COMBINED PREVIOUS YEAR QUESTIONS OF DATA STRUCTURE CREATED BY ABHINAV KAUSHIK

SECTION A

Q.1 How a two dimensional array is represented in memory ?

Ans. A 2D array is represented in memory as a contiguous block of storage where the elements are arranged in rows and columns. The specific representation can vary based on the programming language and memory layout conventions. I'll provide a common representation, particularly in languages like C and C++, using the concept of row-major order.

Row-Major Order Representation (C/C++ Convention):

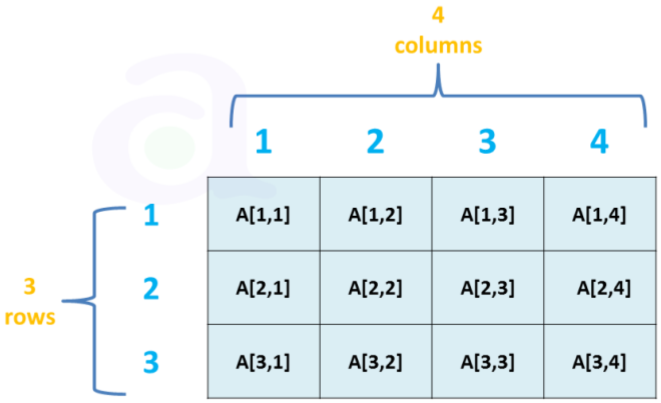
In row-major order, elements are stored row by row in memory. For a 2D array arr[m][n]:

(i). The entire array is a contiguous block of memory, and each element occupies a fixed amount of space.

(ii). The elements of the first row are stored first, followed by the elements of the second row, and so on.

(iii).The memory address of an element at position [i][j] can be calculated using the formula :

address=base_address+(i×number_of_columns+j)×size_of_each_element
address=base_address+(i×number_of_columns+j)×size_of_each_element



	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Fig: Two-dimension 3x4 array

in this example, the memory addresses increase sequentially from the base address, reflecting the row-major order.

Q.2 Discuss the significance of priority queues.

Ans. **A priority queue is a data structure that's important because it allows for efficient management of tasks and data by processing elements based on their priority:**

- **Efficient operations: Priority queues allow for efficient insertion and removal of elements.**
- **Fast access: Priority queues allow for quick access to the highest or lowest priority element.**
- **Flexibility: Priority queues can be implemented using arrays, linked lists, or heaps.**
- **Real-time applications: Priority queues are useful in many real-time applications.**
- **Algorithm efficiency: Priority queues are essential for implementing algorithms like Dijkstra's and A* search, and they increase the efficiency of other algorithms like Heap Sort.**

Q.3 Enlist various primitive operations of stack.

Ans. The following are the basic operations served by stacks.

- **`push`:** Adds an element to the top of the stack.
- **`pop`:** Removes the topmost element from the stack.
- **`isEmpty`:** Checks whether the stack is empty.
- **`isFull`:** Checks whether the stack is full.
- **`top`:** Displays the topmost element of the stack.

Q.4 What do you understand by Indexing in Binary Search Trees ?

Ans. An indexed binary search tree is a search tree that allows us to select the kth element. Imagine we had an ordered array with n elements. We could find the kth element in constant time. With the regular binary search tree we lose this ability but through a little ingenuity we can regain this ability through an index binary search tree. The only different is that it takes log n to find the kth element.

>> The following characteristics determine the nodes' placement in a binary search tree:

- Nodes with keys lower than a given node's key are always found in the left subtree of that node.
- Any node with a key bigger than that node's key will always be found in the right subtree.
- A certain node will also have a left and a right subtree that are both binary search trees.

Q.5 Describe how hashing enhances the system performance.

Ans. Hashing improves system performance in several ways:

- **Faster data retrieval**

  Hashing allows data to be quickly retrieved from cache memory, which reduces the need to access slower storage systems.

- **More efficient space utilization**

Hash tables don't need to store extra information about the order of data, which helps to save file space.

- **Reduced overhead**

  Hashing can reduce the overhead associated with other data structures, such as binary search trees.

- **Improved data integrity**

  The hash code produced by the hash function serves as a unique identifier in the data set, which helps to maintain data integrity.

- **Faster search and retrieval**

  Hash tables allow constant-time search and retrieval, regardless of size.

- **Improved cybersecurity**

  Hashing can be used to create digital signatures, which verify the origin, authenticity, and integrity of a message, document, or transaction.

Hashing is a technique that maps data of any size to a fixed-length value, called a hash or a digest.

Q.6 Differentiate between linear search and binary search technique.

Parameters	Linear Search	Binary Search
Definition	Linear Search sequentially checks each element in the list until it finds a match or exhausts the list.	Binary Search continuously divides the sorted list, comparing the middle element with the target value.
Time Complexity	The time complexity is O(n), where n is the number of elements in the list.	the time complexity is O(log n), making it faster for larger datasets.
Efficiency	Less efficient, especially for large datasets.	More efficient, especially for large datasets.

Data Requirement	Does not require the list to be sorted.	Requires the list to be sorted.
Implementation	Easier to implement.	Requires a more complex implementation.
Search Space	Examines each element sequentially.	Eliminates half of the search space with each comparison.
Use case	Suitable for small and unsorted datasets.	Ideal for large and sorted datasets.

Q.7 Give the use of header node in linked list.

Ans. A header node is used in a linked list to eliminate special cases that can occur when operating near the boundaries of the list:

- When the list is empty
- When the current position is at one end of the list
- Deleting the last node

(i) A header node is the first node in a linked list, and its value is ignored. It's not considered to be an actual element of the list.

(ii) Adding a header node can save space due to smaller code size, because statements to handle the special cases are omitted.

In a doubly linked list, a trailer node is also added at the end of the list to avoid special cases. These "dummy" nodes are known as sentinels (or guards), and they do not store elements of the primary sequence.

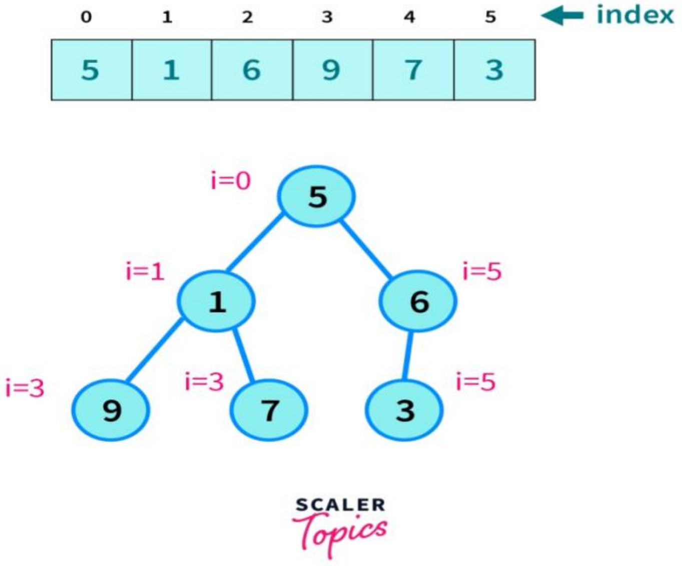
Q.8 Briefly explain complete binary tree with example.

