**Assignment No. 1**

**Problem Statement:**

Represent sets using one-dimensional arrays and implement functions to perform

i. Union

ii. Intersection

iii. Difference

iv. Symmetric difference of two sets

**Objectives:**

To write a neat code by following coding standards by selecting appropriate data structures.

**Outcome:**

CO1: Students will be able to make use of coding standard for application development.

**THEORY:**

**SET OPERATIONS:**

Given the sets A & B, following are the various set operations.

**UNION:**Union of set A & B

                Elements of set A + uncommon elements of set B

                SET A = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B = {88, 55, 39, 41, 24, 51, 11}

                A U B  = {92, 11, 24, 76, 89, 44, 39, 3, 88, 55, 41, 51}

**INTERSECTION:**Intersection of set A & B

                Common elements between set A and set B

    SET A = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B = {88, 55, 39, 41, 24, 51, 11}

                A ∩ B  = {39, 24, 11}

**DIFFERENCE BETWEEN set A & B (A - B)**

Elements of set A – common elements between set A & B

    SET A = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B = {88, 55, 39, 41, 24, 51, 11}

                A - B = {92, 76, 89, 44, 3}

**DIFFERENCE BETWEEN set B & A (B - A)**

Elements of set B – common elements between set A & B

    SET A = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B = {88, 55, 39, 41, 24, 51, 11}

                B - A = {88, 55, 41, 51}

**SYMMETRIC DIFFERENCE BETWEEN set A & B (A – B)  U (B - A)**

    SET A = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B = {88, 55, 39, 41, 24, 51, 11}

                (A – B) U (B – A) = {92, 76, 89, 44, 3, 88, 55, 41, 51}

**ARRAY**1**.**DEFINITION

                        2. DECLARATION SYNTAX

                        3. INITIALIZATION

                        4. ACCEPT ELEMENTS

                        5. DISPLAY ELEMENTS

                        6. PASS 1D ARRAY TO FUNCTIONS

**ALGORITHM:**

**A)**    **UNION OF TWO SETS**

   Procedure union(A, B, n1, n2)

/\* Function to determine union of two sets A & B stored in arrays  \*/

1.      Read the two sets say A and B along with the number of elements in both say  n1 and n2

1. Initialize a variable say k = 0
2. Vary for i = 0 to n1                                         /\*copy elements of array A  to \*/

C[k] = A[i];                                         /\*array C                                    \*/

k++;

             end for

1. Vary for j = 0 to n2                                         /\*every element of array B is\*/

flag = 0;                                               /\*compared with the elements\*/

Vary for I = 0 to n1                            /\*of array A and if it exists in\*/

            Check if B[j] = = A[i] then     /\*array A then flag is set to 1\*/

                        flag = 1;

Break;

            End if

End for

                        Check if flag = = 0                              /\*number B[j] is uncommon\*/

                                    C[k] = B[j];                             /\*copy B[j] to array C\*/

                                    k++;

                        end if

            end for

1. Display the resultant union set ‘C’ .
2. stop.

End procedure union

**B)**    **INTERSECTION OF TWO SETS**

Procedure intersection (A, B, n1, n2)

/\* Function to determine intersection of two sets A & B stored in arrays  \*/

1. Read the two sets say A and B along with the number of elements in both say

      n1 and n2

1. Initialize a variable say k = 0
2. Vary for i = 0 to n1                             /\*copy common elements of array\*/

             Vary for j = 0 to n2                                  /\*A to array B \*/

            Check if B[j] = = A[i] then

                                    C[k] = B[j];

                                    k++;

                                    break;

                                    end if

                        end for

            end for

1. Display the resultant set ‘C’ .
2. stop.

End procedure intersection

**C)**    **DIFFERENCE OF TWO SETS**

Procedure difference (A, B, n1, n2)

/\* Function to determine difference of two sets A & B stored in arrays  \*/

1. Read the two sets say A and B along with the number of elements in both say

      n1 and n2

1. Initialize a variable say k = 0
2. Vary for i = 0 to n1                                         /\*every element of array B is\*/

flag = 0;                                               /\*compared with the elements\*/

Vary for j = 0 to n2                             /\*of array A and if it exists in\*/

            Check if B[j] = = A[i] then     /\*array A then flag is set to 1\*/

                        flag = 1;

Break;

            End if

End for

                        Check if flag = = 0                              /\*number A[i] is common\*/

                                    C[k] = A[i];                             /\*copy A[i] to array C\*/

                                    k++;

                        end if

            end for

1. Display the resultant set ‘C’ .
2. stop.

End procedure difference.

**D)**    **SYMMETRIC DIFFERENCE OF TWO SETS**

Procedure sym\_diff (A, B, n1, n2)

/\* Function to determine symmetric difference of two sets A & B stored in arrays \*/

1. Read the two sets say A and B along with the number of elements in both say

      n1 and n2

1. Initialize a variable say k = 0
2. Vary for i = 0 to n1                                         /\*every element of array B is\*/

flag = 0;                                               /\*compared with the elements\*/

Vary for j = 0 to n2                             /\*of array A and if it exists in\*/

            Check if B[j] = = A[i] then     /\*array A then flag is set to 1\*/

                        flag = 1;

Break;

            End if

End for

                        Check if flag = = 0                              /\*number A[i] is uncommon\*/

                                    C[k] = A[i];                             /\*copy A[i] to array C\*/

                                    k++;

                        end if

            end for

1. Vary for j = 0 to n2                                         /\*every element of array B is\*/

flag = 0;                                               /\*compared with the elements\*/

Vary for i = 0 to n1                             /\*of array A and if it exists in\*/

            Check if B[j] = = A[i] then     /\*array A then flag is set to 1\*/

                        flag = 1;

Break;

            End if

End for

                        Check if flag = = 0                              /\*number B[j] is uncommon\*/

                                    C[k] = B[j];                             /\*copy B[j] to array C\*/

                                    k++;

                        end if

            end for

1. Display the resultant set ‘C’.
2. stop.

End procedure sym\_diff.

**SAMPLE INPUT/ OUTPUT**

Enter the number of elements to be stored in set A: 08

Enter the elements of set A:

Enter A[0] : 92

Enter A[1] : 11

Enter A[2] : 24

Enter A[3] : 76

Enter A[4] : 89

Enter A[5] : 44

Enter A[6] : 39

Enter A[7] : 3

Enter the number of elements to be stored in set B: 07

Enter the elements of set B:

Enter B[0] : 88

Enter B[1] : 55

Enter B[2] : 39

Enter B[3] : 41

Enter B[4] : 24

Enter B[5] : 51

Enter B[6] : 11

**OUTPUT**

UNION

                SET A                     = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B                     = {88, 55, 39, 41, 24, 51, 11}

                A UNION B           = {92, 11, 24, 76, 89, 44, 39, 3, 88, 55, 41, 51}

INTERSECTION

    SET A                                 = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B                                 = {88, 55, 39, 41, 24, 51, 11}

                A INTERSECTION B       = {39, 24, 11}

DIFFERENCE BETWEEN A & B

    SET A                                 = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B                                 = {88, 55, 39, 41, 24, 51, 11}

                A - B                                   = {92, 76, 89, 44, 3}

DIFFERENCE BETWEEN B & A

    SET A                                 = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B                                 = {88, 55, 39, 41, 24, 51, 11}

                B - A                                   = {88, 55, 41, 51}

SYMMETRIC DIFFERENCE BETWEEN A & B

    SET A                     = {92, 11, 24, 76, 89, 44, 39, 3}

                SET B                     = {88, 55, 39, 41, 24, 51, 11}

                (A – B) + (B – A)  = {92, 76, 89, 44, 3, 88, 55, 41, 51}

**TESTING: Test your program for the following test cases**

1. Set A = {92, 11, 24, 76, 89, 44, 39, 3}

Set B = {88, 55, 39, 41, 24, 51, 11}

1. Set A = {87, 18, 68, 1, 49, 23}

Set B = { }

1. Set A = { }

Set B = {81, 61, 9, 31, 2}

**CONCLUSION:**

**Assignment No. 2**

**Problem Statement:**

Represent matrix using two-dimensional arrays and perform the following operations with

pointers: i. Addition ii. multiplication iii. transpose iv. Saddle point

**Objectives:** To write a neat code by following coding standards by selecting appropriate data structures

Outcomes: Students will be able to make use of coding standard for application development

