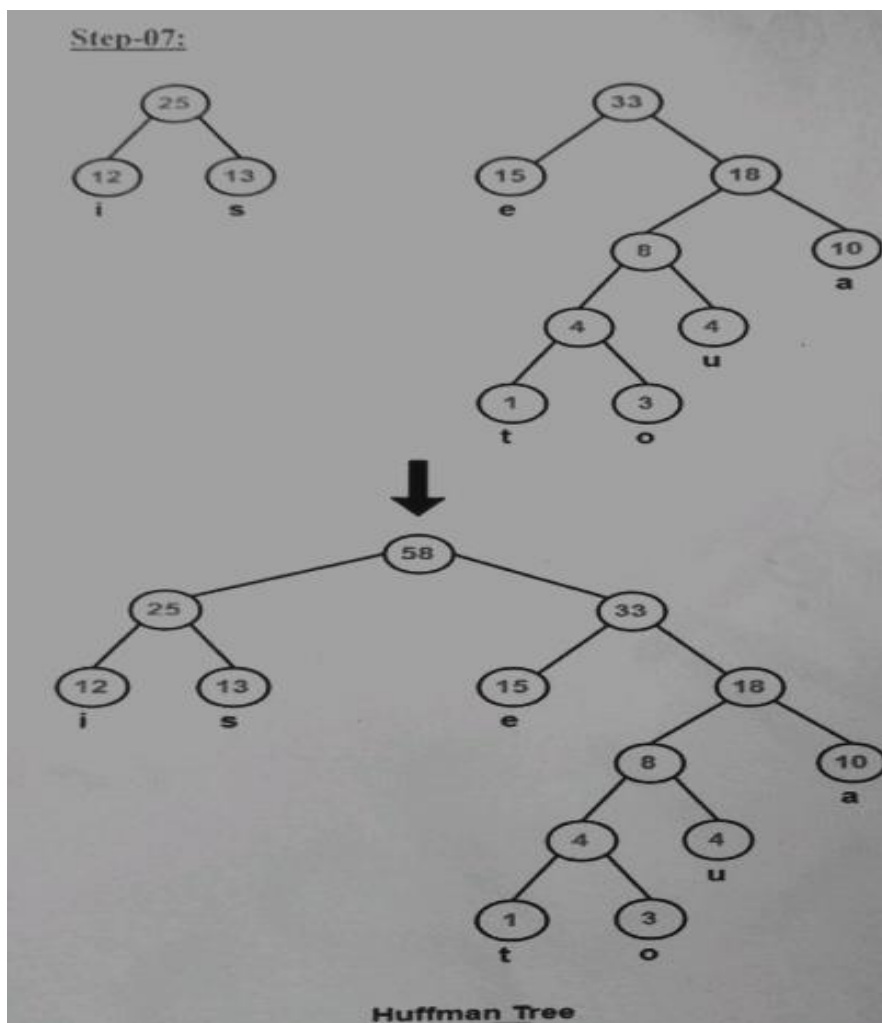
Step-05:



STEP -06:

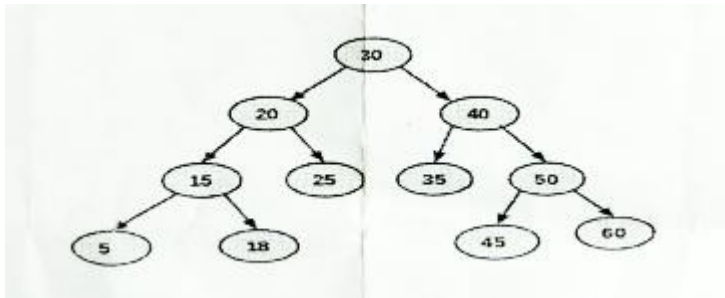


Step-07:



Q-18 : Differentiate between binary search tree and binary tree. From the following binary tree find the sequence of nodes when traversing

