Name: Sujan Biswas **Subject: Data Structure and Algorithms Lab**
Class Roll No: 302010501003 **Assignment Set-2**

Question1.

Define an ADT for Polynomials.

Write C data structure representation and functions for the operations on the Polynomials in a Header file.

Write a menu-driven main program in a separate file for testing the different operations and include the above header file.

Solution Approach:

In mathematics, a polynomial is an expression consisting of variables and constants (coefficients) and involves only addition, subtraction, multiplication and non-negative integer exponentiation of variables. If we consider only one variable, a polynomial P(x) is a function such that

$P(x)=\sum_{i=0}^{n} a_i x^i$ for all x in the domain (usually real numbers, in this case each ai is a real number and are called the coefficients of the polynomial; we can compose polynomials based on any kind of set of "numbers" where addition, subtraction and multiplication are closed binary operations on the set). Each term in the given series is called a term of the polynomial and comprises only the coefficient (positive or negative) multiplied by the variable raised to some non-negative integer exponent (called the degree of the term). If a particular term of degree j is absent, we take aj to be zero for that term. The highest degree of the terms present (n) is called the degree of the entire polynomial. With this definition, we can say a polynomial of degree n is a series of n + 1 terms arranged from degree 0 to degree n, with "absent" terms assigned coefficient 0.

In mathematics, the following operations are available for polynomials:

1. **Addition:**

   P(x) = F(x) + G(x) implies adding all the terms one after the another – P(x) will be such that terms with common degree will have their coefficients added, and "uncommon degree" terms simply listed and all the terms arranged from 0 to n. Example: $(x^3 + 3x^2 + 1) + (x^4 + x^2 - 10x) = x^4 + x^3 + 4x^2 - 10x + 1$ (terms with no listed coefficient have coefficient 1). The degree of resultant in addition operation is max(degree of F, degree of G).

2. **Multiplication with a constant (Called scalar multiplication in linear algebra parlance):**

   P(x) = c . F(x) implies multiplying each coefficient of F(x) with c. Degree in resultant remains the same.

3. **Subtraction:**

   P(x) = F(x) – G(x) implies P(x) = F(x) + (-1) . G(x).

4. **Multiplication with a monomial:**

   A monomial is a special kind of polynomial with a single term having non-zero coefficient. If we take each term of P(x) as $a_i x^i$ and the monomial as bx^c then P(x) . bx^c is such that each ith term is $a_i b x^{i+c}$. The degree in resultant increases (from the multiplicand polynomial) by c.

5. **Multiplication with another polynomial:**

If F(x) has degree m and G(x) has degree n, then F(x).G(x) will be a polynomial of degree m + n and can be computed as follows: take each term from G(x) from degree 0 to degree n as a monomial, multiply with F(x) to find $H_i(x)$ (0 <= i <= m) and add all these $H_i(x)$ so formed.

For the purpose of implementing in a computer system, these operations are also defined:

Checking if the polynomial is zero (meaning all the coefficients are zero and the final value of the polynomial function does not depend on x).

Evaluating the polynomial i.e. finding its value as a (real) number given a value for x.
Obtaining the coefficient of any degree term – absent terms and terms with a higher degree than the degree of the polynomial have coefficient zero.

Finding/obtaining the degree of the polynomial.

For the purposes of implementation, we will represent a polynomial A(x) of degree n as an array A[0..n] (end indexes inclusive) of coefficients; thereby coefficient of term with degree j is A[j] if 0 <= j <= n, otherwise 0 (operation 6). An array was chosen for the advantage of obtaining coefficient of any term in constant time.

Structured Pseudocode:

Notice that in some operations, an array A[0..n] may have some trailing terms 0 so the corresponding degree of polynomial will be less than n. We need to search for the first highest degree non-zero term.

```
degree(A[0..n]) → ans //where ans is the degree of the polynomial. Operation 7

        for i = n to 1 step -1 inclusive

                if A[i] not equal to 0: return i

        return 0 //in any case of the degree 0 term, the degree is zero if we reach here

is_zero(A[0..n]) → boolean //true if the corresponding polynomial is zero.
//Operation 4

        return degree(A) == 0 and A[0] == 0

coeff(A[0..n], exponent) → c /*where c is the coefficient of term with degree exponent;
assumed exponent >= 0. Operation 6*/

        if exponent <= n: return A[exponent]
```

```
                else return 0
```

add(A[0..m], B[0..n]) → (K, C[0..K]) /*get a tuple having the degree of the answer and the final polynomial. We assume A[m] and B[n] are non-zero. Operation 1*/

```
                K = max(m, n)
                Create an array C[0..K]
                for i = 0 to K inclusive: C[i] = coeff(A, i) + coeff(B, i)
                return (K, C)
```

scalar_mul(A[0..n], c) → B[0..n] /*Multiply A(x) with constant/scalar c, assuming A[n] = non-zero. Operation 2*/

```
                Create an array B[0..n]
                for i = 0 to n inclusive: B[i] = A[i] * c
                return B
```

sub(A[0..m], B[0..n]) → (K, C[0..K]) /*Same return definition and requirements as add but performs A(x) – B(x). Operation 3*/

```
                K = max(m, n)
                Create an array C[0..K]
                for i = 0 to K inclusive: C[i] = coeff(A, i) - coeff(B, i) //perform subtraction
                return (K, C)
```

mul(A[0..m], B[0..n] → (K, C[0..K]) → /*Same return definition and requirements as add and sub but performs A(x).B(x). We have only one method for multiplication for both monomial and polynomial. Operation 4 and 5*/

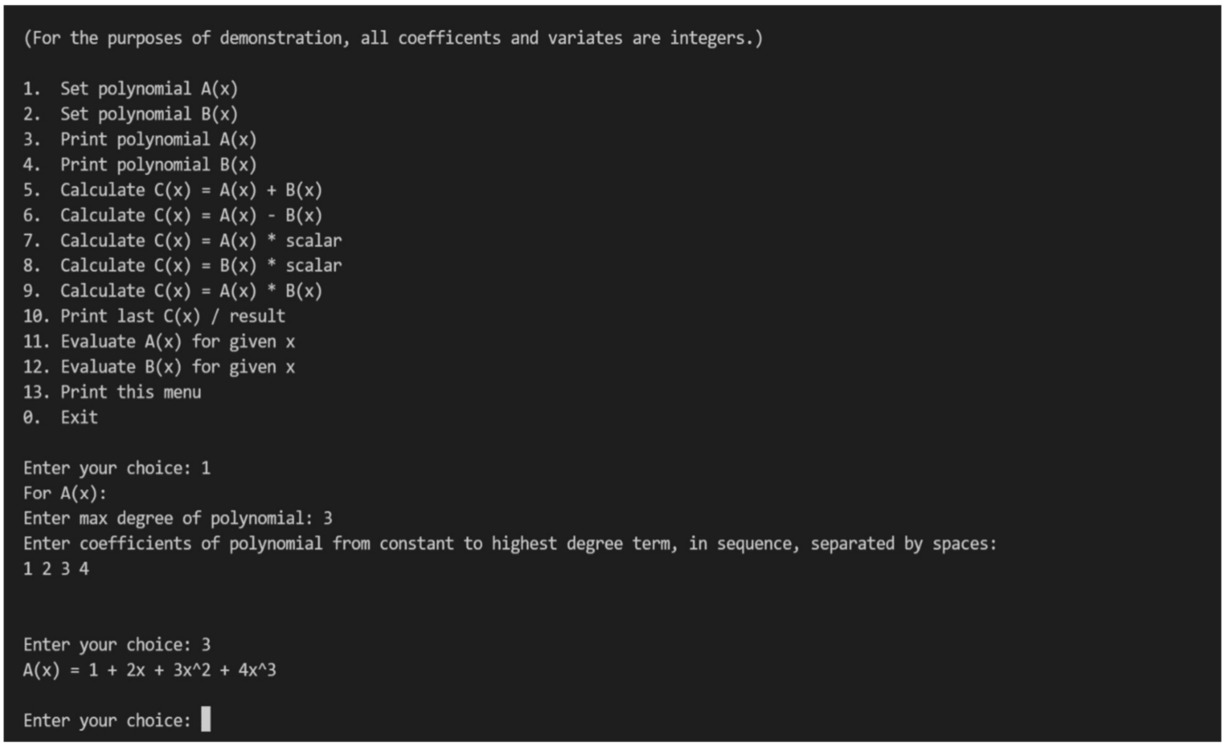
```
                /*We can use the fact that a term ckx k in product C(x) will be such that
                  $c_k=\sum a_I b_J$    for all I + J = k, I <= m, J <= n.
                */
                K = m + n
```

```
        Create an array C[0..K] and initialize all elements to 0
        for i = 0 to K inclusive {
                for j = 0 to i inclusive: C[i] = C[i] + coeff(A, j) * coeff(B, i - j) /*iterate over all
        possible pairs of i, j for that degree*/


        return (K, C)
```

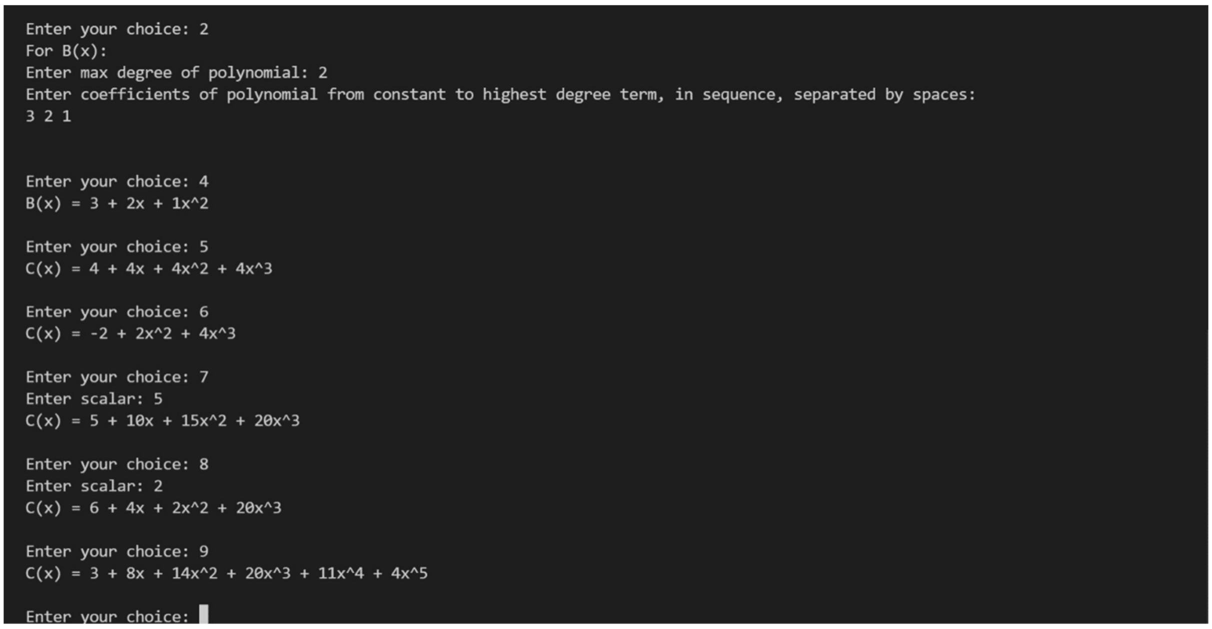
/*We will use something called Horner's rule for evaluation of the polynomial that works by converting it into monomials. For example, 1.x + 2.x2 + 3.x3 + 4.x4 will be evaluated as x(1 + x(2 + x(3 + x.4))). The advantage of using this method is unnecessary exponentiations of the variate are avoided.*/

```
    //evaluate y = A(x). Assume A[n] is nonzero. Operation 5
    evaluate(A[0..n], x) → y {
        y = 0
        for i = n to 0 step -1 inclusive: y = A[i] + y * x
        return y
```