Ans. A complete binary tree is a type of binary tree in which all the levels are completely filled(i.e, have two children), except the possibly the deepest level. The deepest level can have nodes that can have either one left child or can be completely full.

A complete binary tree can be a full binary tree (when the deepest levels are completely full).

Complete Binary Tree is a binary Tree in which at every level *l* except the last level has *2l* nodes and the nodes at last nodes are line up from left side.

It can be represented using array. Given parent is at index i so its left child is at 2i+1 and its right child is given by 2i+2



Q.9 Differentiate between pre-order and post order tree traversal.

Parameters	Pre-order	Post-order
Visit Order	Root, Left Subtree, Right Subtree	Left Subtree, Right Subtree, Root
Sorted Order (BST)	No	Yes
Expression Tree	Prefix (Polish) Notation	Postfix (Reverse Polish) Notation

Evaluation Order	Operators come before operands	Operators come after operands
Memory Usage	No additional memory for tracking traversal, stack space for recursion/iteration	No additional memory for tracking traversal, stack space for recursion/iteration
Left-Right Symmetry	Not symmetric	Not symmetric
Performance	Better for copying/cloning trees, prefix notation	Suitable for deleting nodes, post-fix notation, expression evaluation
Tree Modification	Easy to copy/cloning, preserving original structure	Difficult for modification, requires reversing

Q.10. Define the following terms :

a. Merge Sort : Merge sort is one of the most efficient sorting algorithms. It is based on the divide-and-conquer strategy. Merge sort continuously cuts down a list into multiple sublists until each has only one item, then merges those sublists into a sorted list.

b. Sparse Array : A **sparse array** or **sparse matrix** is an array in which most of the elements are zero.
The sparse array is an array in which most of the elements have the same value(the default value is zero or null).Sparse matrices are those array that has the majority of their elements equal to zero.

A sparse array is an array in which elements do not have contiguous indexes starting at zero.

c. Two-way list: Two-way header linked list is a type of linked list which has a header node at the beginning of the linked list which does not store any data and simply points to the first node of the linked list. All the other nodes have three parts: **data**, **pointer to the previous node** and **pointer to the next node**. Unlike Singly Linked List where each node has one pointer, there are two pointers in each node of Two-Way Header Linked List. The previous pointer of the first node and next pointer of the last node points to NULL.

d. Selection Sort : Selection sort is a simple comparison-based sorting algorithm that divides the input list into a sorted part at the beginning and an unsorted part at the end. The algorithm repeatedly selects the smallest (or largest, depending on the order) element from the unsorted part and swaps it with the first unsorted element, gradually growing the sorted portion until the entire list is sorted.

e. Linear Search : A linear search is the simplest approach employed to search for an element in a data set. It examines each element until it finds a match, starting at the beginning of the data set, until the end. The search is finished and terminated once the target element is located.

f. Inverted Trees : Inverted trees are the data structures used to represent hierarchical file structures. In this case, the leaves are files and the other elements above the leaves are directories. A binary tree is a special type of inverted tree in which each element has only two branches below it.

g. Post order traversal : Postorder traversal is a depth-first search algorithm for a binary search tree that first traverses the left subtree, then the right subtree, and then the root. Its primary use is deleting the tree.

h. Binary Search Tree (BST) : A Binary Search Tree (BST) is a type of Binary Tree data structure, where the following properties must be true for any node "X" in the tree: The X node's left child and all of its descendants (children, children's children, and so on) have lower values than X's value.

i. DATA STRUCTURE : A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need.

J) Hashing Techinques : Hashing in the data structure is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.

(k) Priority Queue : A priority queue is a special type of queue in data structure where each element is associated with a priority. In this queue, elements with higher priority are dequeued before the elements with lower priority. If two elements carry the same priority, they are served as per their ordering in the queue.

(l) Stack : The stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out). Real-life examples of a stack are a deck of cards, piles of books, piles of money, and many more.

(m)Heap Sort : Heap sort is a sorting algorithm that organizes elements in an array to be sorted into a binary heap by repeatedly moving the largest element front he heap and inserting it into the array being sorted. Priority queues are implemented with a heap, a tree-like data structure also used in the heap sort algorithm.

- (n) Linked List : Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.

(o) Collision resolution : In this, the hash function is used to find the index of the array. The hash value is used to create an index for the key in the hash table. The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a collision happens. To handle this collision, we use collision resolution techniques.

SECTION - B

Q.1 Briefly describe the concept of Tridiagonal matrices with examples.

Ans.1 A tridiagonal matrix is a square matrix with non-zero elements only on the main diagonal, the subdiagonal, and the superdiagonal. Tridiagonal matrices are commonly used in computer graphics and scientific computing for accurate and efficient computations.

A tridiagonal matrix is a matrix that is both upper and lower **Hessenberg matrix.**[2] In particular, a tridiagonal matrix is a **direct sum** of *p* 1-by-1 and *q* 2-by-2 matrices such that *p* + *q*/2 = *n* — the dimension of the tridiagonal. Although a general tridiagonal matrix is not necessarily **symmetric** or **Hermitian,** many of those that arise when solving linear algebra problems have one of these properties.

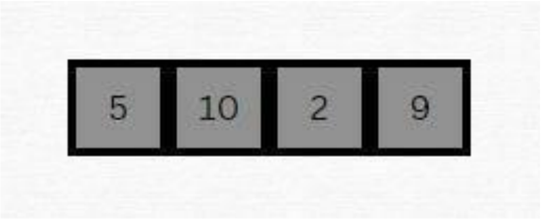
❖ Here are some examples of tridiagonal matrices:

- The determinant of a tridiagonal matrix is given by the continuant of its elements.
- The Lanczos algorithm can be used to orthogonally transform a symmetric matrix to tridiagonal form.
- A tridiagonal system of equations is a system of simultaneous algebraic equations with non-zero coefficients only on the main diagonal, the lower diagonal, and the upper diagonal.

Q.2 Write an Algorithm to perform Merge Sort.

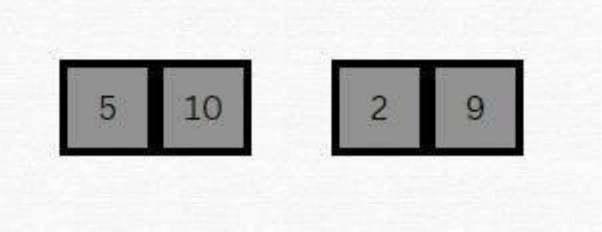
Ans.2 Initial Array

We start with an unsorted array of numbers. The purpose of Merge Sort will be to sort this array in ascending order.