**THEORY:**

**Pointer:**

A pointer is a variable that contains an address which is a memory location of another variable. For example suppose we define a pointer variable P for another variable quantity and memory location of variable quantity is 5000 then the contents of these variables in memory would be as given below

**Variable          Value              address in memory**

**Quantity         179                  5000**

**P                      5000                5048**

**Declaring Pointers**

**Format:           data\_type       \*pt\_name;**

This tells the compiler following three aspects

1.       The ‘\*’ in the beginning tells that pt\_name is a pointer variable

2.       just like other variables pt\_name also needs a memory location

3.       pt\_name points to the variable of type data\_type

for example

            int \*P;              /\*Pointer variable P points to variable of type integer\*/

            float \*Q;          /\*Pointer variable P points to variable of type float\*/

            char \*R            /\* Pointer variable P points to variable of type char\*/

**Initializing Pointers**

After the declaration of pointer variable, it can be made to point to a variable using an assignment statement. For example,

                        P = &Quantity;

Which causes P point to Quantity i.e. pointer variable will get initialized to the address of the variable quantity. This is known as pointer initialization and before using the pointer variable in program it must be declared and initialized. We must ensure that pointer variable always point to the corresponding data type variable only i.e. P and quantity both are having same data type say int.

**Accessing a variable through its pointer**

Once the pointer has been assigned the address of variable, then we can access the value of the variable using the pointer. This is done by another unary operator ‘\*’, usually known as the indirection operator. This operator is to be remembered as “value at”.

For example

                        int quantity, P, n;         /\* P is declared as pointer variable \*/

                        quantiy = 179; /\* variable quantity is initialized \*/

                        P = &quantity; /\* pointer variable P is initialized \*/

n = P;                           /\* variable is accessed through it’s pointer P\*/

**Pointers & Arrays**

**In general for 1-dimensional array &x[i] = x + i**

**And x[i] = \*( x + i ) = value at ( x + i )**

**2 dimensional arrays**

Consider two dimensional array defined as,

                        int x[3][3]  = {

                                                            { 15, 27, 11 },

                                                            { 22, 19, 31 },

                                                            {36, 23, 14 }

                                                }

            The elements of this array will be stored as,

**Element ->    x[0][0]  x[0][1]  x[0][2]    x[1][0]  x[1][1]  x[1][2]  x[2][0]  x[2][1]   x[2][2]**

**Value     ->      15         27            11           22         19          31         36          23        14**

**Address ->   1000      1002       1004      1006       1008    1010      1012     1014      1016**

In above example

                        Base address = address of 1st element in array x[3][3]

                                                = x

                                                = &x[0][0]

Address of 1st row = &x[0] = x = 1000

Address of 2nd row = &x[1] = x + 1 = 1006

Address of 3rd row = &x[2] = x + 2 = 1012

In general for 2-D array address of the row I = &x[i] = x + i \* 4 = \*(x + i )

                                And address of element j in row i = (\*(x + i ) + j )

                                    Value of element j in row i = x[i][j] = \*(\*(x +  i ) + j )

**INPUT/OUTPUT:**

**SAMPLE INPUT/ OUTPUT**

Enter the row and column size of matrix A: 2 2

Enter the elements of matrix A:

Enter A[0][0] : 1

Enter A[0][1] : 2

Enter A[0][2] : 3

Enter A[1][0] : 4

Enter A[1][1] : 5

Enter A[1][2] : 6

Enter A[2][0] : 7

Enter A[2][1] : 8

Enter A[2][2] : 9

Enter the row and column size of matrix B: 2 2

Enter the elements of matrix B:

Enter B[0][0] : 2

Enter B[0][1] : 3

Enter B[0][2] : 4

Enter B[1][0] : 5

Enter B[1][1] : 6

Enter B[1][2] : 7

Enter B[2][0] : 8

Enter B[2][1] : 9

Enter B[2][2] : 0

**OUTPUT**

            Matrix A is

                          1        2       3

                          4        5       6

                          7        8       9

            Matrix B is

                          2        3       4

                          5        6       7

                          8        9       0

Addition of matrix A & B

                          3        5       7

                          9        11    13

                          15      17     9

Matrix A is

                          1        2       3

                          4        5       6

                          7        8       9

            Matrix B is

                          2        3       4

                          5        6       7

                          8        9       0

Multiplication of matrix A & B

                          36      42     18

                          81      96     51

                          126    150   84

Matrix A is

                          1        2       3

                          4        5       6

                          7        8       9

            Transpose of matrix A

                          1        4       7

                          2        5       8

                          3        6       9

**Assignment No: 3**

**Problem Statement:**

Write a Menu driven program in C for a cricket player’s display board. The information of the

cricketer can be (not limited to) Name, Age, Country, Category (Batsman, Bowler, Wicket keeper, All-rounder), Number of ODI’s played, Number of International 20-20’s played. Display

following.

a) Number of batsman of a particular country

b) Sort the Batsman as per the average batting score(Bubble/insertion sort)

c) Batsman with highest average score (Binary search)

d) Number of bowlers of a particular country

e) The bowler that has taken a maximum no of wickets

f) Show particular players the entire “Display board information”

Delete/Modify the record

**Objectives:**

1. To write a neat code by following coding standards by selecting appropriate data structures.

2. To analyze algorithms with respect to time and space complexity.

3. To select appropriate searching and/or sorting techniques in the application development.

**Outcomes:**

Student will be able to

1. Make use of coding standard for application development.

2. Perform analysis of algorithms with respect to time and space complexity.

3. Select appropriate searching and/or sorting techniques in the application development.

**THEORY:**

**Sequential or Serial or Linear search:**

In this method, entities are searched one by one starting from the first to the end. Searching stops as soon as we reach to the required entity or we reach to the end. For example, consider the array containing ‘n’ integers and to search a given number in array we start searching from the 1st element of the array to the end of array. Searching will stop as soon as the element of the array is equal to the given number (number found) or we reach the end of the array (number not found). This method is generally used when the **data stored is not in the order of the key** on which we want to search the table.

**Binary Search:**

If the **array is sorted on the key** & we want to search for a key having given value, then **Binary search** technique can be used which is more efficient as compared to serial search. Binary search uses the ‘**divide & conquer**’ strategy as given below:

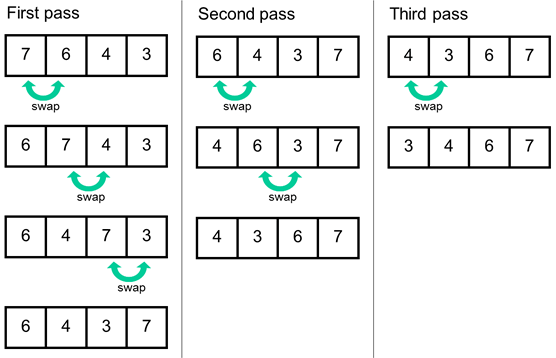
* 1. Divide the list into two equal halves.
  2. Compare the given key value with the middle element of the array or list. There are three possibilities.
     1. middle element = key : key found and search terminates
     2. middle element > key : the value which we are searching is possibly in the first half of the list
     3. middle element < key : the value which we are searching is possibly in the second half of the list
  3. Now search the key either in first half of second half of the list (b or c)
  4. Repeat steps 2 and 3 till key is found or search fails in case key does not exist.

**Bubble Sort:**

Bubble sort algorithm, is the simplest sorting algorithm that runs through the list repeatedly, compares adjacent elements, and swaps them if they are out of order.

The process of traversing the list is repeated until the list is sorted. The comparison sort algorithm is named after smaller or larger elements "bubble" at the top of the list.