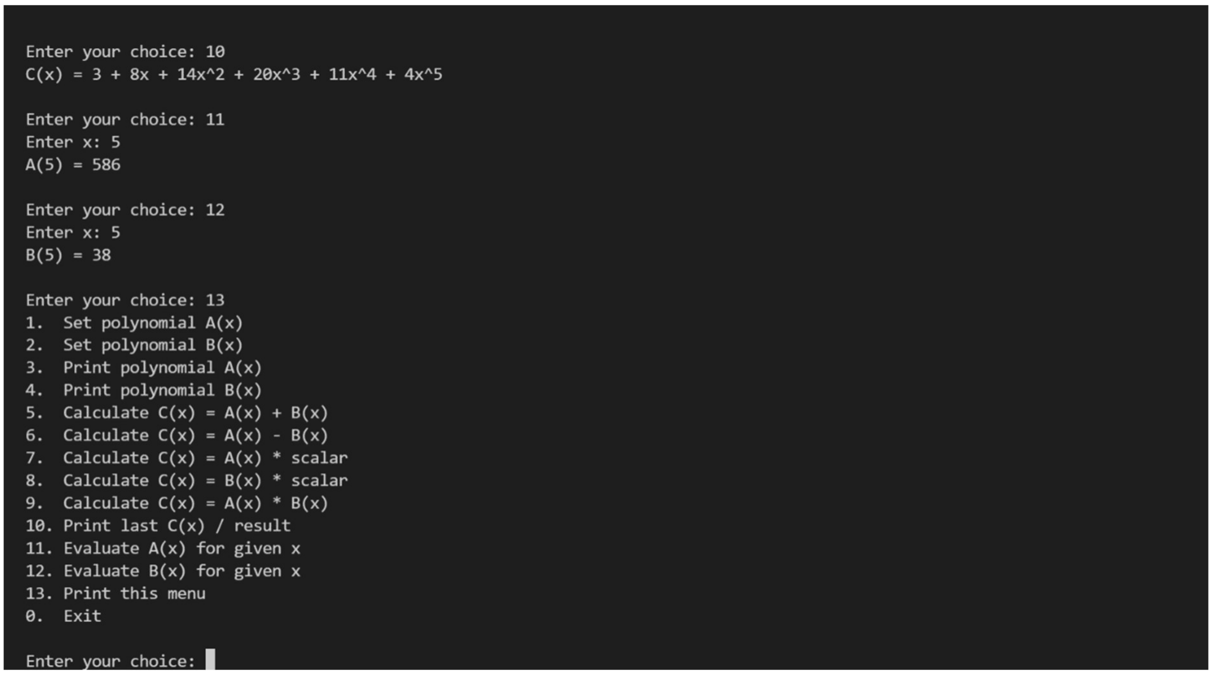
Results:



```
(For the purposes of demonstration, all coefficents and variates are integers.)

1.  Set polynomial A(x)
2.  Set polynomial B(x)
3.  Print polynomial A(x)
4.  Print polynomial B(x)
5.  Calculate C(x) = A(x) + B(x)
6.  Calculate C(x) = A(x) - B(x)
7.  Calculate C(x) = A(x) * scalar
8.  Calculate C(x) = B(x) * scalar
9.  Calculate C(x) = A(x) * B(x)
10. Print last C(x) / result
11. Evaluate A(x) for given x
12. Evaluate B(x) for given x
13. Print this menu
0.  Exit

Enter your choice: 1
For A(x):
Enter max degree of polynomial: 3
Enter coefficients of polynomial from constant to highest degree term, in sequence, separated by spaces:
1 2 3 4


Enter your choice: 3
A(x) = 1 + 2x + 3x^2 + 4x^3

Enter your choice: 
```



```
Enter your choice: 2
For B(x):
Enter max degree of polynomial: 2
Enter coefficients of polynomial from constant to highest degree term, in sequence, separated by spaces:
3 2 1


Enter your choice: 4
B(x) = 3 + 2x + 1x^2

Enter your choice: 5
C(x) = 4 + 4x + 4x^2 + 4x^3

Enter your choice: 6
C(x) = -2 + 2x^2 + 4x^3

Enter your choice: 7
Enter scalar: 5
C(x) = 5 + 10x + 15x^2 + 20x^3

Enter your choice: 8
Enter scalar: 2
C(x) = 6 + 4x + 2x^2 + 20x^3

Enter your choice: 9
C(x) = 3 + 8x + 14x^2 + 20x^3 + 11x^4 + 4x^5

Enter your choice:
```



```
Enter your choice: 10
C(x) = 3 + 8x + 14x^2 + 20x^3 + 11x^4 + 4x^5

Enter your choice: 11
Enter x: 5
A(5) = 586

Enter your choice: 12
Enter x: 5
B(5) = 38

Enter your choice: 13
1.  Set polynomial A(x)
2.  Set polynomial B(x)
3.  Print polynomial A(x)
4.  Print polynomial B(x)
5.  Calculate C(x) = A(x) + B(x)
6.  Calculate C(x) = A(x) - B(x)
7.  Calculate C(x) = A(x) * scalar
8.  Calculate C(x) = B(x) * scalar
9.  Calculate C(x) = A(x) * B(x)
10. Print last C(x) / result
11. Evaluate A(x) for given x
12. Evaluate B(x) for given x
13. Print this menu
0.  Exit

Enter your choice:
```

Source Code:

p1_polynomial.h contains code for the operations on polynomials as described above.
p1_polytest.c is a menu-driven tester program for the code developed in the header file.
Both conform to ANSI C89.

Question2.

Define an ADT for Sparse Matrix.

Write C data structure representation and functions for the operations on the Sparse Matrix in a Header file.

Name: Sujan Biswas **Subject: Data Structure and Algorithms Lab**
Class Roll No: 302010501003 **Assignment Set-2**

Write a menu-driven main program in a separate file for testing the different operations and include the above header file.

Solution Approach:

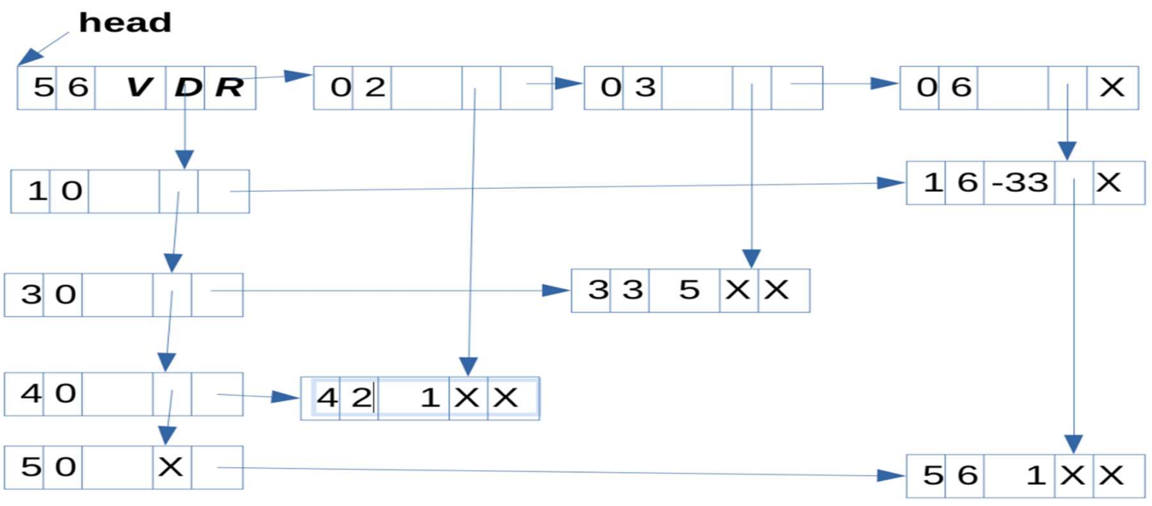
A sparse matrix is a matrix with most of its elements as 0s (usually number of non-zero elements < number of zero elements). They should behave like normal matrices and support at least the basic operations allowed on matrices in computer representations: Finding the dimensions, accessing an element at a position and changing an element in a position (other operations such as addition, multiplication etc. can be implemented on the basis of these operations; although knowing the nature of the implementation the implementer can supply optimised versions). Sparse matrices are considered because usual implementations do not store all of the elements (only the non-zero elements are stored); as a result memory space is saved.

The implementation we have chosen is based on linked lists and is called a multi-list, in which each node is a part of multiple linked lists at once (sort of like a network). The way we chose to implement sparse matrices this way can be best explained with an example.

Original/source matrix:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & -33 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{matrix}$$

...will be represented in our multi-list format as described:



Each node (schematic represented in the node pointed to by head) has these elements: two indexes (5 and 6) , a value field (V) and two references for other nodes: the down reference (D) and the right reference (R). Each node represents a non-zero value in the matrix – but not all the nodes hold values, some are special header nodes. The first node (head) is considered a header for the entire matrix and holds the dimensions of the matrix (5 x 6). The right reference for head is the starting reference to a linked list for the column headers – each node having 0 as the row number (we use indexes starting from 1 for actual row/column positions). The column headers are in a linked list with head through the right reference; and for any particular column the down reference is a starting reference to the nodes in the same column (the ones with both indexes not equal to 0 hold actual values). Similarly, the down reference from head is a starting reference for the row headers (having column index 0) and each row header's right reference is a starting reference for that row. The point to be noted is that empty (all zero) rows and columns do not have headers and zero positions have no node representing them.

This representation is suitable only when it is known that nonzero elements will be few in an otherwise (possibly large) matrix. This representation gets grossly inefficient as the number of nonzero elements increases.