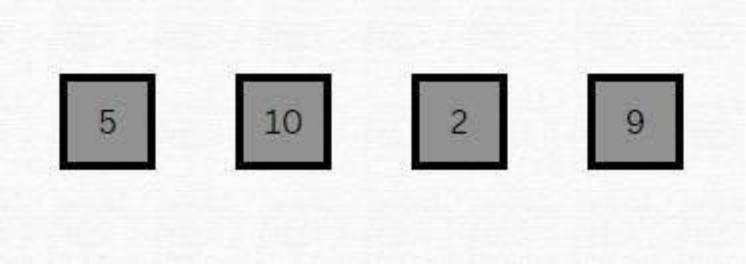
Step 1: Dividing the Array

Merge Sort begins with a divide-and-conquer approach. The array is divided into two halves, which is represented in the second image where the array [5, 10, 2, 9] is split into two subarrays: [5, 10] and [2, 9]. This division continues recursively until we cannot divide the arrays any further, typically when we reach arrays of single elements.



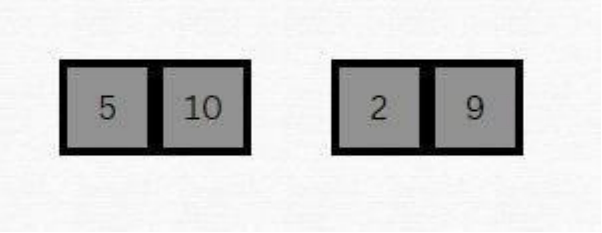
Step 2: Sorting Subarrays

Once we have our subarrays, each is sorted individually. In the case of single-element arrays, they are already sorted by definition. So, our subarrays [5, 10] and [2, 9] are considered sorted because they are composed of single elements after further division.



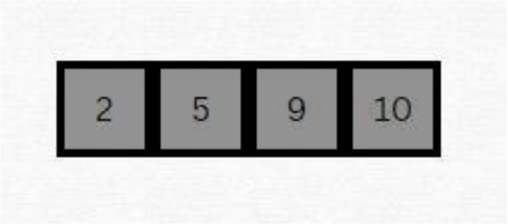
Step 3: Merging Subarrays

After sorting the subarrays, Merge Sort enters the conquer phase, which involves merging the sorted subarrays. This is where we combine our subarrays into larger, sorted arrays. We compare the elements at the beginning of each subarray and place the smaller one into the new array first.



Step 4: Final Merging

The last image represents the final merge where two sorted halves of the original array are combined to form the final sorted array. If there were more elements, the merge process would continue until all subarrays are merged into a single sorted array.



In each merging step, the algorithm follows the same principle: comparing the front elements of each subarray and picking the smaller one to place into the new, sorted subarray. This process is repeated until all elements are sorted and merged.

Q.3 What is Postfix expression form of any infix expression ? Write an algorithm to convert infix to postfix expression.

Ans.3 The postfix expression of an infix expression is written with the operator after its operands. For example, the postfix expression for the infix expression `A + B` is `A B +`.

To convert an infix expression to a postfix expression, you can:

1. Parenthesize the expression completely
2. Move each operator to the right of its right parentheses
3. Rewrite the expression in the new order

ALGORITHM :

1. Push “(“ onto STACK, and add “)” to the end of X.
2. Scan X from left to right and REPEAT Steps3 to 6 for each element of X UNTIL the STACK is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto STACK.

5. If an operator is encountered, then:
(a) Repeatedly pop from STACK and add to Y each operator (on the top of STACK) which has the same precedence as or higher precedence than operator.
(b) Add operator to STACK.
/* End of If structure */
6. If a right parenthesis is encountered, then:
(a) Repeatedly pop from STACK and add to Y each operator (on the top of STACK) until a left Parenthesis is encountered.
b) Remove the left parenthesis. (Do not add the left parenthesis to Y).
/* End of If structure */
7. END.

Q.4 Write a program in C language to search a given element in Linked list.

```
/**
 * C program to search an element by key in Singly Linked List.
 */
#include <stdio.h>
#include <stdlib.h>

/* Node structure */
struct node
{
   int data;         // Data
   struct node *next; // Address
} * head;


/* Function declaration */
void createList(int n);
void displayList();
int  search(int key);
int main()
{
   int n, keyToSearch, index;

   // Input node count to create
   printf("Enter number of node to create: ");
   scanf("%d", &n);

   createList(n);

   // Display list
   printf("\nData in list: \n");
   displayList();

   // Input element to search from user.
   printf("\nEnter element to search: ");
   scanf("%d", &keyToSearch);

   // Call function to search first element by key
   index = search(keyToSearch);

   // Element found in the list
   if (index >= 0)
      printf("%d found in the list at position %d\n", keyToSearch, index + 1);
   else
      printf("%d not found in the list.\n", keyToSearch);



   return 0;
}

/**
```

```
 * Create a list of n nodes
 */
void createList(int n)
{
    struct node *newNode, *temp;
    int data, i;

    head = malloc(sizeof(struct node));

    /*
     * Unable to allocate memory, hence exit from app.
     */
    if (head == NULL)
    {
        printf("Unable to allocate memory. Exiting from app.");
        exit(0);
    }


    /* Input head node data from user */
    printf("Enter data of node 1: ");
    scanf("%d", &data);

    head->data = data; // Link data field with data
    head->next = NULL; // Link address field to NULL

    temp = head;

    /*
     * Create n nodes and add to the list
     */
    for (i = 2; i <= n; i++)
    {
        newNode = malloc(sizeof(struct node));

        /* If memory is not allocated for newNode */
        if (newNode == NULL)
        {
            printf("Unable to allocate memory. Exiting from app.");
            exit(0);
        }

        printf("Enter data of node %d: ", i);
        scanf("%d", &data);

        newNode->data = data; // Link data field of newNode
        newNode->next = NULL; // The newNode should point to nothing

        temp->next = newNode; // Link previous node i.e. temp to the newNode
        temp = temp->next;
    }

}


/**
 * Display entire list
 */
void displayList()
{
    struct node *temp;

    /*
     * If the list is empty i.e. head = NULL,
     * dont perform any action and return.
     */
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }

    temp = head;
    while (temp != NULL)
    {
        printf("%d, ", temp->data);
        temp = temp->next;  // Move to next node
    }
    printf("\n");
}


/**
 * Search an element with given key in linked list.
 * It return a positive integer specifying index of the element
 * on success, otherwise returns -1.
 */
int search(int key)
{
    int index;
    struct node *curNode;

    index = 0;
    curNode = head;

    // Iterate till last element until key is not found.
    while (curNode != NULL && curNode->data != key)
    {
        index++;
        curNode = curNode->next;
    }


    return (curNode != NULL) ? index : -1;
}
```

Q.5 What is B-tree ? how do you construct the B-tree ? explain with example.