Following images demonstrate the working of bubble sort



The Complexity of the Bubble Sort Algorithm

The Time Complexity of the Bubble Sort Algorithm

* Bubble sort employs two loops: an inner loop and an outer loop.
* The inner loop performs O(n) comparisons deterministically.

Worst Case

* In the worst-case scenario, the outer loop runs O(n) times.
* As a result, the worst-case time complexity of bubble sort is O(n x n) = O(n x n) (n2).

Best Case

* In the best-case scenario, the array is already sorted, but just in case, bubble sort performs O(n) comparisons.
* As a result, the [time complexity](https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity) of bubble sort in the best-case scenario is O(n).

Average Case

* Bubble sort may require (n/2) passes and O(n) comparisons for each pass in the average case.
* As a result, the average case time complexity of bubble sort is O(n/2 x n) = O(n/2 x n) = O(n/2 x n) = O(n/2 x n) = O (n2).

**SAMPLE INPUT/ OUTPUT**

Before Sorting array Elements:

12 31 25 8 32 17

After Sorting Array Elements:

8 12 17 23 31 32

**Assignment No: 4**

**Problem Statement:**

Implement Quick Sort to sort the given list of numbers. Display corresponding list in each pass.

**Objectives:**

1. To write a neat code by following coding standards by selecting appropriate data structures.

2. To analyze algorithms with respect to time and space complexity.

3. To select appropriate searching and/or sorting techniques in the application development.

**Outcomes:**

Student will be able to

1. Make use of coding standard for application development.

2. Perform analysis of algorithms with respect to time and space complexity.

3. Select appropriate searching and/or sorting techniques in the application development.

**THEORY:**

**Introduction**

**QUICK SORT :** Quick sort scheme developed by C.A.R. Hoare, has the best average

behaviour in terms of Time & Space Complexity among all the internal

sorting methods.

**BASIC STEP :** In Quick Sort the key Ki (pivot – generally the first element in an array)

controlling the process is placed at the right position with respect to the elements in an array i.e. if key Ki is placed at position s(i) then

Kj < Ks(i) for j < s(i) and

Kj >= Ks(i) for j > s(i)

Thus after positioning of the element Ki has been made, the original array is partitioned into two sub arrays, one consisting of elements K1 to (Ks(i) – 1) and the other partition (Ks(i) + 1) to Kn. All the elements in first sub array are less than Ks(i) & those in the second sub array are greater or equal to Ks(i). These two sub arrays can be sorted independently

**DIVIDE & CONQUER STRATEGY :**

From the above description of Quick Sort, it is evident that Quick Sort is a good example of **Divide & Conquer** strategy for solving the problem. In Quick Sort, we divide the array of items to be sorted into two partitions and then call the same procedure recursively to sort the two partitions. Thus we **divide** the problem in two smaller problems and **conquer** by solving the smallest problem, in this case sorting of partition containing only one element at the end, which is trivial one. Thus Quick Sort is the best example of divide & conquer strategy, where the problem is solved (conquered) by dividing the original problem into two smaller problems and then applying the same strategy recursively until the problem is so small that it can be solved trivially. Thus the conquer part of the Quick Sort looks like this.

**Initial Step**

| **< Pivot1** | **Pivot1** | **>= Pivot1** |
| --- | --- | --- |

**Next Step**

| **< Pivot2** | **Pivot2** | **>= Pivot2** |  | **Pivot1** | **>= Pivot1** |
| --- | --- | --- | --- | --- | --- |

**EXAMPLE :**

**Pass No** : **< -------- Input Array** **Elements -------- > Pivot m n i j**

**Position : 0 1 2 3 4 5 6 7 8 9**

**Pass 1** : **2 1 22 34 4 7 30 1 21 20** 2 0 9 1 9

**Pass 2** : **1 1** 2 34 4 7 30 22 21 20 1 0 1 1 1

**Pass 3** : 1 1 2 **34 4 7 30 22 21 20** 34 3 9 4 9

**Pass 4** : 1 1 2 **20 4 7 30 22 21** 34 20 3 8 4 8

**Pass 5** : 1 1 2 **7 4** 2030 22 2134 7 3 4 4 4

**Pass 6** : 1 1 2 4 720 **30 22 21** 34 30 6 8 7 8

**Pass 7** : 1 1 2 4 720 **21 22** 3034 30 6 7 7 7

**Sorted** **Array** : 1 1 2 4 72021 22 3034

**1) Time Complexity :**

1. **Worst Case :** Already sorted array is the worst case for Quick sort. Let us assume that there are ‘n’ elements in an array, then the number of elements in each partition for each pass will be,

##### Pass No Partitions No. of comparisions

Pass 1 : [0], [ n - 1] (n - 1)

Pass 2 : [0], [0], [n - 2] (n - 2)

Pass 3 : [0],.[0], [0], [n - 3] (n - 3)

. .

. .

. .

Pass (n-1) : [0],.[0], [0], …[0],[1] 1

Total number of comparisons would be = (n - 1) + (n - 2) + (n - 3) + …+ 1

= (n - 1) (n) / 2

= (n\*\*2 / 2) – (n / 2)

= O ( n\*\*2 ) (ignoring lower order

terms)

1. **Best Case :** Best case of Quick Sort would be the case where input elements in the array are such that in every pass **pivot** element is positioned at the middle such that the partition is divided in two partitions of equal size Then for a large value of n, the number of comparisons in each pass would be,

**After 1st Pass :** n comparisons + number of comparisons required to sort two

Partitions of size n / 2.

= n + 2 \* [n / 2]

**After 2nd Pass =** n + 2 \* [ n / 2 + 2 \* (n / 4)]

**=** n + n + 4\*[n / 4]

**=** 2n + 4\*[n / 4]

**After 3rd Pass =** 2n + 4\* [ n / 4 + 2 \* (n / 8)]

**=** 2n + n + 8\*[n / 8]

**=** 3n + 8\*[n / 8] = (log 8) \* 8 + 8\*[8/8] for 8 elements array

**= 8\*log8** since number of comparisons required to

sort partition of size 1 would be zero.

**. . .**

**. . .**

**. . .**

**After nth Pass = n\*log(n)**

1. **Average case :** Experimental results show that number of comparisons required to sort ‘n’ element array using Quick Sort are **O(nlogn)** to an average.

**2) Space Complexity :** In recursive quick sort, additional space for stack is required, which is of the order of **O(logn)** for **best and average case** and **O(n)** for **worst case.**

**SAMPLE INPUT/ OUTPUT**

Input: 56, - 90, 80, 78, 234, 654, 432, 12, 0, -11

Output: -90, -11, 0, 12, 56, 78, 80, 234, 432, 654

**Assignment No: 5**

**Problem Statement:**

Implement a sparse matrix with operations like initialize empty sparse matrix, insert an element, and perform transpose a matrix.

**Outcomes:**

Students will be able to:

4. Construct Linear data structures using arrays and linkedlists.

**THEORY:**

**Sparse Matrices:**

A general matrix of ‘m’ rows & ‘n’ columns of integers is shown below. In this matrix, we can see that there are very **few non-zero** entries or **many zero entries**. Such a matrix is called as **SPARSE MATRIX**. This 6 x 6 matrix contains only 8 non-zero terms out of total 36 terms. So it is a sparse matrix.

**Matrix A**

In practice we come across matrices of large sizes, say 1000 x 1000, which are most of the times are sparse matrices, and if we represent such matrices using conventional two dimensional array, then it would require 1 million (1000x1000) locations of computer memory and inputting 1 million (1000x1000) entries to populate the array of size 1000 x 1000 will take lot of time and chances of errors are also there. Further if we want to multiply two matrices of this size then we need to perform 1 billion (1000x1000x1000) multiplications! To utilize these resources (Inputting, Computer memory & Processor) more efficiently, we need to find out alternate way to represent sparse matrices in computer memory.

**Representation of sparse matrix:**

Sparse matrices can be stored in computer memory just by storing only non-zero entries form the original matrix. This can be achieved by storing the triple (row number, column number, value). Thus above matrix can be treated as an ordered list containing atoms (row number, column number, value) as given below.