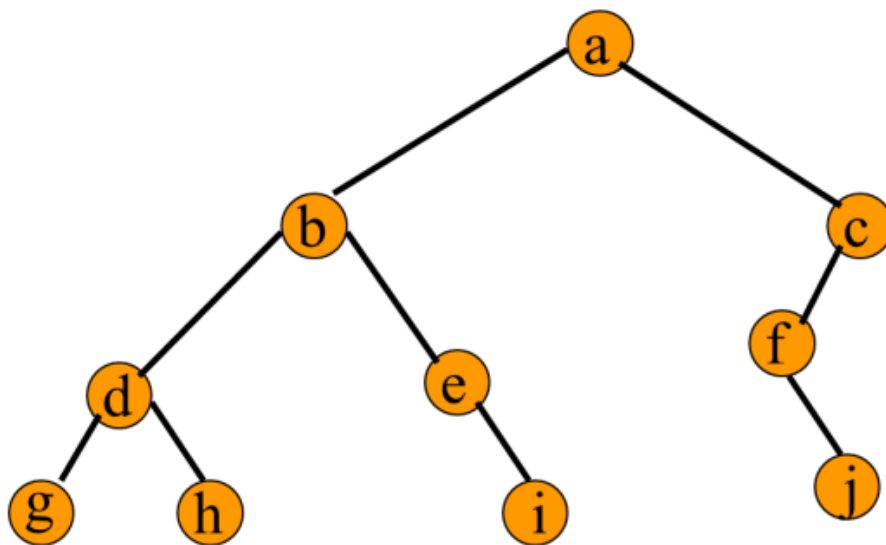
i. Pre-order ii. In-order iii. Post-order



Answer:

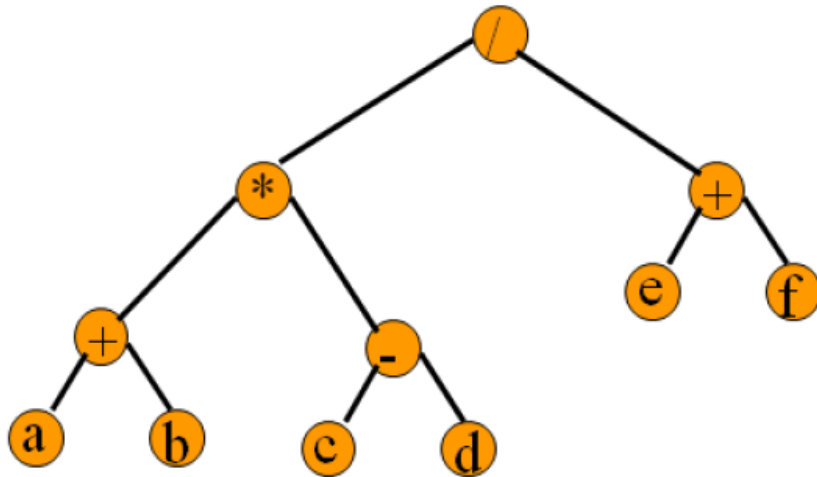
Answer:

Preorder Example



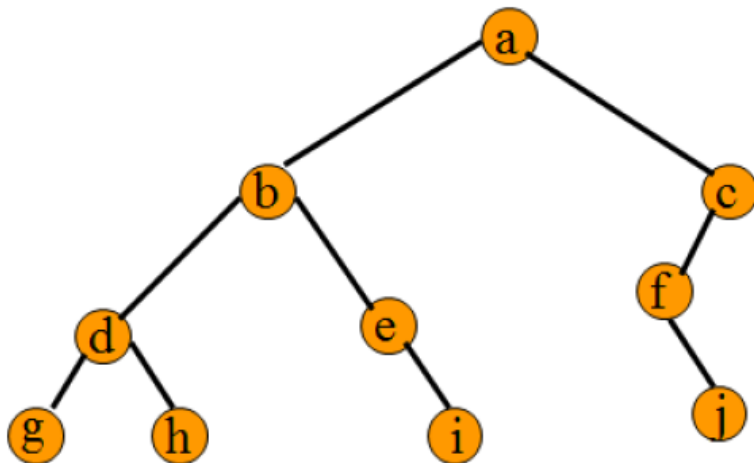
a b d g h e i c f j

Preorder Of Expression Tree



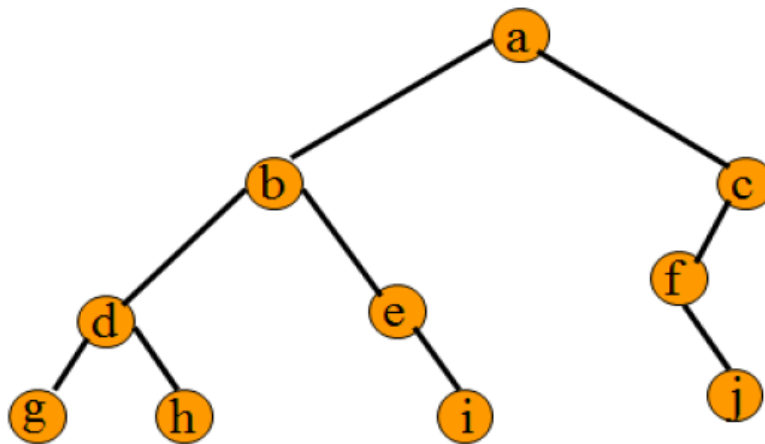
$/ \ * \ + \ a \ b \ - \ c \ d \ + \ e \ f$