Ans.5 A B-tree is a self-balancing data structure that stores data in a sorted manner and allows for efficient operations like searching, inserting, and deleting. B-trees are a generalization of binary search trees, allowing nodes to have more than two children. Here are some properties of B-trees:

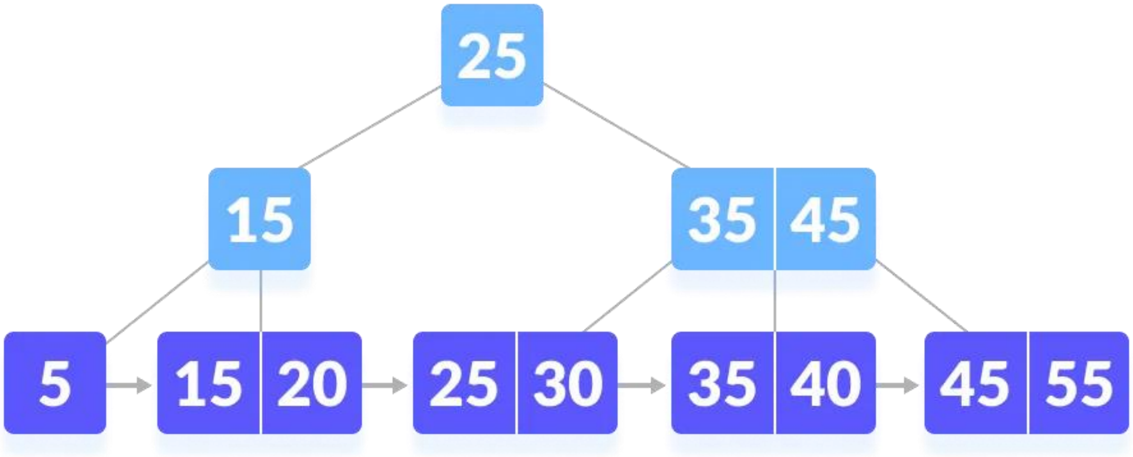
- **Sorted data**: Each node contains multiple keys that are sorted.
- **Pointers to subtrees**: The children of each node are pointers to subtrees that also follow the B-tree properties.
- **Balanced tree**: The structure of B-trees ensures that the tree remains balanced, with all leaf nodes at the same level.

The following steps are followed to search for data in a B+ Tree of order `m`. Let the data to be searched be `k`.

1. Start from the root node. Compare k with the keys at the root node `[k1, k2, k3,......km - 1`.
2. If `k < k1`, go to the left child of the root node.
3. Else if `k == k1`, compare `k2`. If `k < k2`, k lies between `k1` and `k2`. So, search in the left child of `k2`.
4. If `k > k2`, go for `k3, k4,...km-1` as in steps 2 and 3.
5. Repeat the above steps until a leaf node is reached.
6. If `k` exists in the leaf node, return true else return false.

Searching Example on a B+ Tree

Let us search `k = 45` on the following B+ tree.



Q.6 Discuss the Following with example:

Ans 6 (a) Lower triangular matrix

(a) The lower triangular matrix is a type of triangular where all the elements above the principal diagonal are zero.

In simple terms, a matrix L = {l_{ij}} is known as a lower triangular matrix if and only if l_{ij} = 0, for all $i < j$.

$$\begin{pmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{pmatrix}$$

Examples of Lower Triangular Matrix

A simple 2x2 Lower Triangular Matrix

$$\begin{pmatrix} 1 & 0 \\ -3 & 4 \end{pmatrix}$$

(b) Upper triangular matrix

(b) The upper triangular matrix is a type of triangular matrix where all the elements below the principal diagonal are zero.

In simple terms, a matrix $U = \{u_{ij}\}_{nxn}$ is known as an upper triangular matrix if and only if $u_{ij} = 0$, for all i>j.
i.e.,

$$\begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$$

Example of Upper Triangular Matrix

A simple 2x2 Upper Triangular Matrix

$$\begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix}$$

(c) Tridiagonal matrix

(c) A tridiagonal matrix is a square matrix with nonzero elements only on the main diagonal, the first subdiagonal, and the first superdiagonal. Here's an example of a tridiagonal matrix:

An important example is the matrix T_n that arises in discretizating the Poisson partial differential equation by a standard five-point operator, illustrated for $n = 5$ by

$$T_5 = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & 4 & -1 & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix}.$$

Q.7 Write a recursive function to count the number of leaves in a binary tree

Ans 7

The below algorithm does it recursively.

```
If (current node is a leaf node)

   leaf_count += 1;

else

   leaf_cout += leaf_count of left sub tree + leaf_count of right sub tree
```

Q.8 Write a function to reverse a stack using push and pop operation.

Ans 8

Steps to Reverse a Stack

1. **Remove All Elements**: Use a recursive function to remove all elements from the stack one by one until it is empty. Each element is popped and stored in the call stack.
2. **Insert Elements at the Bottom**: Once the stack is empty, start inserting the elements back into the stack from the call stack (which holds the elements in reverse order).

Implementation

Here's a sample implementation in Python:

```
def reverse_stack(stack):
    # Base case: If the stack is empty, return
    if not stack:
        return

    # Step 1: Remove the top element
    top_element = stack.pop()

```

```
  # Step 2: Reverse the remaining stack
        reverse_stack(stack)

        # Step 3: Insert the popped element at the bottom of the reversed stack
        insert_at_bottom(stack, top_element)

    def insert_at_bottom(stack, item):
        # If stack is empty, append the item
        if not stack:
            stack.append(item)
            return

        # Remove all items from the stack
        top = stack.pop()

        # Insert the item at the bottom
        insert_at_bottom(stack, item)

        # Push the top item back onto the stack
        stack.append(top)

    # Example usage
    stack = [1, 2, 3, 4, 5]
    print("Original stack:", stack)
    reverse_stack(stack)
    print("Reversed stack:", stack)
```

Q.9 Write an algorithm to delete last node from a linked list.

Ans.9 Algorithm to delete last node in linked list

- Base Case 1 – If the head is NULL, return NULL
- Base Case 2 – if head -> next is NULL, delete the head and return NULL.
- This means that there was only a single node in the linked list.
- Create a new node, say SecondLast and make it point to the head of the list.

- Now, traverse through the list by incrementing SecondLast, till the second last node of the linked list is reached.
- When we reach the second last node of the linked list, simply delete the next of the second last node i.e delete the last node.
- delete(SecondLast -> next).
- Make the second last node point to NULL. This will make our second last node new tail of the list.
- Return head.

Q.10 Describe the types of sparse matrix. How can we store a 2D sparse matrix in a corresponding single dimensional array ?