{(1,1,15), (1,4,22), (1,6,-15), (2,2,11), (2,3,3), (3,4,-6), (5,1,91), (6,3,28)

This ordered list of triples can be stored in computer memory by using array.

**MATRIX A**

**A(0,0) = No. of rows**

**A(0,1) = No. of columns**

##### A(0,2) = No. of non-zero

**terms**

**Addition / Subtraction of sparse matrices:**

Addition / Subtraction of two sparse matrices using the representation as above is easy. It can be obtained by merging the two arrays on row number & column number. Using conventional representation, for the matrices of size m x n, total number of additions / subtractions would be m x n. But if there are ‘t1’ non-zero terms in one matrix and ‘t2’ non-zero entries in another, then maximum number of operations would be of the order of (t1 + t2). Generally for sparse matrices t1 << mn & t2 << t2 & this approach turns out to be efficient one. **Please refer the addition / subtraction procedures given in the section on Algorithms.**

**Transpose of a matrix:**

Transpose of a matrix is obtained by moving the elements so that the element at {i, j)th position is moved to (j, i)th position. In other words we can say that we are interchanging rows & columns of the elements of the matrix.

**Simple Transpose Method:**

To get the transpose of a matrix (column wise, row wise), we can write down all the elements at 1st column from the original array at the output, then elements of 2nd column are written to the output and so on until last column of the input array. Since the elements of original array are stored row wise, the same order is maintained in output as well when we write the elements at the output column wise. The time complexity of this method would be of the order of **n \* t**, where ‘n’ is the number of columns of the matrix and ‘t’ is the number of non-zero terms in the matrix.

But when ‘t’ approaches ‘mn’ i.e. number of zero terms are very less, then the performance becomes worst of the order of (m \* square of n) which is worst as compared to the conventional method of getting transpose, which is of the order (mn).. Hence this method becomes impracticable when ‘t’ approaches ‘mn’.

**Fast Transpose Method:**

By using some more storage, we can do much better as compared to simple transpose. We can in fact transpose a matrix represented as a sequence of triples in time of the order (n + t) using fast transpose method.

This algorithm proceeds by first determining the number of elements in each column of array A. This gives the number of elements in each row of it’s transpose, say array B. From this information, the starting position of each row in array B can be easily obtained. Thereafter we can just move the elements of array A one by one into their correct position in array B. See the example given below. Since the time complexity is of the order of **(n + t),** even in worst case as‘t’ approaches ‘mn’, it would become n(t + 1) i.e. of the order of ‘mn’.

**void** getdata(struct sparse a [] )

{

1. Declare **int** t1,t2;
2. **Display**(“enter the number of rows, columns & non-zero terms of matrix ”) ;

**Accept** a[0].row, a[0].col, a[0].val //store information in 0th row of

Structure a

1. **Display**(“enter the elements of sparse matrix ”);
2. t1=a[0].val;
3. for(i=1; i<=t1; i++)

Accept a[i].row, a[i].col, a[i].val;

}//end getdata();

**void** add\_spa\_matrix( **struct** sparse a [], **struct** sparse b [])

**{** /\* this procedure adds two sparse matrices A & B (arrays a & b) and store the result in

array c \*/

1. **Declare** struct sparse c[MAX];
2. **Declare** **int** m1 = a[0]. row, n1 = a[0].col, t1 = a[0].val
3. **Declare int** m2 = b[0].row, n2 = b[0].col, t2 = b[0].val, q, i, j;
4. **if** ( m1 != m2 && n1 != n2)

{

**Display** (“Addition not possible”);

**return;**

**}**

1. **if**  (t1 = = 0) /\* Matrix A is zero matrix \*/

{ /\*Copy b to c \*/

**for** ( i = 0; i <= t2; i++)

{ c[i] = b[i];

Display c[i].row, c[i].col, c[i].val;

}

**return;**

}//end if

1. **if**  ( t2 = = 0) /\* Matrix A is zero matrix \*/

{ /\*Copy a to c\*/

**for** ( i = 0; i <= t1; i++)

{ c[i].row = a[i];

Display c[i].row, c[i].col, c[i].val;

}

**return;**

}//end if

1. i=1; j=1; q=1;

**while**  ( i <= t1 || j <= t2)

{

**if** (a [ i ].row > b [ j ].row) /\* compare rows \*/

{

c [ q ] = b [ j ];

j++, q++;

}

**else if** (a [ i ].row < b [ j ].row)

{

c [ q ] = a [ i ];

i++, q++;

}

**else if** (a [ i ].col > b [ j ].col) /\* for equal rows \*/

{

c [ q ] = b [ j ]; /\* compare columns \*/

j++, q++;

}

**else if** (a [ i ].col < b [ j ].col)

{

c [ q ] = a [ i ];

i++, q++;

}

**else /\*** equal row & column \*/

{

c [ q ] = a [ i ];

c[ q ].val + = b [ j ].val

i++, j++, q++;

}

}//end while

1. c[0].row =m1; c[0].col=n1; c[0].val=q;
2. **Diplay** (“Addition of Matrix a and b”)
3. **for**  ( i=1; i<=q; i++ )

Display c[i].row, c[i].col, c[i].val;

} //end add\_spa\_matrix

**void** simple\_transpose( struct sparse\_mtx a [])

{

/\* this procedure transposes the matrix a & stores the result in array b using simple transpose

method \*/

1. struct sparse\_mtx b[MAX};
2. **declare** **int**  n = a[0].col, t = a[0].val, c, i, q=1;
3. b[0].row = a[0].col;

b[0].col = a[0].row;.

b[0].val = a[0].val;

1. **if** ( t = = 0)

**return;**

1. **for** (c = 0; c < n; c++) /\* transpose by column \*/

{

**for** ( i = 1; i <= t; i++) /\* for all non-zero terms in a \*/

{

**if (**a[i].col = = c) /\* insert next term in b \*/

{

b[q].row = a[i].col;

b[q].col = a[i].row;

b[q].val = a[i].val;

q++;

}

}//inner for

}//outer for

1. **Diplay** (“Transpose of Matrix a”)
2. **for**  ( i=1; i<=t; i++ )

Display b[i].row, b[i].col, b[i].val;

}//end simple\_trans

**void fast**\_transpose ( struct sparse\_mtx a[ ])

{/\* this procedure transposes the matrix a & stores the result in array b using fast transpose

method \*/

**Declare** struct sparse b [MAX];

**Declare** int n = a[0].col, t = a[0].val, i, j, q = 1, column;

**Declare** int S[n], T[n];

for (i=0;iNn;i++)

{ S[n]=0; T[n]=0; }

b[0].row = n; b[0].col = a[0].row ; b[0].val = t;

**if** ( t = = 0)

**return;** /\* check for zero matrix \*/

**for** (i = 1; i <= t; i++) /\* S[i]=number of terms in row i of b\*/

{

S[ a[ i ].col ]++;

}

T[0]=1; //T[i]=stating position of row i in b

**for** (i = 1; i < n; i++)

T[ i ] = T[ i -1] + S[ i -1];

**for** ( i = 1; i <= t; i++) /\* move all t elements from a to b\*/

{

k = a[ i ].col; /\* k = value at col in array a \*/

j = T[k]; /\* j = position in array c \*/

b [ j ].row = a [ i ].col; /\* moving triple from a to c \*/

b [ j ].col = a [ i ].row;

b [ j ].val = a [ i ].val;

T[ k ] ++; /\* increment starting position \*/

}

**Display** (“Transpose of Matrix a”)

**for**  ( i=1; i<=t; i++ )

Display b[i].row, b[i].col, b[i].val;

}// end fast\_transpose

**INPUT/OUTPUT:**

**Sample**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* INPUT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter number of rows & columns in Matrix A : 04 04

Enter number of non-zero terms in Matrix A : 05

Enter the row wise, column wise triples of matrix A:

Enter Triple 1 : 0 0 9

Enter Triple 2 : 0 1 15

Enter Triple 3 : 1 2 4

Enter Triple 4 : 2 1 2

Enter Triple 5 : 3 3 6

Enter number of rows & columns in Matrix B : 04 04

Enter number of non-zero terms in Matrix B : 06

Enter the row wise, column wise triples of matrix B:

Enter Triple 1 : 0 1 10

Enter Triple 2 : 1 0 2

Enter Triple 3 : 1 3 4

Enter Triple 4 : 2 1 3

Enter Triple 5 : 3 0 1

Enter Triple 6 : 3 3 2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* OUTPUT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Addition of Sparse Matrix A & B

Matrix A

Row Col Val

A[0] 4 4 5

A[1] 0 0 9

A[2] 0 1 15

A[3] 1 2 4

A[4] 2 1 2

A[5] 3 3 6

Matrix B

Row Col Val

B[0] 4 4 6

B[1] 0 1 10

B[2] 1 0 2

B[3] 1 3 4

B[4] 2 1 3

B[5] 3 0 1

B[6] 3 3 2

A + B =

Row Col Val

C[0] 4 4 8

C[1] 0 0 9

C[2] 0 1 25

C[3] 1 0 2

C[4] 1 2 4

C[5] 1 3 4

C[6] 2 1 5

C[7] 3 0 1

C[8] 3 3 8

## Simple Transpose:

## Matrix A

Row Col Val

A[0] 4 4 5

A[1] 0 0 9

A[2] 0 1 15

A[3] 1 2 4

A[4] 2 1 2

A[5] 3 3 6

Transpose of A

Row Col Val

C[0] 4 4 5

C[1] 0 0 9

C[2] 1 0 15

C[3] 1 2 2

C[4] 2 1 42

C[5] 3 3 6

Total number of comparisons : 20

## Fast Transpose:

## Matrix A

Row Col Val

A[0] 4 4 5

A[1] 0 0 9

A[2] 0 1 15

A[3] 1 2 4

A[4] 2 1 2

A[5] 3 3 6

Transpose of A

Row Col Val

C[0] 4 4 5

C[1] 0 0 9

C[2] 1 0 15

C[3] 1 2 2

C[4] 2 1 42

C[5] 3 3 6

Total number of comparisons : 9

**SAMPLE INPUT/ OUTPUT**

**Assignment No: 06**

**Problem Statement:**  Implement a Polynomial with operations like create an empty polynomial, insert an entry into a polynomial, add two polynomials and evaluate a polynomial.

**Objectives:**

1. To identify suitable data structures and algorithms to solve real-world problems.

**Outcomes:**

Students will be able to:

1. CO4 Construct Linear data structures using arrays and linkedlists

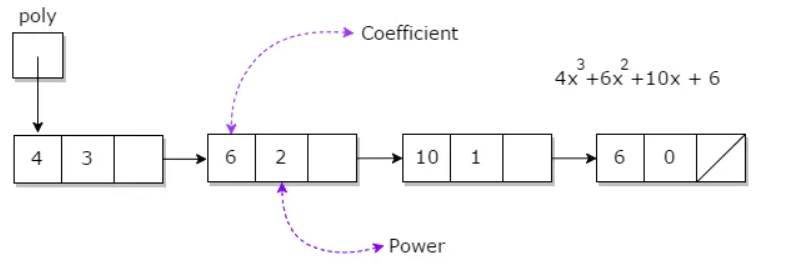
**THEORY:**

A polynomial p(x) is an expression in variable x of the form (axn + bxn-1 +.... + jx+ k), where a, b, c...., k are real numbers and 'n' is a non negative integer, which is known as the degree of the polynomial. Given two polynomial numbers represented by a linked list, a polynomial is made up of several terms, each of which has a coefficient and an exponent. Add these lists with a function that adds the coefficients with the same variable powers

* The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
* Additional terms having equal exponent is possible one
* The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent







**SAMPLE INPUT/ OUTPUT**

Input:

1st number = 6x2 + 3x1 + 2x0

2nd number = -2x1 - 5x0

Output:

6x2+1x1-3x0

Input:

1st number = 6x3 + 2x2 + 2x0

2nd number = 2x^1 - 5x^0

Output:

6x3 + 2x2 + 2x1 - 5x0

**Assignment No: 07**

**Problem Statement:**  Department of Information Technology has a student's club named 'Pinnacle Club'. Students of the Second, third and final year of the department can be granted membership on request. Similarly, one may cancel the membership of the club. The first node is reserved for the president of the club and the last node is reserved for the secretary of the club. Write C program to maintain club member’s information using a singly linked list. Store student PRN and Name. Write functions to  
a) Add and delete the members as well as the president or even secretary.  
b) Compute the total number of members of the club  
c) Display members of the club  
Display list in reverse order using recursion

**Objectives:**

1. To identify suitable data structures and algorithms to solve real-world problems.

**Outcomes:**

Students will be able to:

1. CO4 Construct Linear data structures using arrays and linkedlists

**THEORY:**

Linked Lists, like arrays, are linear data structures. Unlike arrays, linked list elements are not stored in a single location; instead, pointers are used to connect the components.

o A linked list is a collection of objects called nodes that are stored in memory at random.

o A node comprises two fields: data saved at that address and a reference to the next node in the memory.

o The list's last node contains a pointer to the null

## Algorithm

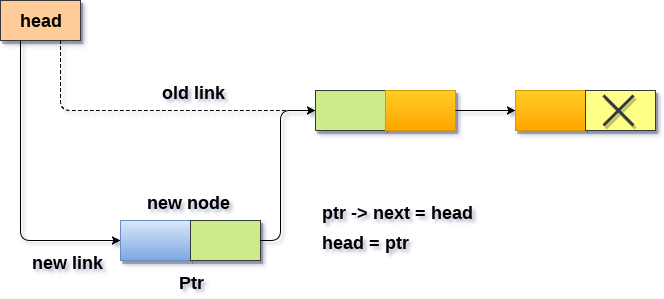
* **Step 1:** IF PTR = NULL

WriteOVERFLOW

     Go to Step 7

    [END OF IF]

* **Step 2:** SET NEW\_NODE = PTR
* **Step 3:** SET PTR = PTR → NEXT
* **Step 4:** SET NEW\_NODE → DATA = VAL
* **Step 5:** SET NEW\_NODE → NEXT = HEAD
* **Step 6:** SET HEAD = NEW\_NODE
* **Step 7:** EXIT



|  | col 1 | col 2 | col 3 | col 4 | col 5 | col 6 |
| --- | --- | --- | --- | --- | --- | --- |
| row 1 | 15 | 0 | 0 | 22 | 0 | -15 |
| row 2 | 0 | 11 | 3 | 0 | 0 | 0 |
| row 3 | 0 | 0 | 0 | -6 | 0 | 0 |
| row 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| row 5 | 91 | 0 | 0 | 0 | 0 | 0 |
| row 6 | 0 | 0 | 28 | 0 | 0 | 0 |
|  | 0 | 1 | 2 |
|  | Row No. | Column No. | Value |
| A(0, | 6 | 6 | 8 |
| A(1, | 1 | 1 | 15 |
| A(2, | 1 | 4 | 22 |
| A(3, | 1 | 6 | -15 |
| A(4, | 2 | 2 | 11 |
| A(5, | 2 | 3 | 3 |
| A(6, | 3 | 4 | -6 |
| A(7, | 5 | 1 | 91 |
| A(8, | 6 | 3 | 28 |



**SAMPLE INPUT/ OUTPUT**

Input:

Enter Name of President: Atharva

Enter PRN Number of President : 001

Enter the number of members: 02

Enter Name of Member: Anamika

Enter PRN Member:007

Enter Name of Member: Shiksha

Enter PRN Member:009

Enter Name ofsecretary: Nakul

Enter PRN Number of secretary: 005

Output:



**Assignment No: 8**

**Problem Statement:**

Implement any database using a doubly-linked list with the following options

a) Insert a record

b) delete a record

c) modify a record

d) Display list forward

Display list backward

**Outcomes:**

Students will be able to:

1. Construct Linear data structures using arrays and linked lists.

**THEORY:**

**Data Structures to be used :** Doubly Linked List

**struct** dll

{

**int** accno;

**char** custname[20];

**int** balance;

**struct** dll \*prev;

**struct** dll \*next;

};

**typedef struct** dll node;

**Doubly Linked List:**

In Singly Linked List, each node contains one link which points to the successor node. In SLL it is not possible to insert or delete ith  node if we are given with only the address of ith  node. For accomplishment of this we need the address of previous node. One has to traverse the list from first node (head node) to the ith  node and maintain the previous node address as well while traversing the SLL.