Inorder Example (Visit = print)



$g \ d \ h \ b \ e \ i \ a \ f \ j \ c$

Postorder Example (Visit = print)



g h d i e b j f c a

Q-19 :Define complete binary tree. Show each step of heap sort

constructing max-heap for the following data 26,33,19,15,7,35,17,10

Answer:

Extra Some Question:

Q-1: Short notes on post order tree traversal.

Answer:

Post-order tree traversal is a method of traversing a binary tree where we visit the left subtree, then the right subtree, and finally the root node itself. This type of traversal is also known as "bottom-up" or "depth-first" traversal.

Q-2: Define the following terms: siblings, successor, ancestor, depth.

Answer:

- ✚ **Siblings:** Siblings are nodes in a tree that share the same parent.
- ✚ **Successor:** In a binary tree, the successor of a node is the node with the next larger value.
- ✚ **Ancestor:** An ancestor of a node in a tree is any parent, grandparent, great-grandparent, etc. of that node.
- ✚ **Depth:** The depth of a node in a tree is the number of edges from the root node to that node.

Q-3: Definition and construction of complete binary tree. Construction of Binary Tree from an expression and writing the output. What is the parent child relationship

Answer:

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Construction of a Binary Tree from an expression involves the following steps:

1. Create an empty stack.
2. Traverse the given expression string from left to right.
3. If the current character in the expression is an operand, create a new node with that operand as its value and push it onto the stack.
4. If the current character in the expression is an operator, pop two nodes from the stack and make them the left and right children of a new node with the operator as its value. Push this new node onto the stack.
5. Repeat steps 3 and 4 until the entire expression has been processed.
6. The final element on the stack will be the root of the constructed binary tree.

Parent-child relationship in a tree refers to the connection between a node and its immediate ancestors and descendants. Each node (except for the root node) has exactly one parent node and zero or more child nodes.

Q-4: Write the algorithm to construct a Huffman tree.

Answer:

The algorithm to construct a Huffman tree involves the following steps:

- ✚ Create a leaf node for each symbol and add it to a priority queue, ordered by ascending frequency.

- ✚ While there are more than one node in the queue: 3. Remove the two nodes of lowest frequency from the queue 4. Create a new internal node with these two nodes as children and with a frequency equal to the sum of their frequencies. 5. Add the new internal node to the queue
- ✚ The remaining node is the root node of the Huffman tree.

Q-5: Define binary tree, complete binary tree and binary search tree

Answer:

A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child. The left child is always less than the parent node, while the right child is always greater than or equal to the parent node.

A complete binary tree is a type of binary tree in which all levels of the tree are completely filled, except possibly for the last level, which is filled from left to right. In other words, every level must be fully populated before adding nodes to the next level.

A binary search tree (BST) is a type of binary tree in which every node's left child has a value less than its own value, and every node's right child has a value greater than or equal to its own value. This property makes searching for a specific node very efficient, as you can eliminate half of the remaining nodes with each comparison. BSTs are commonly used in computer science for searching and sorting operations.

GRAPH

Q- :Show all the steps to evaluate the following postfix expression using postfix expression evaluation algorithm 123+*321-+*

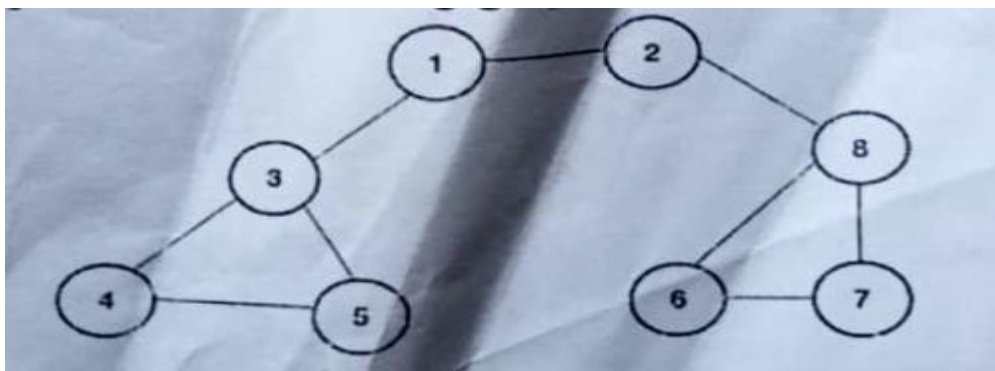
Answer:

ABC+*CBA-+* assum A=1, B=2, C=3

⇒ 123+*321-+*

Symbol	STACK	Operation
1	1	
2	1, 2	
3	1, 2, 3	
+	1, 5	[2+3]
*	5	[1*5]
3	5, 3	
2	5, 3, 2	
1	5, 3, 2, 1	
-	5, 3, 1	[2+1]
+	5, 4	[3+1]
*	20	[5*4]

Q- :Show the DFS traverse output steps of the following graph.(starting node 1)



Answer:

Q-2: Draw the graph from the following Adjacency list

Node	Adjacency list
A	F, C, B
B	G, C
C	F
D	C
E	D, C, J
F	D
G	C, E
J	D, K
K	E, G

Applying the BFS from A to J.

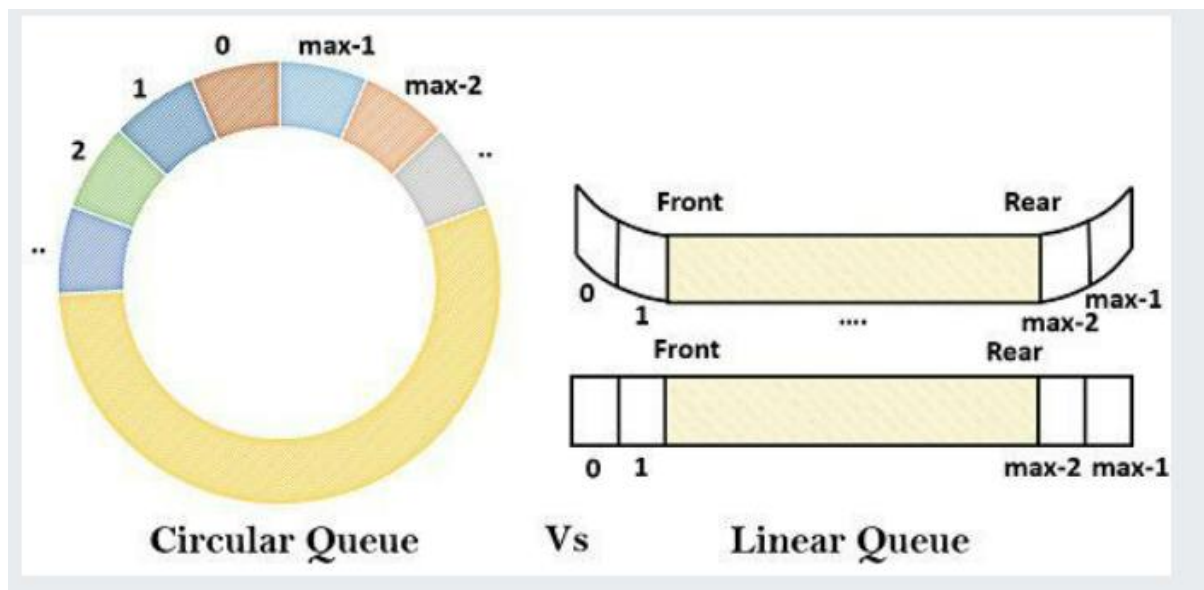
Answer:

* If heights are given the nodes are as -

Binary tree	Max node	Min node
Binary tree	$2^{h+1} - 1$	$h+1$
Full binary	$2^{h+1} - 1$	2^{h+1}
Complete binary	$2^{h+1} - 1$	2^h

* If nodes are given the heights are -

Types	Max height	Min height
Binary tree	$\lceil \log_2(n+1) \rceil - 1$	$n-1$
Full binary	$\lceil \log_2(n+1) \rceil - 1$	$\frac{n+1}{2}$
Complete binary	$\lceil \log_2(n+1) \rceil - 1$	$\log_2 n$



Follow sir note.....

MD RAIHAN ALI
CSE_02
Faculty of Engineering & Technology
University of Dhaka (NITER)