Ans 10. There are 7 types of sparse matrices that you can use. These are:

- csc_matrix: Compressed Sparse Column format
- csr_matrix: Compressed Sparse Row format
- bsr_matrix: Block Sparse Row format
- lil_matrix: List of Lists format
- dok_matrix: Dictionary of Keys format
- coo_matrix: Coordinate format (aka IJV, triplet format)
- dia_matrix: Diagonal format

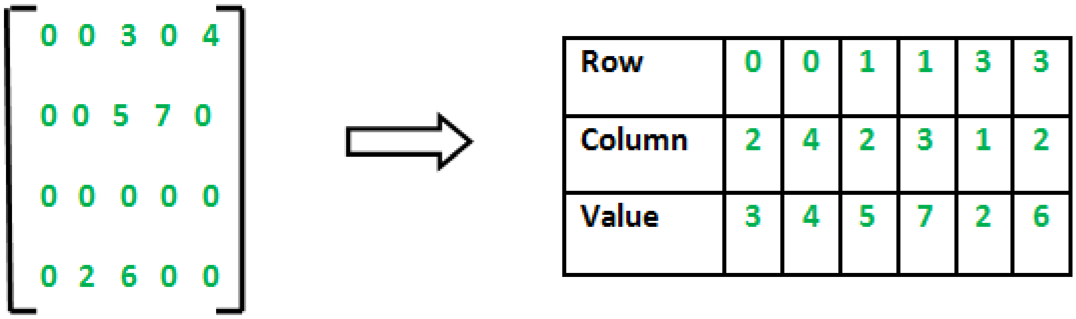
Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



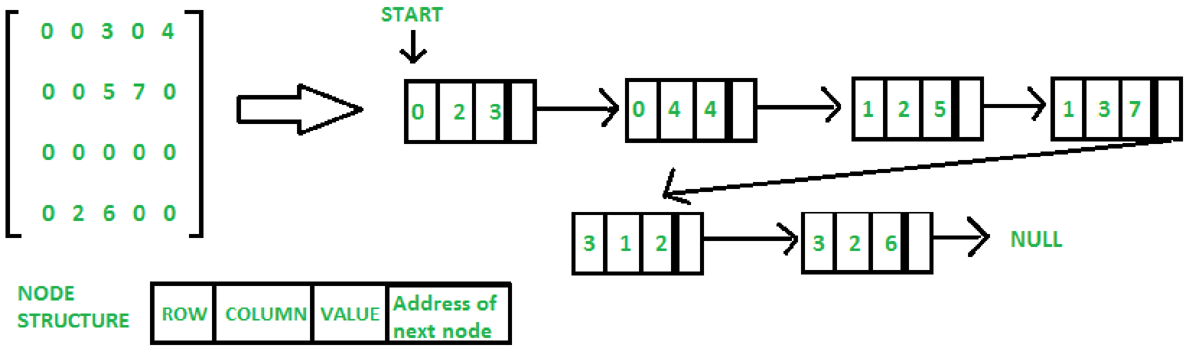
$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

NODE STRUCTURE

ROW	COLUMN	VALUE	Address of next node

❖ find the formula for address calculation.

- In a single dimensional array the address of an element of an array say A[i] is calculated using the following formula Address of A[i]=B+W∗(i–LB) where B is the base address of the array, W is the size of each element in bytes, i is the subscript of an element whose address is to be found and LB is the Lower limit / .

SECTION C

Q.1 Define overflow and underflow conditions of Linked List. Write an algorithm to insert a node into Linked List at a specific location.

Ans.1 in the context of linked list operations, underflow and overflow refer to situations where you try to remove an element from an empty list (underflow) or add an element to a list that has reached its maximum capacity (overflow). However, it's important to note that in a typical linked list implementation, there is no maximum size (other than the limits of your computer's memory), so overflow is not usually a concern.

Underflow, on the other hand, is a common issue that needs to be handled. This occurs when you try to remove an element from an empty list. To prevent underflow, you should always check if the list is empty before trying to remove an element. If the list is empty, you can then return an error message or handle the situation in a way that makes sense for your specific application.

:-- algorithm to insert a node into Linked List at a specific location

The algorithm to insert a node at a specific position in a linked list is given below.

- If the position where we are asked to insert **pos** is smaller than 1 or greater than the **size of the list**, it is invalid, and we will return.
- Else, we will make a variable **curr** and make it point to the **head** of the list.
- Now we will run a for loop using **curr** to reach to the node at (**pos-1**)th position:
- The for loop will be: **for(int i=1;inext;}**
- After the termination of the above loop, **curr** will be standing at the (**position – 1**)th node.

- As explained above, we will simply make **newnode → next = curr → next** and **curr → next = newnode.**
- If the **pos** was equal to 1, we will make the **head** point to **newnode** as **newnode** will become the first node of the list.

Q.2 Define Stack. Write an algorithms for PUSH and POP operations. Also implement stack by using arrays.

Ans.2 A **Stack** is a linear data structure that follows a particular order in which the operations are performed. The order may be **LIFO(Last In First Out)** or **FILO(First In Last Out)**. **LIFO** implies that the element that is inserted last, comes out first and **FILO** implies that the element that is inserted first, comes out last.

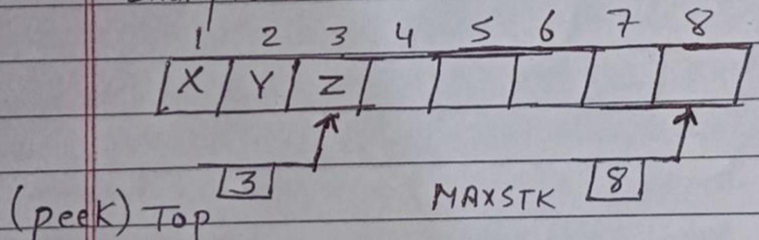
----->> algorithms for PUSH and POP operations :--------

SEE BELOW

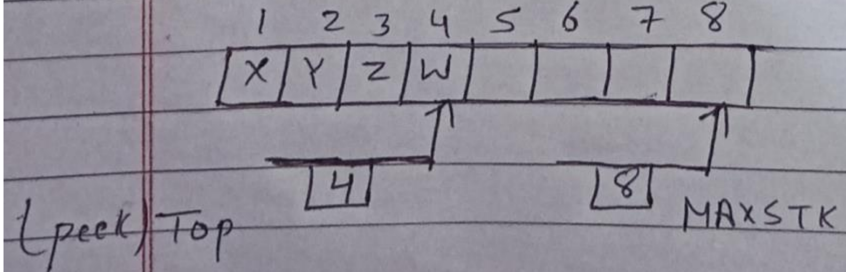
→ Algorithm (PUSH)