To overcome the difficulty mentioned above, we can use Doubly Linked List in which each node contains two links, one to it’s predecessor and othe to it’s successor.

* **Singly Linked List (SLL):**

**first or head**

| A |  |  | B |  |  | C | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- |

* **Doubly Linked List (DLL):**

**first or head**

| 0 | **A** |  |  |  | **B** |  |  |  | **C** | **0** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

**Comparison between SLL & DLL:**

| **Srl. No.** | **Singly Linked List (SLL)** | **Doubly Linked List (DLL)** |
| --- | --- | --- |
| **1** | Link to successor node only | One link to successor & other to predecessor |
| **2** | Successor Link field of last node is set to NULL to mark the end of list | Successor Link field of last node is set to NULL to mark the end of the list & predecessor link of head node is set to NULL |
| **3** | Starting from first node only we can reach to any node in the list | Starting from any node we can reach to any other node in the list |
| **4** | Head node is always the first node of the list | Head node can be first or last node of the list |
| **5** | It is not possible to reach to previous node from the current node in the list | It is possible to reach to previous node from current node |
| **6** | Previous node address is necessary for Insertion / Deletion. | Current node address is enough for Insertion / Deletion of a node |
| **7** | Suitable for the implementation of Stack | Suitable for the implementation of Deque |

**Concept/Working**

**Advantages/Disadvantages**

**Any Software or Hardware Used**

**ALGORITHM:**

**void** main( )

{

/\* menu to implement Doubly Linked List \*/

1. **Declare** structure to represent DLL globally

**struct**  dll

{

**int** accno;

**char** custname[20];

**int** balance;

**struct** dll \*prev;

**struct** dll \*next;

};

**typedef struct** dll node;

1. **Declare**  node \*head;
2. **Declare char** str[20];
3. **Declare node\*** createnode (int ano, char name[], int bal);

**node**\* create\_dll ( );

**node**\* insert\_dll();

**node** \* delete\_dll();

**Void** display\_dll (node \*);

**Void** display\_rev\_\_dll( node \*) ;

1. **Display** the following menu

#### DOUBLY LINKED LIST OPERATIONS

**1. Create DLL to store string**

**2. Insert a new character**

**3.Delete a node from the list**

**4. Display the string**

**5. Reverse Display of the string**

**6. Quit**

**Select the Option :**

1. **Accept** the option
2. **switch** (opt)

{

**case 1:**

**{**

head=create\_dll();

**break;**

**}**

**case 2:**{

head=insert\_dll(s);

**break;**

}

**case 3:**{

head=delete\_dll(s);

**break;**

}

**case 4:**{

display\_dll(head);

**break;**

}

**case 5:**{

display\_rev\_dll(head);

**break;**

}

**case 6: exit(0);**

}

}//end main

--------------------------------------------------------------------------------------------------------------------

**node\* createnode (int ano, char name[], int bal;)**

**{**/\* this procedure creates a doubly linked list node for given character in a string\*/

1. **Declare** node \*temp=NULL;

temp = (node \***)**malloc(sizeof (node));

1. if( temp = = NULL)

{

Display “ Allocation Failed”

exit(0);

}

else

{

temp->prev=temp->next=NULL;

temp->accno=ano;

strcpy(temp-> custname, name);

temp-> balance=bal;

}

1. **return**(temp);

**}//**end create\_dll

-

**node \*** create\_dll ()

{

/\* this procedure creates a doubly linked list for given string\*/

1. **Declare** node \*h=NULL,\*new1,\*temp;
2. **Declare int** i=0;
3. Declare ano, name bal,
4. Enter no of records n
5. for( i=0; i<n; i++)
6. {

**Display** (“enter the data in each record”);

Accept ano, name, bal;

temp =new1;

new1 = createnode (ano , name, bal);

if( h = = NULL)

h=new1;

else

{

temp->next = new1;

new1->prev = temp;

}

}

1. **return**(h);

**}//**end create\_dll

---------------------------------------------------------------------------------------------------------------------

**node \*** insert\_dll (node \*head)

{ /\* this procedure inserts a new node in a doubly linked list \*/

1. **Declare** node \*new1, \*temp, \*p**;**
2. **Declare int before\_ano;**;
3. Declare ano,name bal,

1. **Display** (“enter the data to be inserted”);

Accept ano, name, bal;

1. new1 = createnode(ano,name ,bal); **/\*** dynamic memory allocation\*/
2. **Display** (“enter the account no. before which you want insert the new record”);
3. Accept before\_ano
4. **if** (head = = NULL) /\* insertion when SLL is empty \*/

head = new1;

**else if** (before\_ano =**=** head->accno) /\* insertion prior to head node \*/

{ new1->next = head; /\*i.e at the front\*/

head->prev= new1;

head = new1;

}

**else**

{ /\*insertion at the middle or at the end\*/

p=head;

while ( p!=NULL )

{

if( p->accno = = before\_ano)

**break;**

p=p->next;

}

temp = p->prev;

temp->next = new1;

new1->prev = temp;

new1->next = p;

p->prev =new1;

}//else

**10) return** (head);

}//end insert\_dll

--------------------------------------------------------------------------------------------------------------

**node \*** delete\_dll (node \*head)

{ /\* this procedure deletes node in a doubly linked list \*/

1. **Declare** node \*tp,\*ta \*p**;**
2. **int** flag=0;
3. **Display** (“enter the account no. to be deleted”);
4. **Accept** del\_ano;
5. **if** (head = = NULL) /\* DLL is empty \*/

{ Display (“DLL is emty”);

**return**(head);

}

1. **if** (del\_ano =**=** head->accno) /\* deletion of the head node \*/

{ p=head;

head=head->next;

head->prev= NULL;

**delete** (p);

**return**(head);

}

1. p=head /\*deletion at the middle or at the end\*/
2. while ( p!=NULL )

{

if( p->accno = = del\_ano)

{

flag=1;

**break;**

}

p=p->next;

}

if (flag = = 1)

{

(p->prev)->next = p->next;

(p->next)->prev = p->prev;

delete(p);

}

else

Display(“Data not found”);

1. **return**(head);

}//end delete\_dll

-------------------------------------------------------------------------------------------------------------------

**void** display\_dll (node \*head)

{

/\* this procedure displays all all characters of the strings in forward manner\*/

1. **declare** node \*p=head**;**
2. **if** (p = = NULL)

{ **display (**“DLL is empty”)

**return;**

}

1. while ( p!=NULL)

{

**Display** p->accno;

**Display** p->custname;

**Display** p->balance;

p=p->next;

}

}//end display\_dll

--------------------------------------------------------------------------------------------------------------------

**void** display\_rev\_dll (node \*head)

{

/\* this procedure displays all all characters of the strings in reverse order\*/

1. **declare** node \*p=head**;**
2. **if** (p = = NULL)

{ **display (**“DLL is empty”)

**return;**

}

1. while ( p->next!=NULL)

p=p->next;

1. while( p!= NULL)

{

**Display** p->accno;

**Display** p->custname;

**Display** p->balance;

p = p->prev;

}//end display\_dll

**Assignment No: 09**

**Problem Statement:**  To convert decimal numbers to binary numbers using the stack as sequential or linkedrepresentation.

**Objectives:**

To implement abstract properties of various data structures such as stacks, queues.

**Outcomes:**

Students will be able to:

CO4 Construct Linear data structures using arrays and linkedlists.