Structured Pseudocode:

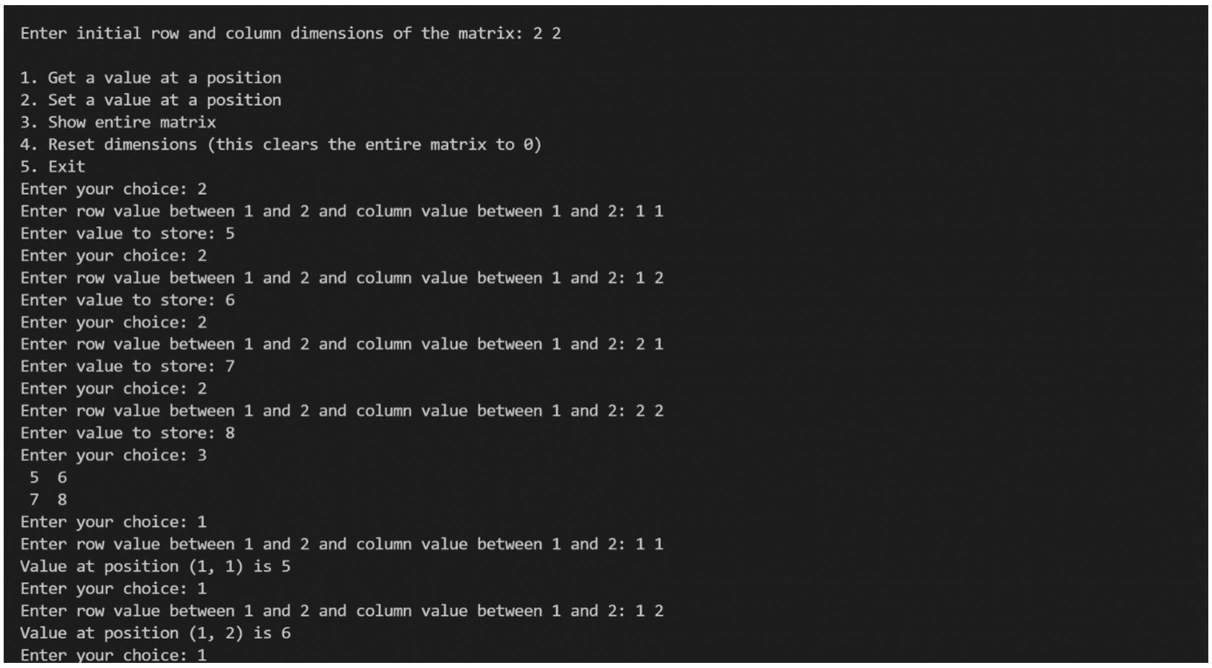
We will assume there is a special value NIL for references that don't point to anything (X's in the above picture).

```
//Definition for node, it has these properties

struct node

    row, column, info, ref down, ref right

//Dimensions can be obtained from head.row and head.column

get_element(matrix_header, r, c) → value /*where value is the value at row r and column c,
assume r and c are within dimensions*/


    row_p = matrix_header→down //→used for member access

    while row_p→row < r: row_p = row_p→down //Search for the row moving down

    if row_p→row > r: return 0 //We overshot row hence entire row is zero

    col_p = row_p->right

    while col_p→column < c: col_p = col_p→right //Search for the column moving right

    if col_p→column > c: return 0 //We overshot the column hence zero

    else return col_p→info


set_element(matrix_header, r, c, value) //Set value at r, c position, assumed valid


    target = NIL //To hold the target node

    //Search for col header

    prev_col_hdr = matrix_header, col_hdr = matrix_header→right

    while col_hdr→column < c {

        prev_col_hdr = col_hdr

        col_hdr = col_hdr→right
```

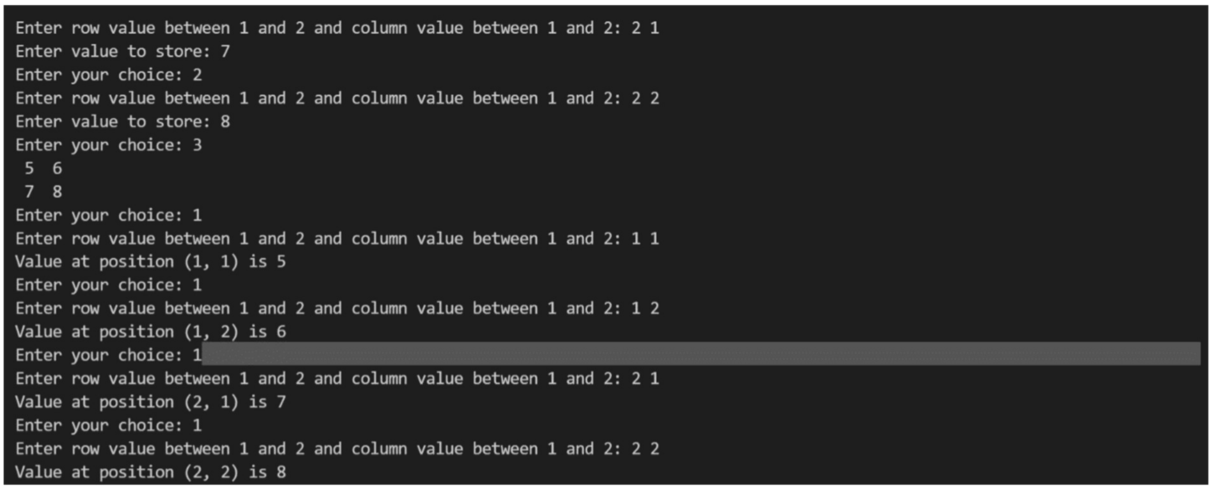
```
if col_hdr→column > c //Column does not exist(entire column was zero)
    if value == 0: return //We do not need to do anything
    new_col = Construct a new column header node
    Insert new_col into the list using prev_col_hdr and col_hdr
    col_hdr = new_col
    col_hdr→down = NIL //To indicate this is a new column
    //Column header inserted

prev_row_hdr = matrix_header, row_hdr = matrix_header→down
//Now searching for row header
while row_hdr→row < r
    prev_row_hdr = row_hdr
    row_hdr = row_hdr→down

if row_hdr→row > r //Row does not exist (entire row was zero)
    if value == 0: return //We do not need to do anything
    Insert a new row header and make row_hdr refer to the correct header as
was done for the column header
    row_hdr→right = NIL //to indicate a new row

//Now search within the row as we did for column headers
prev_row_elem = row_hdr, row_elem = row_hdr→right
while row_elem→column < c
    prev_row_elem = row_elem
    row_elem = row_elem→right

(Similarly search within the column moving down, current element in col_elem and
previous element in prev_col_elem)
if row_elem and col_elem refer to the correct node
    if value == 0
```

```
            (We need to remove the element; remove the element from both
        the column and row using prev_col_elem and prev_row_elem.)

        else
            target = row_elem
            target→info = value //Place correct value as required

    elif value != 0 //Insert only if value was not zero
        target = new node
        Insert target into row and column lists using prev_row_elem, row_elem and
    prev_col_elem, col_elem

    (It is possible that row_hdr→right is NIL or col_hdr→down is NIL, due to removal of
a node. Remove them from row headers list and column headers list as well.)
```

Results:



```
Enter initial row and column dimensions of the matrix: 2 2

1. Get a value at a position
2. Set a value at a position
3. Show entire matrix
4. Reset dimensions (this clears the entire matrix to 0)
5. Exit
Enter your choice: 2
Enter row value between 1 and 2 and column value between 1 and 2: 1 1
Enter value to store: 5
Enter your choice: 2
Enter row value between 1 and 2 and column value between 1 and 2: 1 2
Enter value to store: 6
Enter your choice: 2
Enter row value between 1 and 2 and column value between 1 and 2: 2 1
Enter value to store: 7
Enter your choice: 2
Enter row value between 1 and 2 and column value between 1 and 2: 2 2
Enter value to store: 8
Enter your choice: 3
 5  6
 7  8
Enter your choice: 1
Enter row value between 1 and 2 and column value between 1 and 2: 1 1
Value at position (1, 1) is 5
Enter your choice: 1
Enter row value between 1 and 2 and column value between 1 and 2: 1 2
Value at position (1, 2) is 6
Enter your choice: 1
```



```
Enter row value between 1 and 2 and column value between 1 and 2: 2 1
Enter value to store: 7
Enter your choice: 2
Enter row value between 1 and 2 and column value between 1 and 2: 2 2
Enter value to store: 8
Enter your choice: 3
 5  6
 7  8
Enter your choice: 1
Enter row value between 1 and 2 and column value between 1 and 2: 1 1
Value at position (1, 1) is 5
Enter your choice: 1
Enter row value between 1 and 2 and column value between 1 and 2: 1 2
Value at position (1, 2) is 6
Enter your choice: 1
Enter row value between 1 and 2 and column value between 1 and 2: 2 1
Value at position (2, 1) is 7
Enter your choice: 1
Enter row value between 1 and 2 and column value between 1 and 2: 2 2
Value at position (2, 2) is 8
```

Source Code:

p2_sparsematrix.h defines the spnode data type (and implements related operations as given above) which represents a single node in the multi-list and also acts as the header for an entire matrix.

p2_test.c is a menu_driven program for testing the code in the header file and demonstrating its use. Both conform to ANSI C89.

Question3.

Define an ADT for List.

Write C data structure representation and functions for the operations on the List in a Header file with array as the base data structure.

Write a menu-driven main program in a separate file for testing the different operations and include the above header file. Two data structures with and without using sentinels in arrays are to be implemented

Solution Approach:

A list is a sequence of (possibly duplicated) elements of same type (sequence implies there is a start element, end element and other elements will have a predecessor and successor). The very basic operations as can be defined on lists are insertion of elements, deletion of elements and access of elements (traversal). Since we are using arrays as a backing structure for lists (called array lists or vectors) the operations that are defined and implemented are as follows:

1. Obtain the length of the list (number of elements)

2. Read the list from either direction (start to end or in reverse)

3. Retrieve the ith element

4. Store a new value (overwrite) at ith position

5. Insert a new element at ith position (we state/define that the old elements starting from position i to the end are shifted one position towards the end)

6. Delete an element at ith position (we state/define that the elements starting from position i+1 to the end are shifted one position towards the start)

7. Search the list for a specified given value (called the key)

8. Sort the list in some order on the value of the elements. For the given implementation of the sorting algorithm in this paper, we use selection sort which proceeds as follows: for ascending order, search the minimum element, put it into the 1st position, and then repeat the process considering the sublist formed from the 2nd position to the end, and so continue to put all elements at their correct positions. We chose selection sort as it is conceptually quite easy to understand (any other sorting algorithm can be used as desired).

For the given operations below, we will consider NIL to be a sentinel value indicating the end of the array. Such given operations can easily be converted to non-sentinel versions by holding the length of the list as a property of the list.