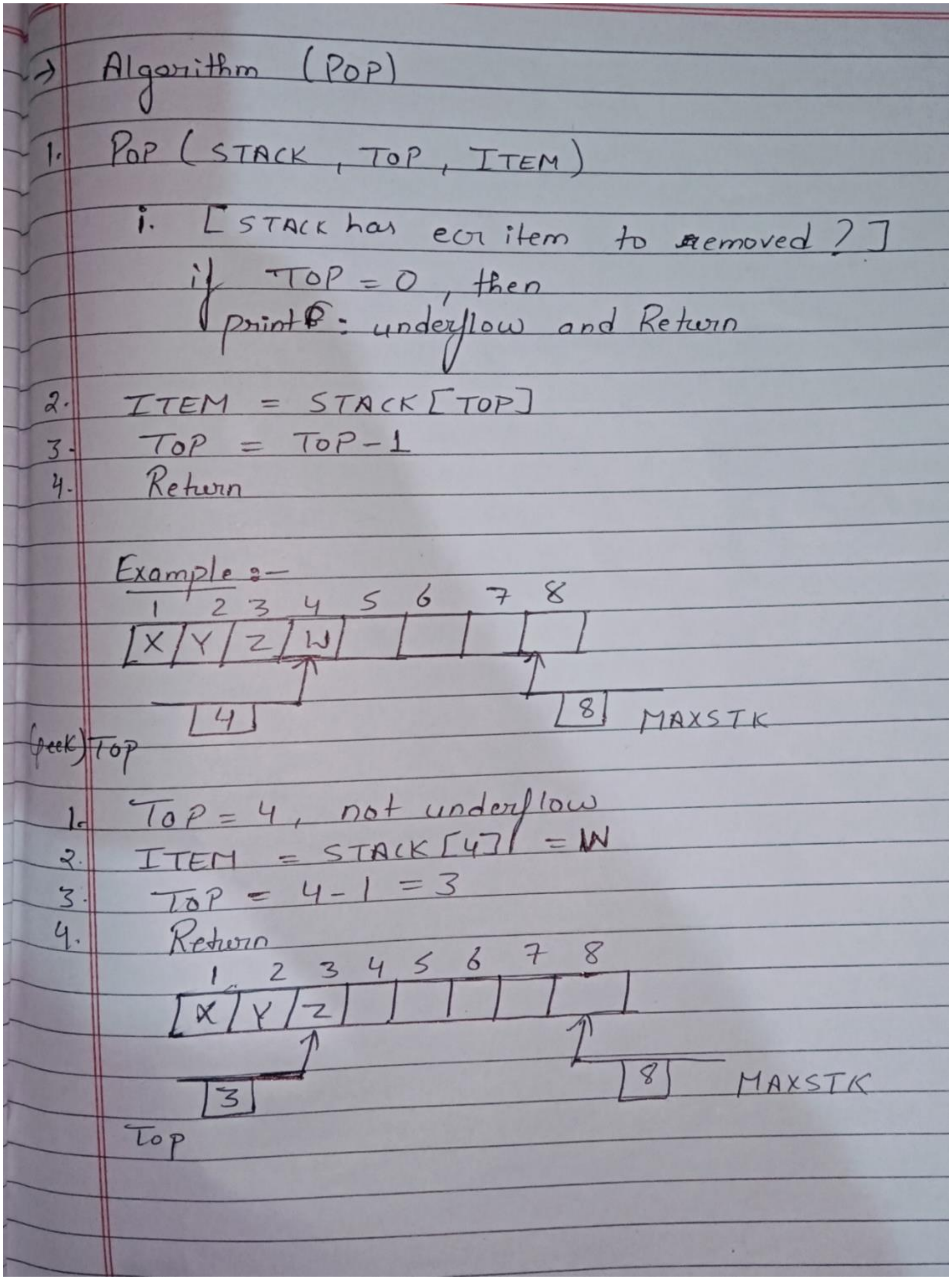
```
1.  PUSH ( STACK, TOP, MAXSTK, ITEM)
        [stack already filled ?]
        if TOP = MAXSTK, then
        print : overflow and return

2.  Set TOP = TOP + 1
3.  STACK [TOP]: = ITEM
4.  Return
```

→ Example :-



```
    PUSH (STACK, W)
1.  TOP = 3 , not overflow
2.  TOP = 3+1 = 4
3.  STACK [TOP] = STACK [4] = W
4.  Return
```





→ Algorithm (PoP)

```
1. PoP (STACK, TOP, ITEM)
    i. [STACK has an item to removed ?]
       if TOP = 0, then
       print: underflow and Return
2. ITEM = STACK[TOP]
3. TOP = TOP-1
4. Return
```

Example :-

1	2	3	4	5	6	7	8
X	Y	Z	W				

[4] (peek) Top

[8] MAXSTK

```
1. TOP = 4, not underflow
2. ITEM = STACK[4] = W
3. TOP = 4-1 = 3
4. Return
```

1	2	3	4	5	6	7	8
X	Y	Z					

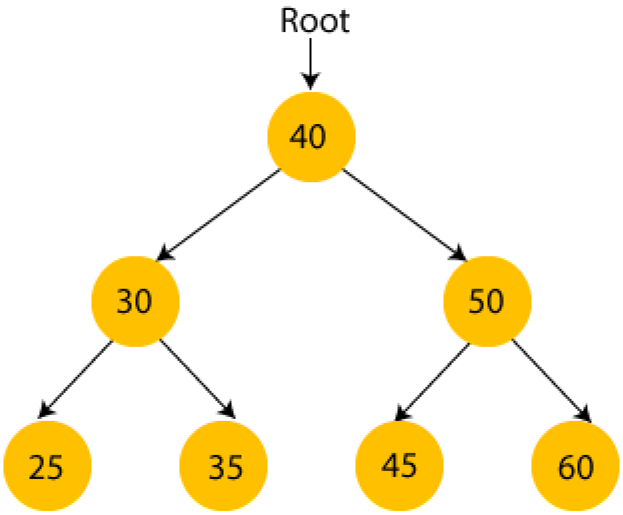
[3] Top

[8] MAXSTK

Q.3 What is Binary Search Tree?

Ans.3 A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Q.4 What do you mean by TBT ? Give the recursive algorithm for various types of binary tree traversal with suitable example.

Ans.4 Binary tree traversal is the process of visiting each node in a binary tree and processing its data in a specific order. The order in which the nodes are visited determines the type of traversal. Some common types of binary tree traversal include:

- **Preorder traversal**: Visits a node before its children
- **Postorder traversal**: Visits a node after its children and their subtrees
- **Inorder traversal**: Visits the left child, then the node, and finally the right child

Tree traversal is also known as tree search or walking the tree. It's a form of graph traversal, and is different from linear data structures like arrays, bitmaps, and matrices, where traversal is done in a linear order.

There are several algorithms for traversing a binary tree, including:

- **Inorder traversal**

  Visit the left subtree, then the root, and finally the right subtree. This order is useful for retrieving elements in sorted order in a binary search tree.

- **Preorder traversal**

  Visit the root, then the left subtree, and finally the right subtree.

- **Postorder traversal**

  Visit the left subtree, then the right subtree, and finally the root. This method is useful for operations that require processing children before their parents, such as tree deletions.

- **Level order traversal**

  Visit nodes level by level from left to right. This traversal uses a queue data structure to keep track of nodes.

To figure out the order in which a binary tree's nodes will be visited, you can use the "Tick Trick". This involves drawing an arrow around the nodes of the binary tree diagram, and drawing a line or tick mark on one of the sides or the bottom of each node.