CO5 Make use of abstract properties of various data structures such as stacks, queues

**THEORY:**

## Whatisstack?Explainstackoperationswithneatdiagrams.

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **stack** is a last in, first out ([LIFO](http://en.wikipedia.org/wiki/LIFO_(computing))) [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) and [datastructure](http://en.wikipedia.org/wiki/Data_structure). A stack can have any [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) as an element, but is characterized byonly two fundamental operations:push and pop. The push operation adds an item to thetop of the stack, hiding any items already on the stack, or initializing the stack if it is empty.A pop either reveals previously concealed items, or results in an empty stack. A stack is arestricted data structure, because only a small number of operations are performed on it.The nature of the pop and push operations also mean that stack elements have a naturalorder. Elements are removed from the stack in the reverse order to the order of theiraddition:therefore,thelowerelementsarethosethathavebeenonthestackthelongest.A collection of items in which only the most recently added item may be removed. Thelatest added item is at the top. Basic operations are push and pop. Often top and isEmptyareavailable,too.Also knownas"last-in, first-out"orLIFO.

The programperformsthefollowingfunctionsandoperations:

* + **Push**:Pushesanelementtothestack.Ittakesanintegerelementasargument.Ifthestackisfullthenerrorisreturned.
  + **Pop**:Popanelementfromthestack.Ifthestackisemptythenerrorisreturned.

Theelementisdeletedfromthetopofthestack.

* + **DisplayTop**:Returnsthetopelementonthestackwithoutdeleting.Ifthestackisempty thenerrorisreturned.

Given a decimal number as input, we need to write a program to convert the given decimal number into equivalent binary number.  
 **Algorithm**:

1. Store the remainder when the number is divided by 2 in an array.
2. Divide the number by 2
3. Repeat the above two steps until the number is greater than zero.
4. Print the array in reverse order now.

**SAMPLE INPUT/ OUTPUT**

Input:

Input : 7

Output : 111

Input : 10

Output : 1010

Input: 33

Output: 100001

**Assignment No: 10**

**Problem Statement:**

Implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix, and evaluation of postfix and prefix expression.

**Objectives:**

4. To implement abstract properties of various data structures such as stacks, queues.

**Outcomes:**

Students will be able to:

5. Make use of abstract properties of various data structures such as stacks, queues.

**THEORY:**

**Stacks:**

Stacks are more common data objects found in computer algorithms. It is a special case of more general data object, an ordered or linear list. It can be defined as an ordered list, in which all insertions & deletions are made at one end, called as ‘top’ i.e. Last element inserted is outputted first (**L**ast **I**n **F**irst **O**ut or **LIFO**) or first element inputted is outputted last (**F**irst **I**n **L**ast **O**ut – **FILO**).

Stack can be represented mathematically as :

S = {a1, a2, …,an} where a1 is the bottommost element & an is topmost element.&

ai+1 is on the top of the element ai , 1 < i <= n.

**Implementation of stack as an ADT using array :**

Following operations are required to implement stack as an ADT.

* **Creation of Stack:**

In ‘C’ stack of integers can be defined as,

struct st

{ int top;

int stack[maxsize];

};

struct st stk; /\* stk is a working variable \*/

Initialize top to –1 to ensure that stack is empty to begin with.

* **Pushing an element into Stack:**

If ‘n’ is the size of the stack then for pushing (adding) a new element into stack following ‘C’ code is necessary.

if (top >= (n-1))

**display “stack is full”**

stop;

top = top + 1;

stack[top] = item to be inserted or pushed into stack;

* **Popping an element from Stack**

For the deletion of top element from stack following ‘C’ code is necessary.

if (top < 0)

**display “stack is empty”**

stop;

item = stack[top];

top = top – 1;

**Implementation of stack as an ADT using Linked List :**

Following operations are required to implement stack as an ADT using linked list.

* **Creation of Stack:**

In ‘C’ stack of integers can be defined as,

struct linked-list /\* structure for SLL \*/

{ int number;

struct linked-list \*next;

} \*top, q, p**; /** all are pointers of type struct linked-list\*/;

typedef struct linked-list node;

node stk; /\* stk is a working variable \*/

Initialize top to NULL to ensure that stack is empty to begin with.

* **Pushing an element into Stack:**

It is equivalent to add a node before head node in SLL

p = (node \*)malloc(sizeof(node));

p->number = accepted as input;

p->next = top;

top = p; /\* new node becomes head node \*/

inserted or pushed into stack;

* **Popping an element from Stack**

It is equivalent to deletion of a head node from SLL.

if (top = NULL)

**display “stack is empty”**

stop;

q = top;

top = top->next;

q->next = NULL

free(q);

**Applications of Stacks:**

* Conversion of Decimal Number into Binary Number
* Reversing the character string
* Recursive procedures
* Well formed parenthesis
* Expression conversion and evaluation

For the detailed algorithms on expression conversion and evaluation, please refer to the section on ALGORITHMS.

typedef struct node

{

char data;

struct node \*next;

}stack;

stack \*push(stack \*top,char data)

{

stack \*new\_node=(stack\*)malloc(sizeof(stack));

new\_node->data=data;

new\_node->next=NULL;

if(top==NULL)

{

top=new\_node;

}

else

{

new\_node->next=top;

top=new\_node;

}

return top;

}

int is\_empty(stack \*top)

{

if(top==NULL)

return 1;

return 0;

}

stack\* pop(stack \*top)

{

stack \*temp=top;

// char ch=top->data;

top=top->next;

free(temp);

return top;

}

void display(stack\*top)

{

stack \*temp=top;

if(top==NULL)

{

printf("Stack is empty");

return;

}

while(temp!=NULL)

{

printf("\n\t|%d|",temp->data);

temp=temp->next;

}

}

char readtop(stack\*top)

{

return top->data;

}

int precedence(char x)

{

if(x=='(')

return 0;

if(x=='+' || x=='-')

return 1;

if(x=='\*' || x=='/')

return 2;

return 3;

}

void infix\_postfix(char infix[20],char postfix[20])

{

stack \*top=NULL;

int i=0,j=0;

char token,x;

for(i=0;infix[i]!='\0';i++)

{

token=infix[i];

if(isdigit(token))

{

postfix[j]=token;

j++;

}

else if(token=='(')

{

top=push(top,token);

}

else if(token==')')

{

while((x=readtop(top))!='(')

{

postfix[j]=x;

j++;

top=pop(top);

}

top=pop(top);

}

else

{

while(!is\_empty(top) && precedence(token)<=precedence(readtop(top)))

{

postfix[j]=readtop(top);

top=pop(top);

j++;

}

top=push(top,token);

}

}

while(is\_empty(top)!=1)

{

postfix[j]=readtop(top);

top=pop(top);

j++;

}

postfix[j]='\0';

}

Procedure eval\_postfix(char postfix[20])

{

stack \*top=NULL;

int i=0,j=0;

char token,x;

for(i=0;postfix[i]!='\0';i++)

{

token=postfix[i];

if(isdigit(token))

{

top=push(top,token-48);

}

else

{

int op1,op2;

op2=readtop(top);

top=pop(top);

op1=readtop(top);

top=pop(top);

switch(token)

{

case '+':x=op1+op2; break;

case '-':x=op1-op2; break;

case '\*':x=op1\*op2; break;

case '/':x=op1/op2; break;

}

top=push(top,x);

}

}

x=readtop(top);

top=pop(top);

return x;

}

**Assignment No: 11**

**Problem Statement:**

Implement Circular Queue using Array. Perform following operations on it.

a) Insertion (Enqueue)

b) Deletion (Dequeue) Display

**Objectives:**

1. To write a neat code by following coding standards by selecting appropriate data structures.

4. To implement abstract properties of various data structures such as stacks, queues.

**Outcomes:**

Students will be able to:

5. Make use of abstract properties of various data structures such as stacks, queues.

**THEORY:**

**Queues:**

Queues are more common data objects found in computer algorithms. It is a special case of more general data object, an ordered or linear list. It can be defined as an ordered list, in which all insertions are allowed at one end, called as ‘rear’ & all deletions are made at other end, called as ‘front’ i.e. First element inserted is outputted first (**F**irst **I**n **F**irst **O**ut or **FIFO**) or last element inputted is outputted last (**L**ast **I**n **L**ast **O**ut – **LILO**).