Structured Pseudocode:

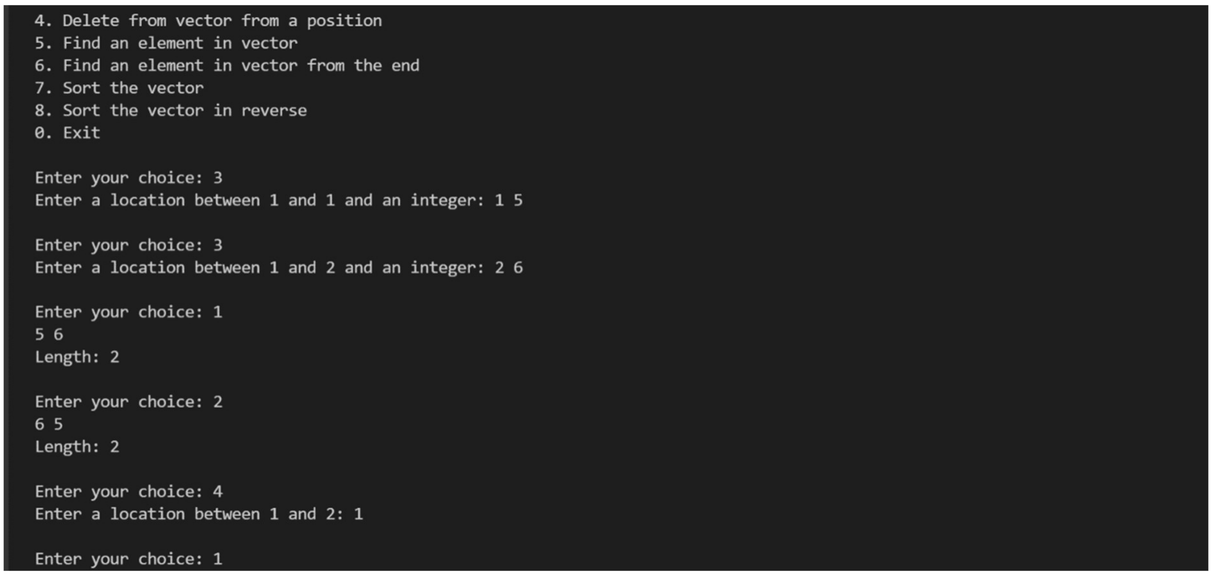
We call our such list objects backed by an array as vectors, which are nothing but arrays. Positions start from

1. Vector read (iteration/traversal) in either direction, store and retrieval operations are trivial.

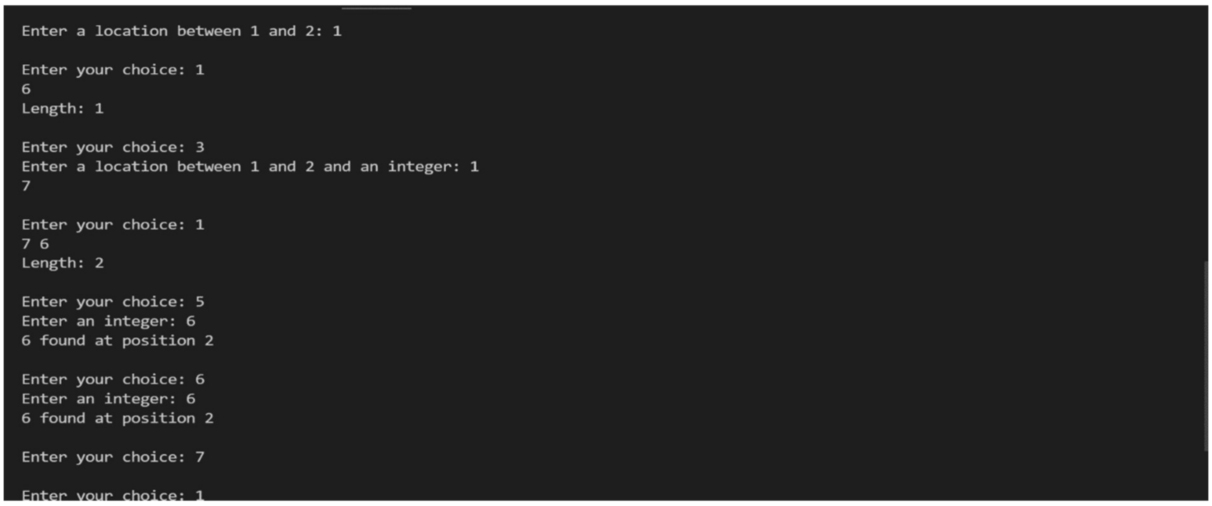
```
vector_length(V) //get length of V.
  i = 1
  while V[i] is not equal to NIL: i = i + 1
  return i – 1 //Length of vector is number of elements excluding NIL.


vector_traverse(V, direction) //traverse V in given direction.
  if direction is first to last
        i = 1
      while V[i] is not equal to NIL
        visit(V[i]) //Visit the element (traversal)
        i = i + 1



  else
        len = vector_length(V)
        for i = len to 1 step -1 inclusive: visit(V)
```

```
    vector_insert(V, i, element) /*Insert element at position i. Assume 1 <= i <= length of V + 1
and V has space to hold the extra element.*/
    j = vector_length(V)
    V[j + 2] = NIL //Assign proper terminator for vector
        while j >= i
            V[j + 1] = V[j] //Shift each towards the end
            j = j – 1

        V[i] = element

    vector_delete(V, i) //Delete element at position i. Assume 1 <= i <= length of V.
        j = i
        while V[j+1] is not NIL
            V[j] = V[j + 1] //Shift each towards the start
            j = j + 1

        V[j] = NIL //Terminate the vector

    vector_search(V, element, direction) /*Search V for element in the direction specified.
Return suitable index.*/
        i = 1
        idx = NIL //Store final index
        while V[i] is not NIL
            if V[i] equals element
                idx = i
                if direction is first to last: return idx //Exit on first occurrence in case
            of first-to-last search

            i = i + 1

        return idx //Return last occurrence in case of last-to-first search
        }

    vector_sort_ascending(V) /*Sort the vector in ascending order, here we use selection sort;
this assumes elements that can be compared*/

        len = vector_length(V)

        for i = 1 to len inclusive

            Search for the minimum element between i to len inclusive, and store the
            index of such element in k Swap V[i] with V[k]
```

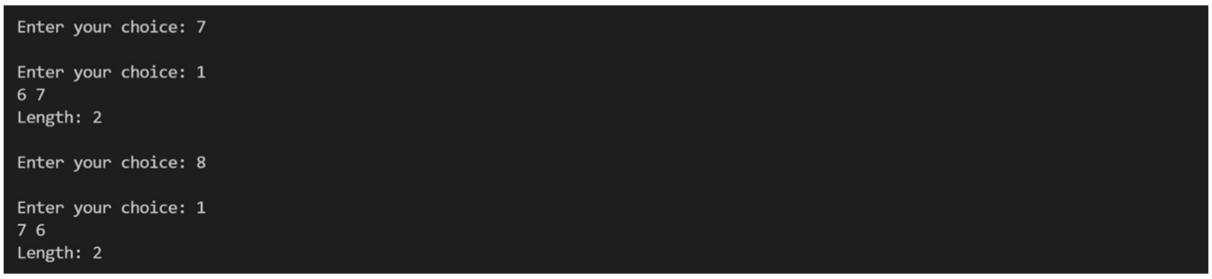
Results:



```
4. Delete from vector from a position
5. Find an element in vector
6. Find an element in vector from the end
7. Sort the vector
8. Sort the vector in reverse
0. Exit

Enter your choice: 3
Enter a location between 1 and 1 and an integer: 1 5

Enter your choice: 3
Enter a location between 1 and 2 and an integer: 2 6

Enter your choice: 1
5 6
Length: 2

Enter your choice: 2
6 5
Length: 2

Enter your choice: 4
Enter a location between 1 and 2: 1

Enter your choice: 1
```



```
Enter a location between 1 and 2: 1

Enter your choice: 1
6
Length: 1

Enter your choice: 3
Enter a location between 1 and 2 and an integer: 1
7

Enter your choice: 1
7 6
Length: 2

Enter your choice: 5
Enter an integer: 6
6 found at position 2

Enter your choice: 6
Enter an integer: 6
6 found at position 2

Enter your choice: 7

Enter your choice: 1
```



```
Enter your choice: 7

Enter your choice: 1
6 7
Length: 2

Enter your choice: 8

Enter your choice: 1
7 6
Length: 2
```

Name: Sujan Biswas **Subject: Data Structure and Algorithms Lab**
Class Roll No: 302010501003 **Assignment Set-2**

Source Code:

p3_vector_sentinel.h is the implementation which uses sentinels and

p3_vector_nonsentinel.h is the implementation which does not use sentinels. p5_test.c is the testing program for the non-sentinel version, which can be converted to test the sentinel version instead by defining the macro USE_SENTINELS at the top of the file. This can also be done during compilation by using the following:

1. On GCC:

   ```
   gcc -DUSE_SENTINELS p3_test.c
   ```

2. On Clang:

   ```
   clang -DUSE_SENTINELS p3_test.c
   ```

3. On MSVC:

   ```
   CL /DUSE_SENTINELS p3_test.c
   ```

All source code conforms to ANSI C89.

Question4.

Define an ADT for Set.

Write C data representation and functions for the operations on the Set in a Header file, with array as the base data structure.

Write a menu-driven main program in a separate file for testing the different operations and include the above header file.

Solution Approach:

We will implement sets to represent mathematical (finite) sets as closely as possible, with the following properties:

1. Sets are collections of similar objects called elements (with no particular order) and no two elements in a set are same.

2. (Finite) sets have a cardinality which is basically the number of elements in the set.

3. We can check for membership (containment) of elements within a set.

4. Union, intersection and subtraction can be done on two (or more than two) sets.

5. We can also check for subsets (whether one set is a subset/proper subset of the other) and whether two sets are disjoint; we can also check for null sets.

In addition to the above, we will also be adding support for addition and removal of elements in our sets. We will also need support for iterating/traversing over the elements in the set (the order of the elements is irrelevant).

Since the problem requires we use an array as a backing data structure, we will use array lists/vectors developed in problem 3 of this assignment for the implementation.

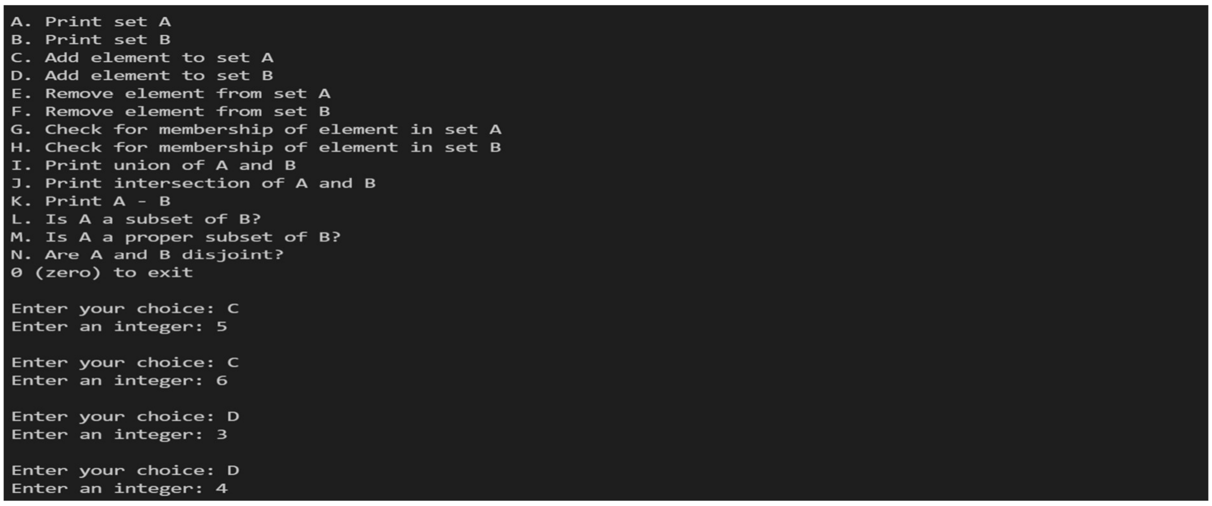
Structured Pseudocode:

Assume the following operations are defined for lists (which are array lists, we use the following as general so alternative implementations can also be easily switched in):

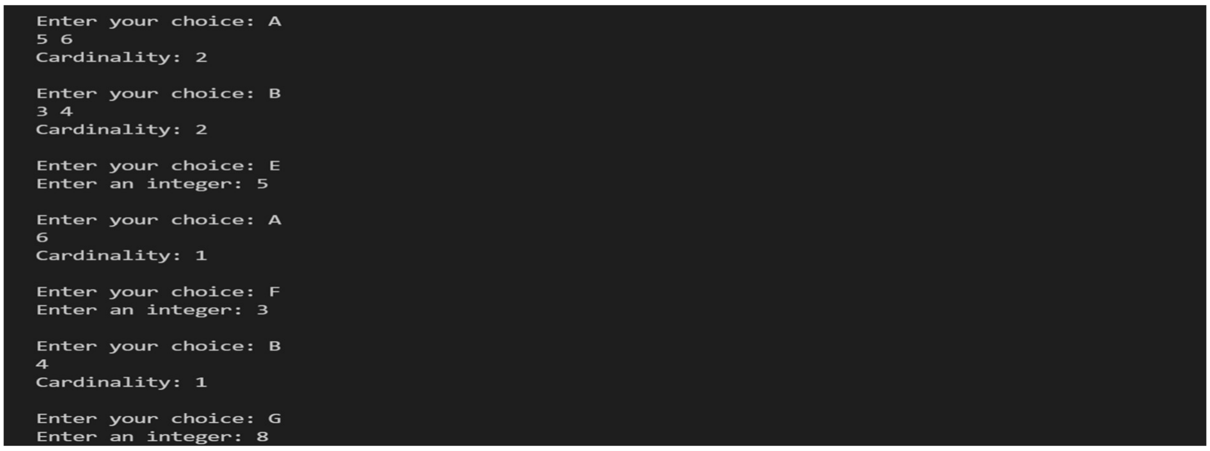
```
list_length(list) to obtain the length of list

list_search(list, element) to search for element in list, true if present and false otherwise
list_insert(list, element) to add an element in the list (location is irrelevant)

list_remove(list, element) to remove an element in the list (location is irrelevant)

And also the list object must have the property that we can traverse/iterate over the
elements in the list. This traversal operation can be used as-is for sets.

set_cardinality(S) → list_length(S) //number of elements in set S the list length

set_is_null(S) → set_cardinality(S) == 0 //Null sets have cardinality 0

set_is_member(S, x) → list_search(S, x) //Search in set S for checking memebership of x in S
set_add_member(S, x) //Add a unique element x into the set

    if set_is_member(S, x) is true: return with status indicating x was already present
    list_insert(S, x)

    return with status indicating x was added


set_remove_member(S, x) → list_remove(S, x) //Remove from set = Remove from list
set_union(A, B) //Return union of sets A and B

    Create C as a new empty set/list

    for each x in A: set_add_member(C, x)

    for each x in B: set_add_member(C, x) //set_add_member takes care of common
    elements

    return C


set_intersection(A, B) //Return intersection of A and B

    Create C as a new empty set/list

    for each x in A

        if set_is_member(B, x) is true: set_add_member(C, x) //Add iff present in
    both A and B


        return C
```

```
set_subtraction(A, B) //Subtract B from A and return result
    Create C as a new empty set/list
    for each x in A
        if set_is_member(B, x) is false: set_add_member(C, x) //Add iff present in A but not in B

    return C

set_is_subset(A, B) //Check if A is a subset of B
    for each x in A
        if set_is_member(B, x) is false: return false //Not a subset if at least one element of A is not present in B

    return true //All elements of A are in B

set_is_proper_subset(A, B) → set_is_subset(A, B) and set_cardinality(A) < set_cardinality(B)
/*A is a proper subset of B if A is a subset of B and if there is at least one element of B not in A; or in case of finite sets, cardinality of B is more than that of A*/
set_is_disjoint(A, B) //Check if A and B have no elements common
    for each x in A
        if set_is_member(B, x) is true: return false //Not disjoint if at least one element of A is common to B

    return true //All elements of A are not in B
```

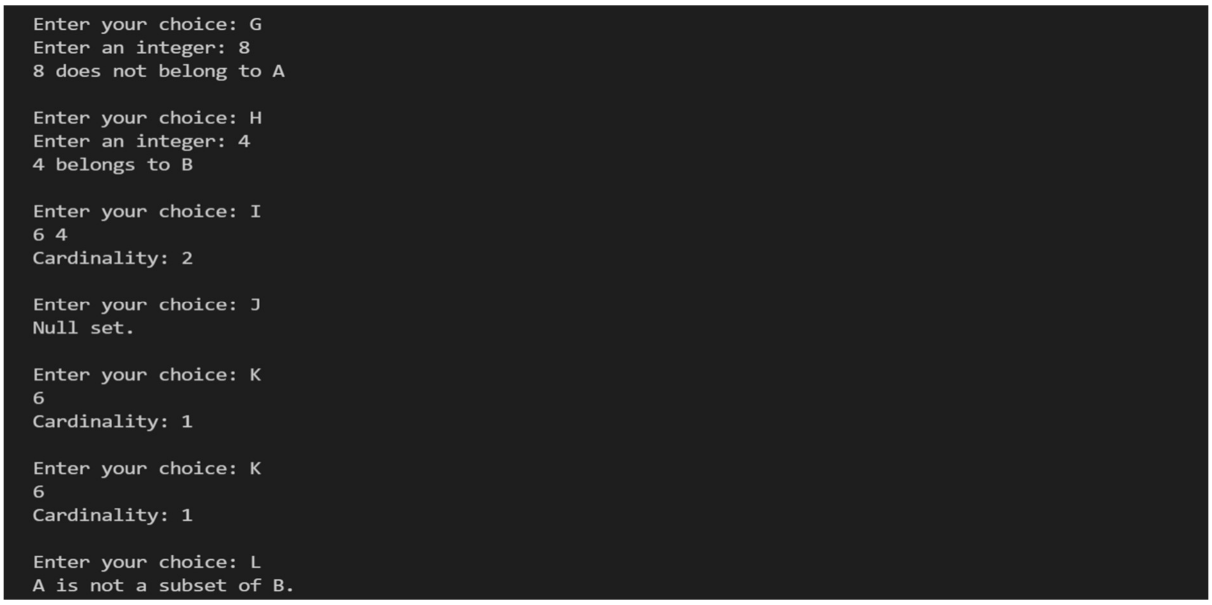
Results:



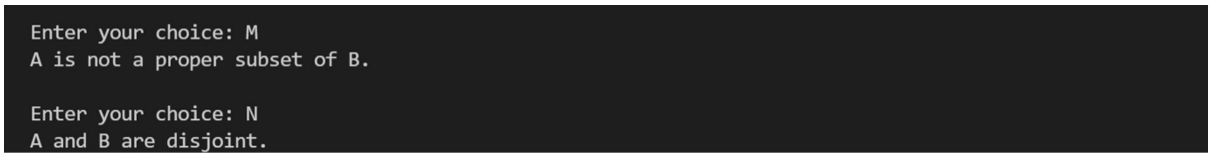
```
A. Print set A
B. Print set B
C. Add element to set A
D. Add element to set B
E. Remove element from set A
F. Remove element from set B
G. Check for membership of element in set A
H. Check for membership of element in set B
I. Print union of A and B
J. Print intersection of A and B
K. Print A - B
L. Is A a subset of B?
M. Is A a proper subset of B?
N. Are A and B disjoint?
0 (zero) to exit

Enter your choice: C
Enter an integer: 5

Enter your choice: C
Enter an integer: 6

Enter your choice: D
Enter an integer: 3

Enter your choice: D
Enter an integer: 4
```



```
Enter your choice: A
5 6
Cardinality: 2

Enter your choice: B
3 4
Cardinality: 2

Enter your choice: E
Enter an integer: 5

Enter your choice: A
6
Cardinality: 1

Enter your choice: F
Enter an integer: 3

Enter your choice: B
4
Cardinality: 1

Enter your choice: G
Enter an integer: 8
```



```
Enter your choice: G
Enter an integer: 8
8 does not belong to A

Enter your choice: H
Enter an integer: 4
4 belongs to B

Enter your choice: I
6 4
Cardinality: 2

Enter your choice: J
Null set.

Enter your choice: K
6
Cardinality: 1

Enter your choice: K
6
Cardinality: 1

Enter your choice: L
A is not a subset of B.
```



```
Enter your choice: M
A is not a proper subset of B.

Enter your choice: N
A and B are disjoint.
```

Source Code:

p3_vector_nonsentinel.h was borrowed from problem 3 of this assignment to provide the backing data structure for the set. This header file is used by p4_set.h which contains the structure definition and operations for implementing sets as described above. Lastly p4_test.c is a menu-driven program to test the header files. All source code conforms to ANSI C89.

Question5.

Define an ADT for String.

Write C data representation and functions for the operations on the String in a Header file, with array as the base data structure, without using any inbuilt function in C.

Write a menu-driven main program in a separate file for testing the different operations and include the above header file.

Solution Approach:

We can define strings as sequences of characters, where each character is accessible by its position (an integer); "Hello" may be said to be an example of a string where 'H' has position 1, 'e' has position 2, position 3 and 4 have 'l' (multiple positions may have common characters) and so on. Since strings are very similar (almost congruent) to arrays of characters in this regard, we will implement strings as arrays of characters, where each character has the property that any two characters have some comparison based ordering defined ('A' may be defined to be less than 'B', etc.). We will also use the sentinel value NIL as a character to indicate the end of a string (NIL is present immediately after the last character of the string). This description is very much identical to how strings are implemented in the standard C library available to programmers, and so we will also define a similar suite of operations paralleling the standard C library:

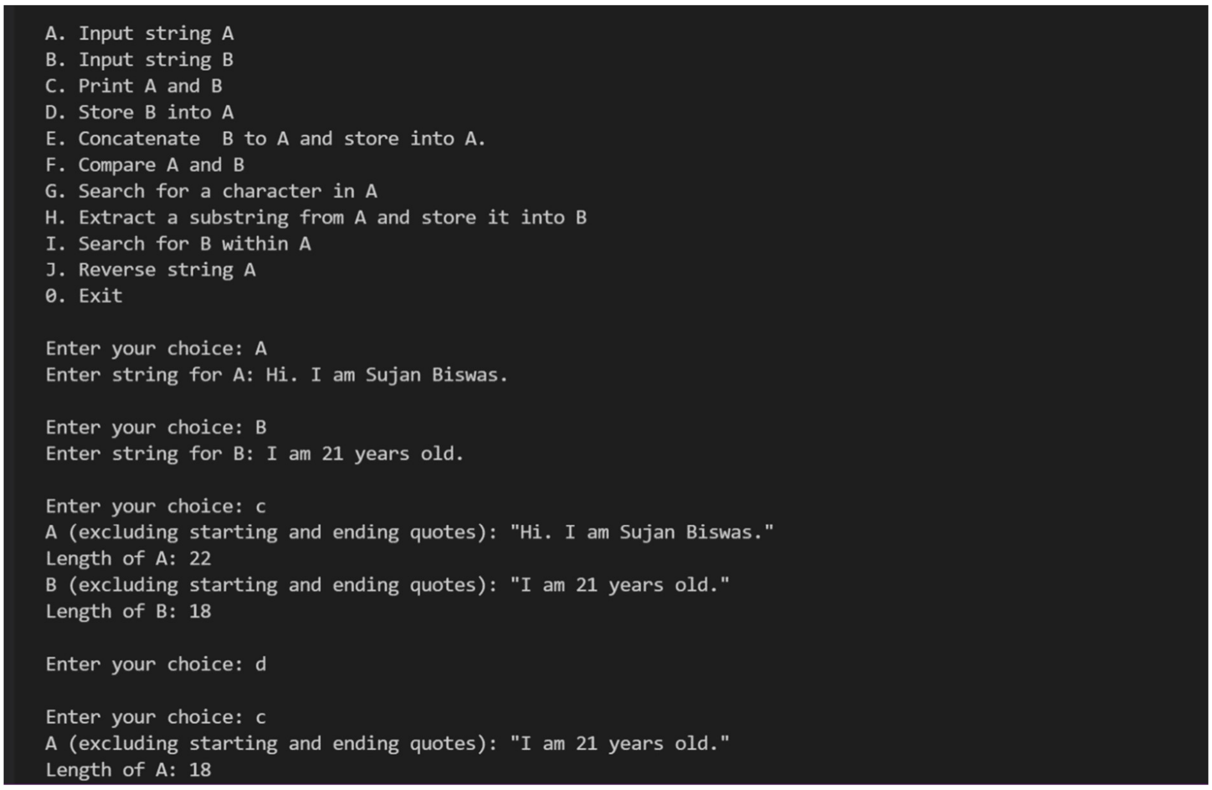
1. Finding the length/number of characters of a string (excluding NIL) (paralleling strlen)
2. Copying a string from a source array to a destination array (paralleling strcpy)
3. Adding a string immediately after the end of a second string (called concatenation operation, paralleling strcat)
4. Comparison of two strings in lexicographical order as defined by character comparisons (paralleling strcmp). We define equality of strings as strings having same length and same characters at same positions. To find whether a string A is less than string B, we use the following method which is the same as those found in dictionaries: We start from the starting position of both A and B. If at such a position the character at A is less than (comes before in ordering) than the character at B in the same position, then A is less than B. (Example, "axe" will come before "ball" inside a dictionary.) While advancing, if we reach the end of A before we reach the end of B (A has less number of characters than B) then A is less than B. (Example, "as" comes before "assimilate".)
5. Searching for a character from the starting end of a string (paralleling strchr)
6. Obtaining a substring (part of a string) from a bigger string (paralleling strncpy)
7. Searching for a substring within a string from the starting end; for example "ll" is found in "Hello" from position 3 (paralleling strstr)
8. Reversing the character sequence within the string (many earlier C++ implementations had strrev for this; although now it is absent)

Name: Sujan Biswas **Subject: Data Structure and Algorithms Lab**
Class Roll No: 302010501003 **Assignment Set-2**

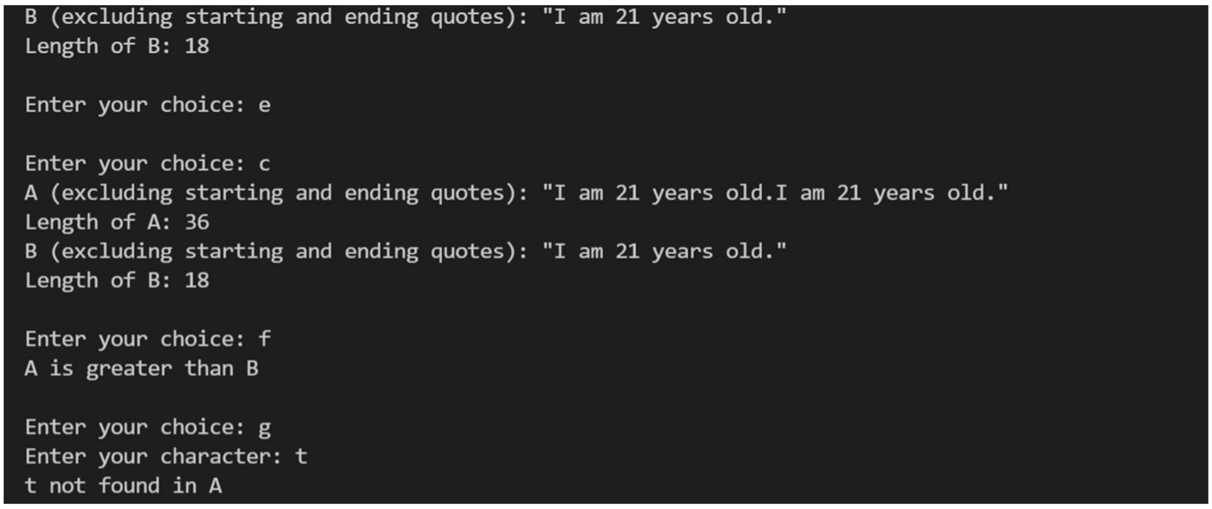
Structured Pseudocode:

```
//Assume all indexes start from 1. Assume each character can be compared as in integers.
string_length(A) → length //Return the length of the string

        i = 1

        while A[i] is not equal to NIL: i = i + 1

        return i – 1 //number of (valid) characters is the number of characters excluding NIL

string_copy(D, S)  //Copy S to D

        i = 1

        while S[i] is not equal to NIL {

                D[i] = S[i] //copy valid character

                i = i + 1


        D[i] = NIL //Indicate valid end of string in destination


string_concatenation(D, S)  //Copy S to the end of D

        i = 1

        while D[i] is not equal to NIL: i = i + 1 //First go to the end of D

        j = 1

        while S[j] is not equal to NIL

                D[i] = S[j] //Copy S to D starting from the end of D

                i = i + 1, j = j + 1


        D[i] = NIL


string_compare(A, B) //Check whether A comes before B, is equal to B or after B

        i = 1

        while A[i] is not equal to NIL and B[i] is not equal to NIL

                if A[i] < B[i]: return A is less than B

                elif A[i] > B[i]: return A is greater than B

                i = i + 1
        if A[i] is not equal to B[i]   //One of the strings is smaller, they did not reach NIL at
        the same position.
            if A[i] is NIL: return A is less than B //because A finished before B
            else return A is greater than B //because B finished earlier than A

        else return A is equal to B //Same characters at same positions and same length

    string_char_search(A, ch) //Search for ch inside A and return the index
        i = 1
        while A[i] is not equal to NIL {
            if A[i] equals ch: return I
            i = i + 1

        return NIL //As we did not find the character.

    sub_string(A, startindex, endindex) //Index parameters inclusive (assumed valid), return
substring
        Create string (array) B with space for (endindex – startindex + 2) characters
        for i = startindex to endindex inclusive: B[i – startindex + 1] = A[i]
        B[endindex - startindex + 2] = NIL //Properly terminate the string
        return B

    /*Assume A and B are valid in the sense that neither A nor B are empty strings and length of
    B <= length of A. This searches for B within A and returns starting index where B begins
    within A.*/
    string_sub_search(A, B)
        i = 1, len = string_length(B)
        while A[i] is not equal to NIL
            if A[i] equals B[1] { //We found a possible beginning of substring.
        /*We could call sub_string but that would require auxiliary space. Instead we check
        within the given parameter memory space.*/
                flag = false
                for j = 2 to len inclusive //Check if this is a valid substring, compare all len characters
                    if B[j] not equal to A[i + j – 1]
                        flag = true break out of the immediate above for

                if not flag: return i //We found a valid substring as we never broke out of the for

            i = i + 1

        return NIL //As we did not find the string

    string_reverse(S) //Reverse string S in-place.
        len = string_length(S)
        for i = 1 to floor(len/2) inclusive: swap(S[i], S[len – i + 1]) //Swap a character with its corresponding character from the other end
```

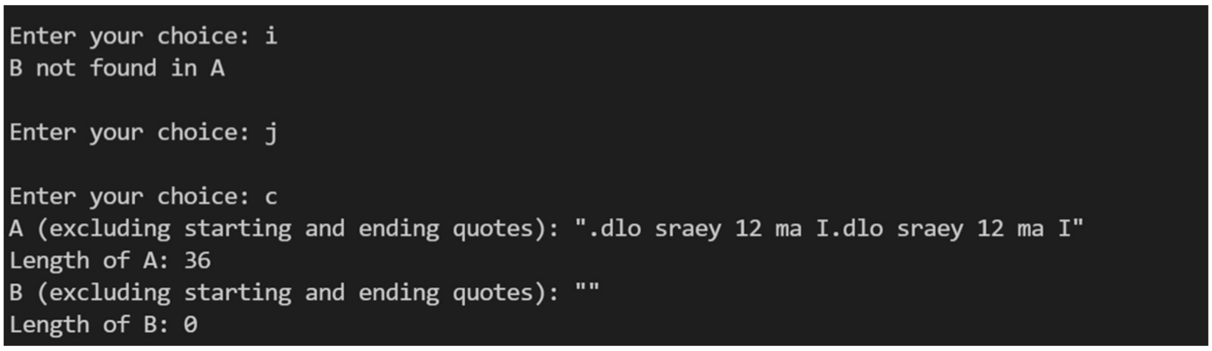
Outputs:



```
A. Input string A
B. Input string B
C. Print A and B
D. Store B into A
E. Concatenate  B to A and store into A.
F. Compare A and B
G. Search for a character in A
H. Extract a substring from A and store it into B
I. Search for B within A
J. Reverse string A
0. Exit

Enter your choice: A
Enter string for A: Hi. I am Sujan Biswas.

Enter your choice: B
Enter string for B: I am 21 years old.

Enter your choice: c
A (excluding starting and ending quotes): "Hi. I am Sujan Biswas."
Length of A: 22
B (excluding starting and ending quotes): "I am 21 years old."
Length of B: 18

Enter your choice: d

Enter your choice: c
A (excluding starting and ending quotes): "I am 21 years old."
Length of A: 18
```



```
B (excluding starting and ending quotes): "I am 21 years old."
Length of B: 18

Enter your choice: e

Enter your choice: c
A (excluding starting and ending quotes): "I am 21 years old.I am 21 years old."
Length of A: 36
B (excluding starting and ending quotes): "I am 21 years old."
Length of B: 18

Enter your choice: f
A is greater than B

Enter your choice: g
Enter your character: t
t not found in A
```



```
Enter your choice: i
B not found in A

Enter your choice: j

Enter your choice: c
A (excluding starting and ending quotes): ".dlo sraey 12 ma I.dlo sraey 12 ma I"
Length of A: 36
B (excluding starting and ending quotes): ""
Length of B: 0
```

Source Code:

p5_strings.h is the implementation for the operations as described above; p5_test.c is a testing program for the same. Both conform to ANSI C89.