Q.5 How a stack is represented in an array? Describe the various application stacks. Explain prefix, infix, post-fix expression with the help of examples.

Ans.5 An array is used to store an ordered list of elements. Using an array for representation of stack is one of the easy techniques to manage the data. But there is a major difference between an array and a stack.

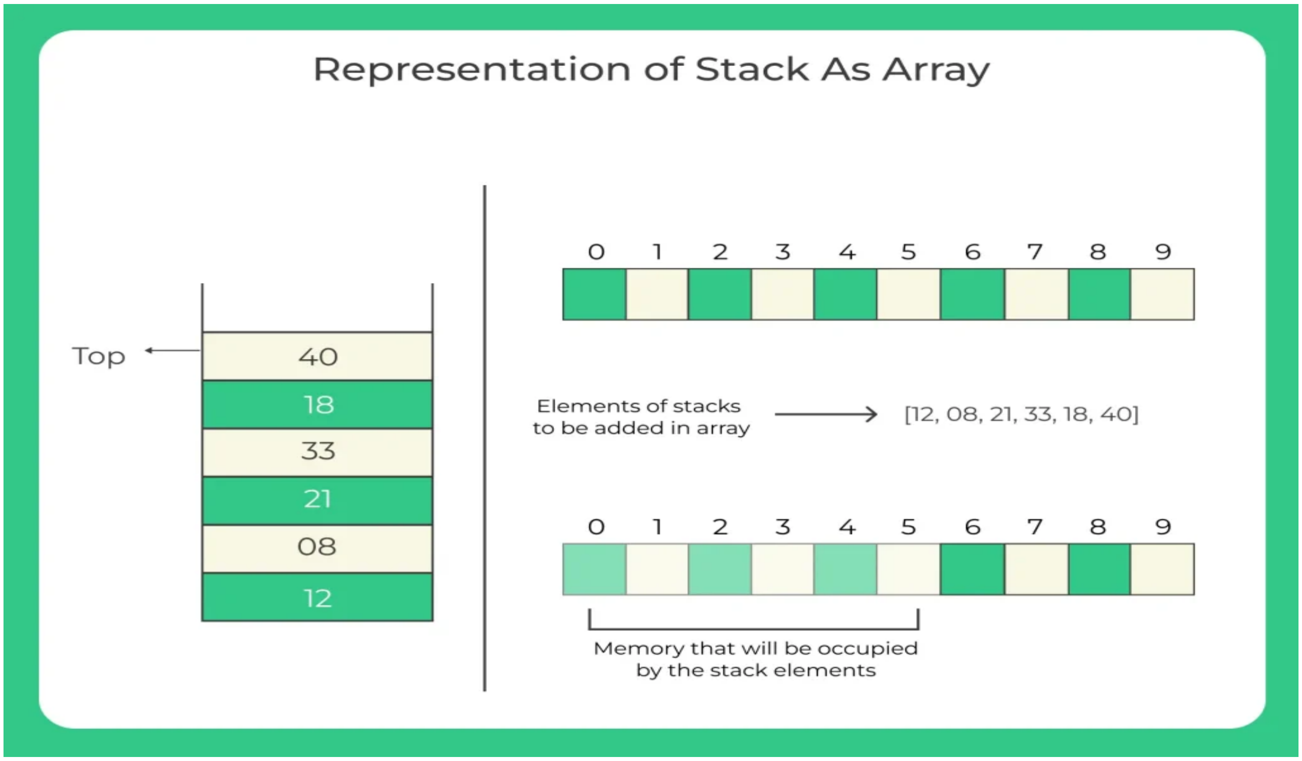
- Size of an array is fixed.

- While, in a stack, there is no fixed size since the size of stack changed with the number of elements inserted or deleted to and from it.

Despite the difference, an array can be used to represent a stack by taking an array of maximum size; big enough to manage a stack.

For Example:

We are given a stack of elements: **12 , 08 , 21 , 33 , 18 , 40.**


Representation of Stack As Array
Top
40
18
33
21
08
12
0 1 2 3 4 5 6 7 8 9
Elements of stacks to be added in array
[12, 08, 21, 33, 18, 40]
0 1 2 3 4 5 6 7 8 9
Memory that will be occupied by the stack elements

- **Application Of Stack**

1. Reverse of string

2. Checking validity of an expression,containing nested, parenthesis.

3. Function Call

4. Conversion of infix expression to postfix

5. Evaluation of Postfix form

For Example : # 3+5*(7-4)^2

Scan	Operator	Operation
3	(	3
+	(+	3
5	(+	35
*	(+*	35
(	(+*(	35
7	(+*(	357
-	(+*(-	357
4	(+*(-	3574
)	(+*	3574-
^	(+*^	3574-
2	(+*^	3574-2^*+

❖ There are Some Notations/expressions in Stack :

1. Infix Notation

Infix is the day to day notation that we use of format A + B type. The general form can be classified as ***(a op b)*** where ***a*** and ***b*** are operands(variables) and ***op*** is Operator.

- Example 1 : A + B
- Example 2 : A * B + C / D

2. Postfix Notation

Postfix is notation that compiler uses/converts to while reading left to right and is of format ***AB+*** type. The general form can be classified as ***(ab op)*** where ***a*** and ***b*** are operands(variables) and ***op*** is Operator.

- Example 1 : AB+
- Example 2 : AB*CD/+

3. Prefix Notation

Prefix is notation that compiler uses/converts to while reading right to left (some compilers can also read prefix left to right) and is of format +***AB*** type. The general form can be classified as ***(op ab)*** where ***a*** and ***b*** are operands(variables) and ***op*** is Operator.

- Example 1 : +AB
- Example 2 : +*AB/CD

Q.6 Write an algorithm to delete the kth node from a two way linked list. Explain the algorithm by taking an example.

