# Insert element into an array

1. Start
2. Reset size of the array. ] set size = size + 1
3. Initialize counter variable. ] Set i = size - 1
4. Repeat Step 05 and 06 for i = size - 1 to i >= pos - 1
5. [Move ith element forward. ] set arr[i+1] = arr[i]
6. [Decrease counter. ] Set i = i - 1
7. [End of step 04 loop. ]
8. [Insert element. ] Set arr[pos-1] = x
9. Stop

# Delete element from an array

1. Start
2. [Initialize counter variable. ] Set i = pos - 1
3. Repeat Step 04 and 05 for i = pos - 1 to i < size
4. [Move ith element backward (left). ] set a[i] = a[i+1]
5. [Increase counter. ] Set i = i + 1
6. [End of step 03 loop. ]
7. [Reset size of the array. ] set size = size - 1
8. Stop

# Traversing an Array:

1. Start
2. [Initialize counter variable. ] Set i = LB.
3. Repeat for i = LB to UB.
4. Apply process to arr[i].
5. [End of loop. ]
6. Stop

# Linear Search

1. First, read the search element (Target element) in the array.
2. Set an integer i = 0 and repeat steps 3 to 4 till i reaches the end of the array.
3. Match the key with arr[i].
4. If the key matches, return the index. Otherwise, increment i by 1.

# Binary Search

1. Sort the array in ascending order.
2. Set the low index to the first element of the array and the high index to the last element.
3. Set the middle index to the average of the low and high indices.
4. If the element at the middle index is the target element, return the middle index.
5. If the target element is less than the element at the middle index, set the high index to the middle index – 1.
6. If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
7. Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

# Selection Sort

1. Begin
2. Set MIN to location 0
3. Search the minimum element in the list
4. Swap with value at location MIN
5. Increment MIN to point to next element
6. Repeat until list is sorted
7. End

# 

# Bubble Sort

.

1. Begin
2. begin BubbleSort(list)
3. for all elements of list
4. if list[i] > list[i+1]
5. swap(list[i], list[i+1])
6. end if
7. end for

8

9 return list

1. end BubbleSort
2. End

# 

# Matrix Addition

1. Start
2. Declare matrix mat1[row][col];  
    and matrix mat2[row][col];  
    and matrix sum[row][col]; row= no. of rows, col= no. of columns
3. Read row, col, mat1[][] and mat2[][]
4. Declare variable i=0, j=0
5. Repeat until i < row
6. Repeat until j < col
7. sum[i][j]=mat1[i][j] + mat2[i][j]
8. Set j=j+1
9. Set i=i+1
10. sum is the required matrix after addition
11. Stop

## 

## 

# Matrix Multiplication

1. matrixMultiplication(M1, M2):
2. Matrix dimension of M1 is (r1 x c1) and dimension of M2 is (r2 x c2)
3. Begin
4. if r1 is not equal to c2, then exit
5. otherwise define a new matrix M3 of dimension (r1 x c2)
6. for i in range 0 to r1, do
7. for j in range 0 to c2, do
8. for k in range 0 to r2, do
9. C[i, j] = C[i, j] + (A[i, k] \* A[k, j])
10. done
11. done
12. done
13. End

# Transpose of a Matrix

1. Declare and initialize a 2-D array p[a][b] of order axb.
2. Read the matrix p[a][b] from the user.
3. Declare another 2-dimensional array **t** to store the transpose of the matrix. This array will have the reversed dimensions as of the original matrix.
4. The next step is to loop through the original array and convert its rows to the columns of matrix t.
5. Declare 2 variables i and j.
6. Set both i,j=0
7. Repeat until i<b
8. Repeat until j<a
9. t[i][j] = p[j][i]
10. j=j+1
11. i=i+1
12. The last step is to display the elements of the transposed matrix t.

# 11. Sparse Matrix

1. declare array of structure Spar[][].
2. Accept conventional matrix A with minimal nonzer0 elements.
3. Initialize k=1;
4. For i=0 to row
5. For j=0 to column

if( A[i][j] !=0)

store i in spar[k].row

store j in spar[k].col

store A[i][j] in spar[k].val

1. Spar[0].row = row
2. Spar[0].col = column,
3. Spar[0].val = k

# 12. Stack Operations

1. Begin
2. begin procedure push: stack, data
3. if stack is full
4. return null
5. endif
6. top= top + 1
7. stack[top]=data
8. end procedure
9. begin procedure pop: stack
10. if stack is empty
11. return null
12. endif
13. data= stack[top]
14. top= top - 1
15. return data
16. end procedure
17. begin procedure peek
18. return stack[top]
19. end procedure
20. End

# 13. Infix to Postfix

1. Begin
2. Create a stack
3. for each character t in the input stream
4. if (t is an operand)
5. output t
6. else if (t is a right parentheses)
7. POP and output tokens until a left parentheses is popped(but do not output)
8. else
9. POP and output tokens until one of lower priority than t is encountered or a left parentheses is encountered or the stack is empty
10. PUSH t
11. POP and output tokens until the stack is empty.
12. End

# 

# 14. Postfix Evaluation

1. Begin
2. Create a stack to store operands (or values).
3. Scan the given expression and do following for every scanned element.
4. If the element is a number, push it into the stack
5. If the element is a operator, pop operands for the operator from stack.
6. Evaluate the operator and push the result back to the stack
7. When the expression is ended, the number in the stack is the final answer
8. End

# 15. Queue Operations

1. Begin
2. procedure enqueue(data)
3. if queue is full
4. return overflow
5. endif
6. rear=rear + 1
7. queue[rear]=data
8. return true
9. end procedure
10. procedure dequeue
11. if queue is empty
12. return underflow
13. end if
14. data = queue[front]
15. front=front + 1
16. return true
17. end procedure
18. begin procedure peek
19. return queue[front]
20. end procedure
21. End