__Question6.__

Given a large single dimensional array of integers, write functions for sliding window filter with maximum, minimum, median, and average to generate an output array. The window size should be an odd integer like 3, 5 or 7. Explain what you will do with the boundary values.

Solution Approach:

A window (as required by the given problem) over an array is nothing but a subarray of the array, with these special constraints:

1. The window has an odd number of elements greater than 1 (so that it has a distinct middle element)

2. The middle element of the window is mapped to one of the elements in the parent array. For example over the array , a 3-element window over the 4th element (13) would be . An important boundary condition arises when there are no elements which exist to fill the window, for example when choosing a window over the first or last element. In that case, we can fill empty spaces in the window by 0's. For example, in the given array a 5-element window over the 1st element (4) would

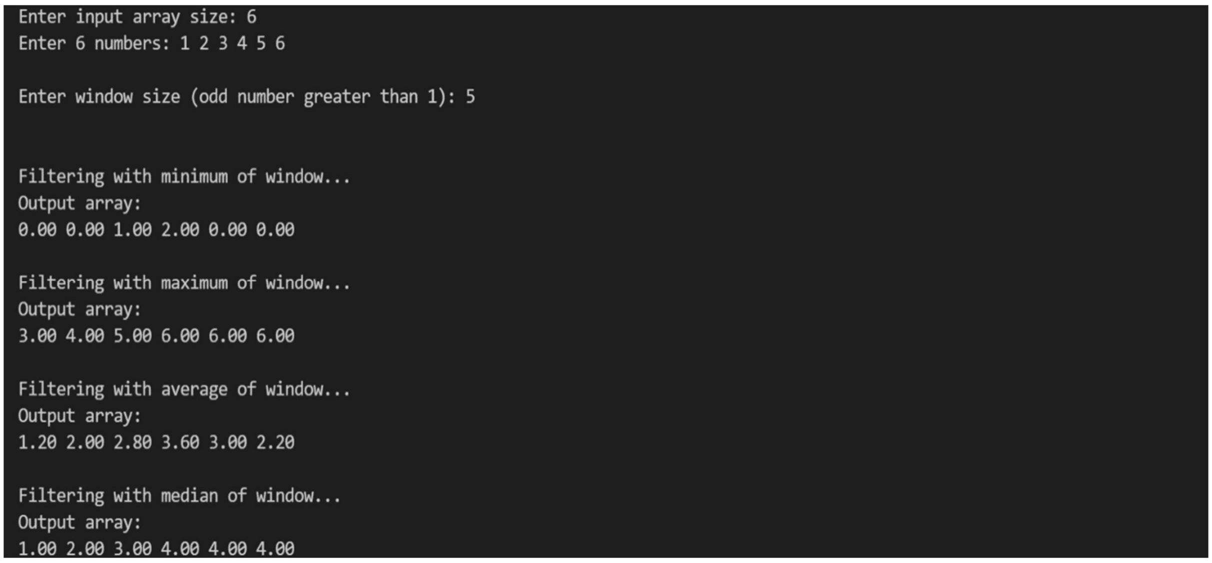
be . Since we can choose any position in the given input array to place the window, this window is called a sliding window (to visualise, think sliding vernier scales on vernier callipers).

From this sliding window, another array of same size as the original parent array can be generated by a reduction operation which generates a single element out of the elements in the window, which is then mapped back to the same position to which the window was mapped to. For example, in the description given previously, if our reduction operation is the average of the elements of the window, then the 4th element of the final array, considering a 3-element window, would be the average of = 5.667. Moving the sliding window over an input array and a reduction operation to generate an output array is called sliding window filtering, sliding window used for this operation is called a sliding window filter.

Structured Pseudocode:

```
/*Assume 1-based indexes*/

get_window(A, position, win_size) /*Get a window array over A, of size win_size (odd number), mapped over position*/

    Create an array W of size win_size

    for i = 1 to win_size inclusive

        pos = position – floor(win_size / 2) + i – 1 //Calculate position from where to get element from A

        if pos < 1 or pos > length of A: W[i] = 0 //Put 0 in window if out of bounds for A

        else W[i] = A[pos]


    return W


Assume reduce(W) exists to reduce a window to a single element

filter(A, win_size) //Output an array of same length as A with sliding window filter of size win_size

    Create an array B of same length as A

    for i = 1 to length of A inclusive: B[i] = reduce(get_window(A, i, win_size))

    return B
```

Results:



```
Enter input array size: 6
Enter 6 numbers: 1 2 3 4 5 6

Enter window size (odd number greater than 1): 5


Filtering with minimum of window...
Output array:
0.00 0.00 1.00 2.00 0.00 0.00

Filtering with maximum of window...
Output array:
3.00 4.00 5.00 6.00 6.00 6.00

Filtering with average of window...
Output array:
1.20 2.00 2.80 3.60 3.00 2.20

Filtering with median of window...
Output array:
1.00 2.00 3.00 4.00 4.00 4.00
```

Source Code:

p6_swf.c demonstrates the implementation of the above described algorithms in ANSI C89.

Question7.

Take an arbitrary Matrix of positive integers, say, 128 X 128. Also take integer matrices of size 3 X 3 and 5 X 5. Find out an output matrix of size 128 X 128 by multiplying the small matrix with the corresponding submatrix of the large matrix with the centre of the small matrix placed at the individual positions within the large matrix. Explain how you will handle the boundary values.

Solution Approach:

The given problem is fairly straightforward; however erroneous cases arise when a submatrix cannot possibly be obtained (such as when obtaining a 3 X 3 submatrix centered at position 1, 1). We choose to fill the missing elements with zeroes.

We also need to consider how to put the generated “small” matrix (of size 3 X 3 or 5 X 5) back to an output matrix of same order as the input matrix. Notice we need to generate a single value out of our “small” matrix considered. In the given example we will consider the minimum, maximum and median of all the values in the “small” matrix, which will be put into the output matrix at the position from which the original “small” matrix was centred upon.

(We note that for the problem to work, the “small” matrix must be a square matrix of odd order like 3 or 5.)

Structured Pseudocode:

Assume matrix elements can be accessed as M[i, j] with i, j being 1-based positions of the elements.

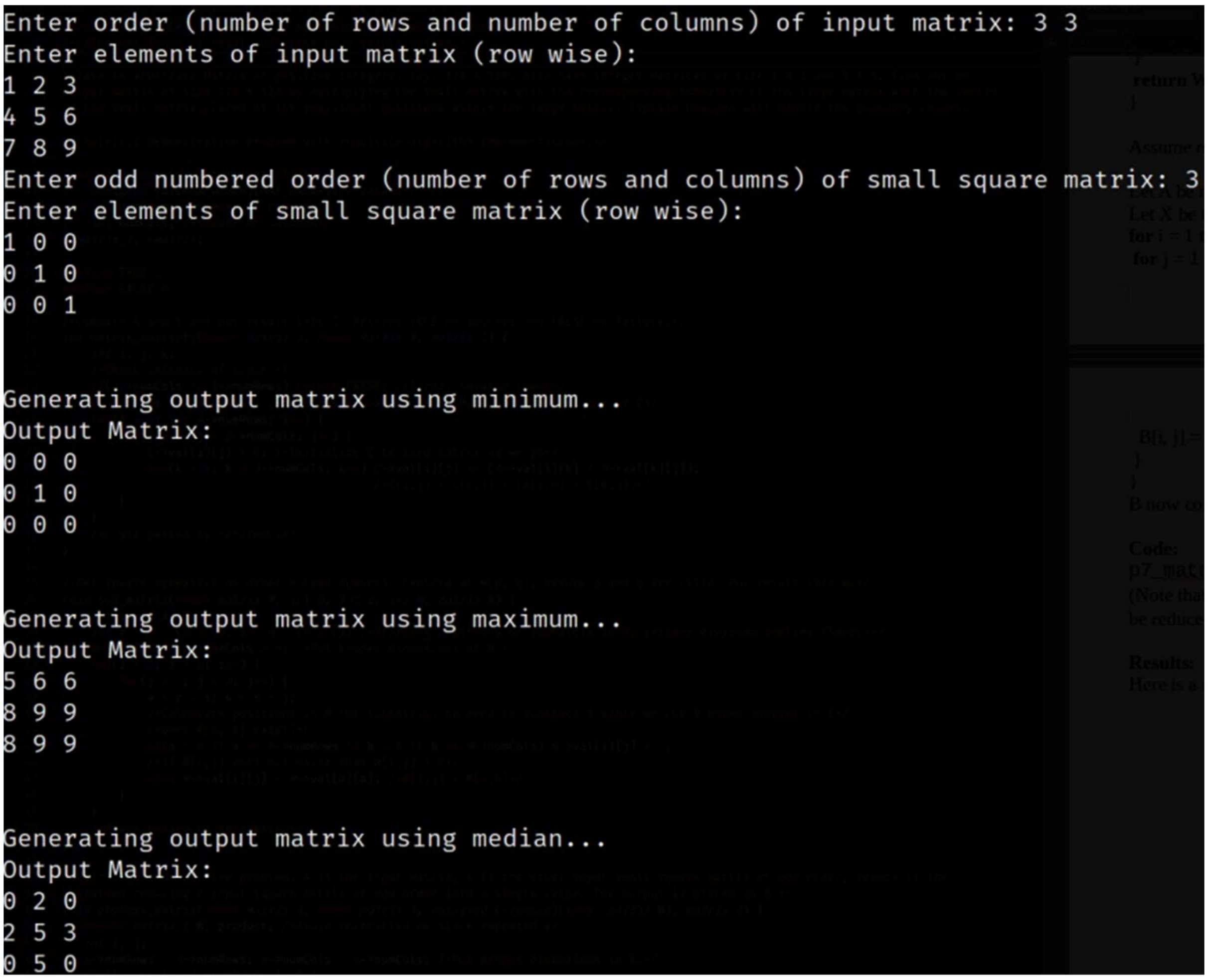
```
matrix_multiply(A, B) //Compute A*B
    if number of columns in A != number of rows in B: error //Invalid input in this case
```

```
        Create a zero matrix C of order (number of rows in A, number of columns in B)
        for i = 1 to number of rows in A inclusive
            for j = 1 to number of columns in B inclusive
                for k = 1 to number of columns in A inclusive: C[i, j] = C[i, j] + (A[i, k]
            * B[k, j])

        return C
```

```
    sub_matrix(M, p, q, n) /*Get square submatrix of order n (odd number), centred at M[p, q],
assume p and q are valid.*/
        r = p – floor(n / 2), s = q – floor(n / 2) //Starting positions of submatrix in M
        Create a square matrix W of order n
        for i = 1 to n inclusive
            for j = 1 to n inclusive
                a = r + i – 1, b = s + j – 1 //Calculate the positions in M for submatrix
                if M[a, b] does not exist: W[i, j] = 0
                else W[i, j] = M[a, b]

        return W
```

```
    Assume reduce(W) exists to reduce matrix W to a single value.
    Let A be the input matrix (already with values) and B be the output matrix of same
dimensions.
    Let X be the other smaller input square matrix of odd order.
    for i = 1 to number of rows in A inclusive
        for j = 1 to number of rows in B inclusive
            B[i, j] = reduce(matrix_multiply(X, sub_matrix(A, i, j, number of rows in X)))
    B now contains requisite output.
```

Outputs:



```
Enter order (number of rows and number of columns) of input matrix: 3 3
Enter elements of input matrix (row wise):
1 2 3
4 5 6
7 8 9
Enter odd numbered order (number of rows and columns) of small square matrix: 3
Enter elements of small square matrix (row wise):
1 0 0
0 1 0
0 0 1


Generating output matrix using minimum...
Output Matrix:
0 0 0
0 1 0
0 0 0


Generating output matrix using maximum...
Output Matrix:
5 6 6
8 9 9
8 9 9


Generating output matrix using median...
Output Matrix:
0 2 0
2 5 3
0 5 0
```

Source Code:

p7_matrix.c is a program containing the algorithms described above implemented in ANSI C89. (Note that the value of the constant MAX_ORDER defined at the top of the given C source file may need to be reduced for the program to run without segmentation/address boundary errors).