Queue can be represented mathematically as :

S = {a1, a2, …,an} where a1 is the first element & an is last element.&

ai is ahead of ai+1 , 1 =< i < n.

**Implementation of queue as an ADT using array :**

Following operations are required to implement queue as an ADT.

* **Creation of Circular Queue:**

In ‘C’ queue of integers can be defined as,

class que

{ int front;

int rear;

int number[maxsize];

};

que queue; /\* queue is a object \*/

Initialize front = rear = 0 to ensure that queue is empty to begin with.

* **Addition of an element into Queue:**

Insertion of an item in circular queue stored as number(0:maxsize –1), rear points to the last item & front points to one position less counterclockwise from first item in a queue. Following is the ‘C’ code to do this.

queue.rear = (queue.rear + 1) mod (maxsize);

if (queue.front = queue.rear)

**display “queue is full”**

stop;

queue.number[queue.rear] = item to be inserted into queue.

* **Deletion of an element from Queue:**

For deletion of front element from circular queue following ‘C’ code is necessary.

if (queue.front = queue.rear)

**display “queue is empty”**

stop;

queue.front = (queue.front + 1) mod (maxsize)

item = queue.number[queue.front];

**Implementation of queue as an ADT using Linked List :**

Following operations are required to implement queue as an ADT using linked list.

* **Creation of Circular Queue:**

In ‘C’ queue of integers can be defined as,

class linked-list /\* structure for SLL \*/

{ int number;

struct linked-list \*next;

} \*rear, q, p**; /** all are pointers of type struct linked-list\*/;

typedef struct linked-list node;

node queue; /\* queue is a working variable \*/

Initialize queue.rear to NULL to ensure that queue is empty to begin with.

* **Addition of an element into Queue:**

It is equivalent to add a node after last node in CSLL

p = (node \*)malloc(sizeof(node));

p->number = accepted as input;

if (queue.rear != NULL)

p->next =queue.rear->next

queue.rear->next = p;

queue.rear = p; /\* new node becomes head node \*/

else p->next = p

queue.rear = p;;

* **Deletion of an element from Queue**

It is equivalent to deletion of a head node from CSLL.

if (queue.rear = NULL)

**display “queue is empty”**

stop;

q = queue.rear->next;

if (q = queue.rear) /\* only one node in CSLL \*/

rear = NULL;

else queue.rear->next = q->next; /\* more than one nodes in CSLL\*/

queue.rear->next = q->next;

q->next = NULL

free(q);

**Applications of Queues:**

* Job scheduling
* Railway / Air-line reservation system

**Assignment No: 12**

**Problem Statement:**  Implement priority queue as ADT using a single linked list for servicing patients in a hospital withpriorities as  
a) Serious (top priority)  
b) medium illness (medium priority)  
c)General (Least priority)

**Objectives:**

To implement abstract properties of various data structures such as stacks, queues.

To identify suitable data structures and algorithms to solve real-world problems

**Outcomes:**

Students will be able to:

CO5 Make use of abstract properties of various data structures such as stacks, queues.

CO6 Identify suitable data structure and algorithm to solve a real-world problem.

**THEORY:**

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure. Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer. Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists. Queue is a data structure that maintain "First In First Out" (FIFO) order. And can be viewed as people queueing up to buy a ticket. In programming, queue is usually used as a data structure for BFS (Breadth First Search).

# ALGORITHM:

DefinestructureforQueue(Priority,Patient Info,Next Pointer).

# EmptyQueue:

ReturnTrueifQueueis EmptyelseFalse.isEmpty(Front)

Front is pointer of structure,which is first element of Queue.Step 1:IfFront==NULL

Step 2:Return1

Step 3:Return0

# InsertFunction:

InsertPatient inQueuewithrespect to the Priority.

Front ispointervariableof type Queue,whichis1stnodeofQueue.

PatientisapointervariableoftypeQueue,whichholdtheinformationaboutnewpatient.

Insert(Front, Queue )

Step 1: IfFront==NULL//QueueEmptyThenFront =Patient;

Step 2: Else if Patient->Priority >Front->PriorityThen i)Patient->Next=Front;

* 1. Front=Patient;

Step 3:ElseA) Temp =Front;

* + 1. Do StepsawhileTemp!=NULLAndPatient->Priority<=Temp->Next->Priority
       1. Temp=Temp->Next;

c) Patient->Next = Temp->Next;Temp->Next=Patient;

Step4: Stop.

# DeletePatientdetailsfromQueueafterpatientgettreatment:

Front is pointer variable of type Queue,which is 1st node of Queue.DeleteNodefromFront.

Delete(Front )

Step1:Temp=Front;

Step 2: Front = Front->Next;Step 3:returnTemp

# DisplayQueueFront:

Front is pointer variable of type Queue,which is 1st node of Queue.Display(Front)

Step1:Temp=Front;

Step2:Do Stepswhile Temp!=NULL

1. DisplayTempData
2. IfPriority 1 Then ―General Checkup‖;ElseIfPriority2ThenDisplay―Non-serious";ElseIfPriority 3Then Display "Serious"

ElseDisplay "Unknown";

1. Temp = Temp->Next;Step3:Stop.

# DisplayQueuerear:

Front is pointer variable of type Queue,which is 1st node of Queue.Display(Rear)

Step1:Temp=Rear;

Step2:Do Stepswhile Temp!=NULL

1. DisplayTempData
2. IfPriority1Then―GeneralCheckup‖;

**Assignment No: 13**

**Problem Statement:**

Department maintains student information. The file contains roll number, name, division, and address. Write a program to create a sequential file to store and maintain student data. It should allow the user to add, delete information of the student. Display information of the particular employee. If the record of the student does not exist an appropriate message is displayed. If a student record is found it should display the student details.

**Outcomes:**

Students will be able to:

1. Make use of coding standards for application development.

**THEORY:**

**ypes of Files**

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

**1. Text files**

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

**2. Binary files**

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

**File Operations:**

* Creation of a new file.
* Opening an existing file.
* Reading data from a file.
* Writing data in a file.
* Closing a file

| Function | Uses/Purpose |
| --- | --- |
| [fopen](https://www.w3schools.in/c-tutorial/file-handling/fopen/) | Opens a file. |
| [fclose](https://www.w3schools.in/c-tutorial/file-handling/fclose/) | Closes a file. |
| [getc](https://www.w3schools.in/c-tutorial/file-handling/getc/) | Reads a character from a file |
| [putc](https://www.w3schools.in/c-tutorial/file-handling/putc/) | Writes a character to a file |
| [getw](https://www.w3schools.in/c-tutorial/file-handling/getw/) | Read integer |
| [putw](https://www.w3schools.in/c-tutorial/file-handling/putw/) | Write an integer |
| [fprintf](https://www.w3schools.in/c-tutorial/file-handling/fprintf/) | Prints formatted output to a file |
| [fscanf](https://www.w3schools.in/c-tutorial/file-handling/fscanf/) | Reads formatted input from a file |
| [fgets](https://www.w3schools.in/c-tutorial/file-handling/fgets/) | Read string of characters from a file |
| [fputs](https://www.w3schools.in/c-tutorial/file-handling/fputs/) | Write string of characters to file |
| [feof](https://www.w3schools.in/c-tutorial/file-handling/feof/) | Detects end-of-file marker in a file |

# **MODE OF OPERATIONS PERFORMED ON A FILE IN C LANGUAGE:**

There are many modes in opening a file. Based on the mode of file, it can be opened for reading or writing or appending the texts. They are listed below.

* r – Opens a file in read mode and sets pointer to the first character in the file. It returns null if file does not exist.
* w – Opens a file in write mode. It returns null if file could not be opened. If file exists, data are overwritten.
* a – Opens a file in append mode.  It returns null if file couldn’t be opened.
* r+ – Opens a file for read and write mode and sets pointer to the first character in the file.
* w+ – opens a file for read and write mode and sets pointer to the first character in the file.
* a+ – Opens a file for read and write mode and sets pointer to the first character in the file. But, it can’t modify existing contents.