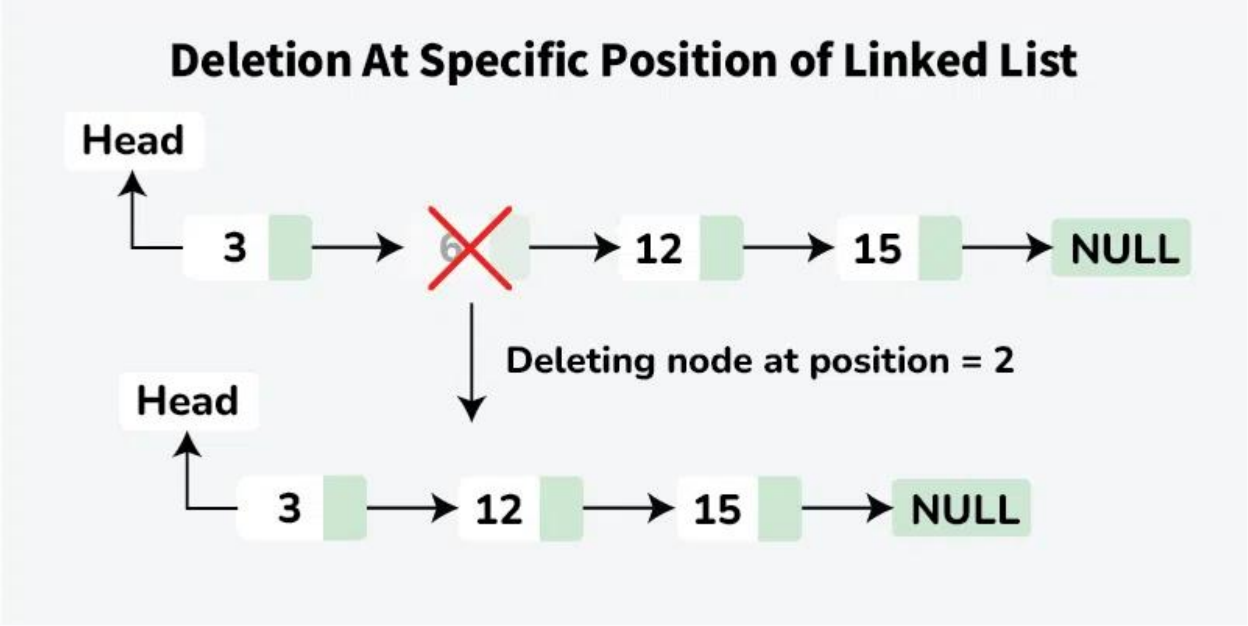
To delete the kth node from a doubly linked list, you can follow these steps:

1. Traverse the list to the kth node while keeping track of the count.
2. Use a temporary pointer to point to the kth node.
3. Set the previous node to the one that the temporary pointer's back pointer is pointing to.
4. Set the next pointer of the previous node to the front node.
5. Set the back pointer of the front node to the back node.
6. Set the temporary pointer's next and back pointers to null.
7. Delete the previous node (in C++).

Here are some things to consider:

- If the linked list is empty, return null.
- If k is equal to 1, delete the first node.
- If k is equal to the length of the linked list, delete the tail of the linked list.
- If the linked list has a single element, k can only take the value 1.

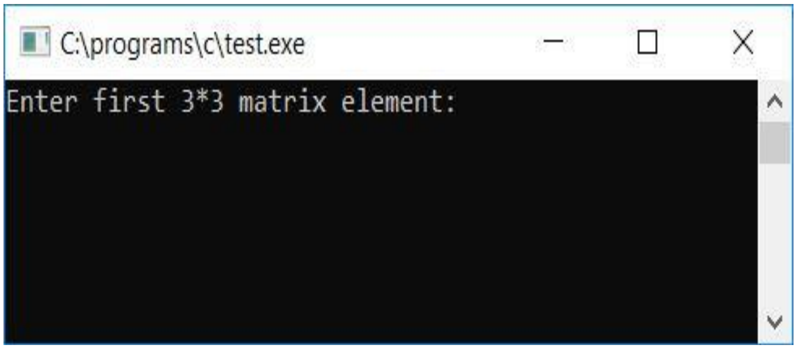
For Example :


Deletion At Specific Position of Linked List
Head
3
6
12
15
NULL
Deleting node at position = 2
Head
3
12
15
NULL

Q.7 Write a C program to multiply two 2-D matrix of 3*3 and store the result in another matrix.

Ans.7

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int mat1[3][3], mat2[3][3], mat3[3][3], sum=0, i,
j, k;
    printf("Enter first 3*3 matrix element: ");
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            scanf("%d", &mat1[i][j]);
    }
    printf("Enter second 3*3 matrix element: ");
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            scanf("%d", &mat2[i][j]);
    }
    printf("\nMultiplying two matrices...");
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
        {
            sum=0;
            for(k=0; k<3; k++)
                sum = sum + mat1[i][k] * mat2[k][j];
            mat3[i][j] = sum;
        }
    }
    printf("\nMultiplication result of the two given
Matrix is: \n");
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d\t", mat3[i][j]);
        printf("\n");
    }
    getch();
    return 0;
}
```



```
Enter first 3*3 matrix element:
```

Q.8 Describe hashing and various hashing techniques in detail.

Ans.8 Hashing is a technique that maps a key or string of characters to a shorter, fixed-length value, known as a hash. This process is used to store and retrieve data quickly, and is commonly used in data indexing, digital signatures, cryptography, and cybersecurity.

In C, hashing is used to create hash tables, which are data structures that store key and value pairs in a list. A hash function is used to calculate the index for a key, and the value is stored at that location in the hash table. The benefit of hash tables is that they have a very fast access time, typically with a constant O(1) access time.

Here are some things to consider when using hashing in C:

- **Hash function**

  A good hash function should be easy to compute, provide a uniform distribution across the hash table, and minimize collisions. Collisions occur when two different keys are mapped to the same index.

- **Collision resolution**

  Even with a good hash function, collisions are likely to occur. To maintain performance, you should use collision resolution techniques to manage collisions.

- **Table expansion**

  If the hash table gets too full, you may need to expand the table. One common approach is to double the capacity of the table when it's half full.

Q.9 Explain the properties of B-trees. Also create a B-tree of order 3 for following data.

Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec

Ans.9 **Properties of B+ Tree in Data Structure**

- Every node in a B+ Tree contains at most “m” children.

- Every node in a B+ Tree, except the root node and the leaf node contain at least “m/2” children.
- Nodes can have a maximum of (m-1) keys.
- Nodes can have a minimum of ⌈m/2⌉-1 keys.

❖ Its a B-tree of order 3 for following data.

```
     JAN
     /  \
 FEB   MAR
         \
         MAY
         \
         APR
         \
         JUN
         \
         SEP
         \
         AUG
         \
         OCT
         \
         NOV
         \
         DEC
```

so, you would have to start by creating a new node for the root of the tree and setting its value to the first element in the list, which is "JAN" in this case.

For each subsequent element in the list, compare the element to the root node. If the element is less than the root node, add it as the left child of the root node. If the element is greater than the root node, add it as the right child of the root node. Repeat this process for the newly added node, comparing it to its parent node and adding it as the left or right child as appropriate.

Q.10

(a). Explain the method of deletion in single linked list.

Ans (a) To delete a node from a single linked list, you can follow these steps:

1. Locate the node's previous node
2. Change the next of the previous node

Free the memory for the node being deleted

(b). What is the sorting ? explain Selection Sort.

Ans (b) Sorting in data structure is the process of arranging data elements in a specific order, either increasing or decreasing. This makes it easier to search for elements in the data structure, and to analyze and visualize the information.

Sorting is a fundamental part of data structures and algorithms (DSA) in computer science. There are many different sorting algorithms available, each with different time and space complexities.

- Selection Sort : **Selection Sort** is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
3. We keep doing this until we get all elements moved to correct position.

I hope you like my efforts
And I am 100 % Sure The paper will get stuck in this ---------ABHINAV KAUSHIK