<u>**Question8.**</u>

Find whether an array is sorted or not, and the sorting order

Solution Approach:

We can use the following steps to identify the “sortedness” of an array (we assume elements of the array can be compared)

1. An array of length 0, 1 and 2 is sorted by default.

2. For an input array A of length >= 2, for each possible consecutive pair A[i], A[i+1] we can find the order in which they are arranged (ascending, descending or equal).

3. An array is sorted if all such consecutive pairs are either ascending or equal, or descending or equal, but not ascending or descending. (Example: is sorted even though some consecutive pairs are equal, but isn't because some pairs are ascending and one pair is descending.)

4. If all consecutive pairs are equal or ascending, the array is in ascending order. If all consecutive pairs are equal or descending, the array is descending order. And obviously if all consecutive pairs are equal, all the elements are same (the array can said to be in both ascending in descending order).

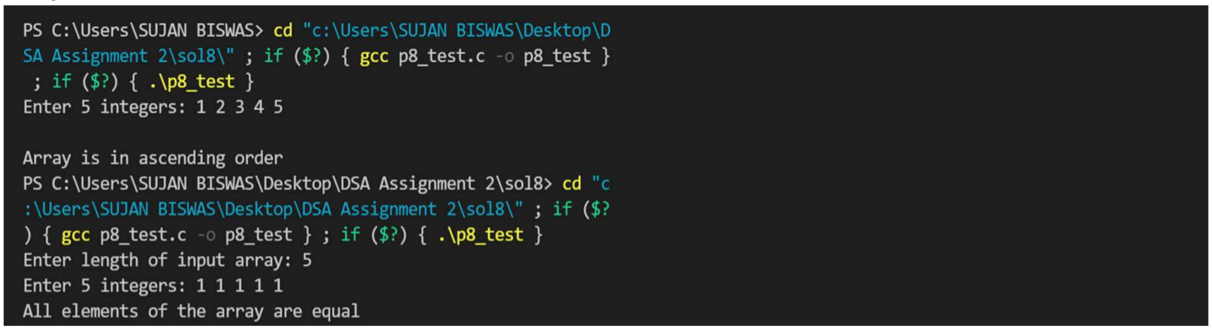
Structured Pseudocode:

We assume a state type sortedness which has the values ASCENDING, DESCENDING, EQUAL and UNSORTED to indicate possible outputs of the procedure.

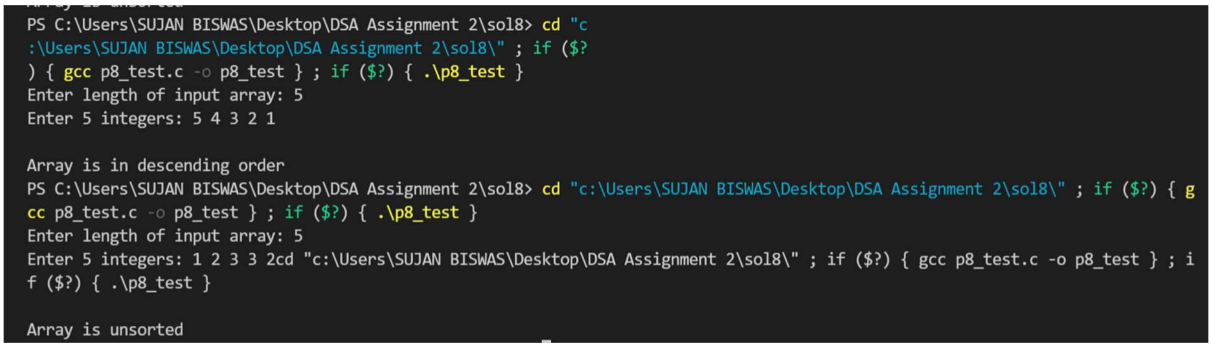
```
check_2state(a, b) //Return a sortedness value for an array
    if a < b: return ASCENDING
    elif a == b: return EQUAL
    else return DESCENDING


check_sorted(A) //Returns state of type sortedness
    if length of A is either 0 or 1: return EQUAL //Because we have nothing to compare to
    overall_state = EQUAL //Initially we assume all elements are equal.
    for i = 1 to (length of A)-1 inclusive  //Assuming 1-based indexes.
        current_state = check_2state(A[i], A[i+1]) //State of current pair
        if overall_state was EQUAL: overall_state = current_state
        //We can change from EQUAL to EQUAL, ASCENDING or DESCENGING
        elif overall_state does not equal current_state and current state is not EQUAL: return UNSORTED
        //If we change from DESCENDING to ASCENDING or vice versa, input is unsorted

    return overall_state
```

Name: Sujan Biswas **Subject: Data Structure and Algorithms Lab**
Class Roll No: 302010501003 **Assignment Set-2**

Outputs:



```
PS C:\Users\SUJAN BISWAS> cd "c:\Users\SUJAN BISWAS\Desktop\D
SA Assignment 2\sol8\" ; if ($?) { gcc p8_test.c -o p8_test }
 ; if ($?) { .\p8_test }
Enter 5 integers: 1 2 3 4 5

Array is in ascending order
PS C:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8> cd "c
:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8\" ; if ($?
) { gcc p8_test.c -o p8_test } ; if ($?) { .\p8_test }
Enter length of input array: 5
Enter 5 integers: 1 1 1 1 1
All elements of the array are equal
```



```
PS C:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8> cd "c
:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8\" ; if ($?
) { gcc p8_test.c -o p8_test } ; if ($?) { .\p8_test }
Enter length of input array: 5
Enter 5 integers: 5 4 3 2 1

Array is in descending order
PS C:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8> cd "c:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8\" ; if ($?) { g
cc p8_test.c -o p8_test } ; if ($?) { .\p8_test }
Enter length of input array: 5
Enter 5 integers: 1 2 3 3 2cd "c:\Users\SUJAN BISWAS\Desktop\DSA Assignment 2\sol8\" ; if ($?) { gcc p8_test.c -o p8_test } ; i
f ($?) { .\p8_test }

Array is unsorted
```

Source Code:

p8_sortedness.h is a C header containing implementations for the algorithms discussed above. p8_test.c is a testing driver program for the header. Both conform to ANSI C89.

<u>Question9.</u>

Given two sorted arrays, write a function to merge the array in the sorting order

Solution Approach:

Imagine the 2 input arrays as two stacked decks of cards with numbers on them, least number at the top. Using this analogy, we can have a “rough” algorithm for merging these two sorted stacks into one sorted stack:

Step 1:

Look at the top of the two stacks, pick the lower numbered card from the two tops of the two input stacks and put this card onto a 3rd card stack (which is initially empty). If one of the input stacks is empty; choose the top of the other stack and put it onto the 3rd stack.

Step 2:

Repeat step 1 until both input stacks are finished.

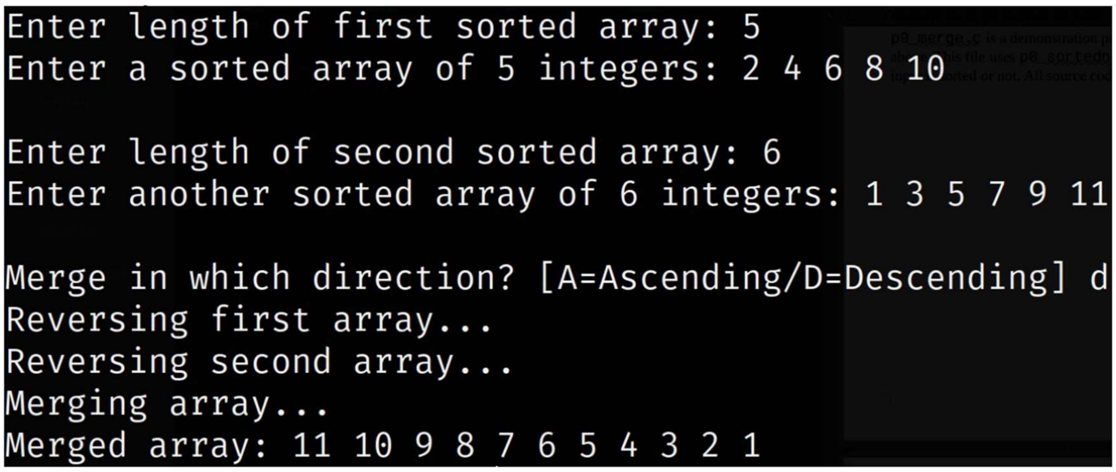
Step 3:

Notice that the 3rd stack is in the reverse order (largest numbered card on top). Reverse this 3rd stack to get final result.

Structured Pseudocode:

```
merge(A, B) //Merge A and B and return result, assuming A and B are in ascending order.
    Create a new array C of length (length of A + length of B)
    i = 1, j = 1, k = 1 //Tops of stacks A, B and C, assume 1-based indexes
    while i <= length of A and j <= length of B //while the input stacks have cards
        if A[i] <= B[j]: C[k] = A[i], i = i + 1 /*Put top of A on C. If input is in descending
    order, reverse this comparison*/
        else: C[k] = B[j], j = j + 1//Put top of B on C
        k = k + 1//Advance top of C

    //Elements might remain in either A or B (not both), and these elements >=
    elements already put on C
    if elements exist on A: append A to C
    elif elements exist on B: append B to C
    //Notice C is in the correct order already.
    return C
```

Results:



```
Enter length of first sorted array: 5
Enter a sorted array of 5 integers: 2 4 6 8 10

Enter length of second sorted array: 6
Enter another sorted array of 6 integers: 1 3 5 7 9 11

Merge in which direction? [A=Ascending/D=Descending] d
Reversing first array...
Reversing second array...
Merging array...
Merged array: 11 10 9 8 7 6 5 4 3 2 1
```

Source Code:

p9_merge.c is a demonstration program containing implementation for the merging algorithm discussed above. This file uses p8_sortedness.h developed in the previous assignment for checking whether the input is sorted or not. All source code conforms to ANSI C89.