Name: Ritabroto Ganguly
Roll: 001910501090 BCSE-II

DSA-Assignment2

1. 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.

Instead of pseudo code, I have used actual C code for this problem with extensive commenting.

polynomial.h

```
#include <stdio.h>
#include <stdlib.h>
#define COMPARE(a,b) a==b ? 0 : (a>b ? 1 : -1)

typedef struct term{//stores the exponent of x and the coeffecient of x for each
term in the polynomial
 int exp;
 float coeff;
}term;
term *t;//an array for storing terms of the polynomial

typedef struct polynomial{//stores the starting and the ending index of a
polynomial in the array t
 int start;
 int end;
}polynomial;
polynomial* p; //an array for storing the starting and ending indices of multiple
polynomials in array t

  int pi = 0;//polynomial array index, initially 0
  int ti = 0;//term array index, initially 0
  int max_t = 0;//maximum terms
  int max_p = 0;//maximum polynomials

  void initiate(){//initiate terms array and polynomials array
    if(max_t>0 && max_p>0){
      printf("Already initiated!\n");
      return;
    }
```

```
    else{
      printf("Enter maximum number of polynomials required: ");
      scanf("%d",&max_p);
      printf("Enter maximum number of terms required: ");
      scanf("%d",&max_t);
      t = (term*)malloc(max_t*sizeof(term));//initialize t
      p = (polynomial*)malloc(max_p*sizeof(polynomial));//initialize p
    }
  }

  polynomial zero(){//return a polynomial p(x) = 0
    polynomial x = {-1,-1};
    return x;
  }

  void insert(polynomial* a){//insert into terms(t) and polynomials(p) arrays
    if(max_t<=0 || max_p<=0){
      printf("Initiate first by calling initiate()\n");
      return;
    }

    int terms;//number of terms in the polynomial
    printf("Enter number of terms for for the %dth polynomial: ",pi);
    scanf("%d",&terms);

    if(terms>(max_t-ti) || pi>=max_p){//if not enough space in terms or polynomials
array(p)
      printf("Not enough space\n");
      return;
    }

    a->start = ti;//starting index of polynomail a in array t
    a->end = (ti+terms-1);//ending index of polynomail a in array t
    p[pi++] = *a;//store in array p

    printf("Input in strictly desecnding order of exponents\n");
    for(int i=0;i<terms;i++){
       int e;float c;
      do{
       printf("Enter power of variable(less than the previous terms): ");
       scanf("%d",&e);

       if(i>0 && e>=t[ti-1].exp)
          printf("Input in strictly desecnding order of exponents!!!\n");
       else
          break;
      }while(1);
```

```
      printf("Enter coefficient: ");
      scanf("%f",&c);
      term x = {e,c};
      t[ti++] = x;//store the polynomial after taking input from user in array t, and
increment ti in the process to point to the index after this polynomial
    }
  }

  polynomial padd(polynomial a,polynomial b){//add two polynomials
    if(max_t<=0 || max_p<=0){
      printf("Initiate first by calling initiate()\n");
      return zero();
    }
    if(pi>=max_p){//if polynomial array(p) is full
      printf("Not enough space\n");
      return zero();
    }

    int tti = ti;//save the initial index of terms array(t)

    if(a.start==-1){//if a is a zero polynomial

      for(int i=a.start;i<=a.end;i++){//copy data one by one from a into terms array(t)
        t[ti].exp = t[i].exp;
        t[ti++].coeff = t[i].coeff;
      }
      //update array p accordingly
      polynomial tp = {tti,(ti-1)};
      p[pi++] =  tp;

      return tp;
    }

    if(b.start==-1){//if b is a zero polynomial

      for(int i=b.start;i<=b.end;i++){//copy data one by one from b into terms array(t)
        t[ti].exp = t[i].exp;
        t[ti++].coeff = t[i].coeff;
      }
      //update array p accordingly
      polynomial tp = {tti,(ti-1)};
      p[pi++] =  tp;

      return tp;
    }

    /********if none of the polynomials are zero polynomials go below*********/
```

```
  int tas = a.start;//save starting index of polynomial a
  int tbs = b.start;//save starting index of polynomial b

  while(tas<=a.end && tbs<=b.end){//while one of the two polynomials is fully read

    if(ti>=max_t){//if terms array(t) is full
      printf("Not enough space\n");
      ti = tti;
      return zero();
    }

    int res = COMPARE(t[tas].exp,t[tbs].exp);//compare exponents of the current
term

    if(res==0){//if the exponents are equal

      term tt = {t[tas].exp,(t[tas].coeff+t[tbs].coeff)};//simply add the coefficients
      t[ti++] = tt;//store the term in array t
      tas++;tbs++;
    }else if(res==-1){//if the exponent of a is lesser than b

      term tt = {t[tbs].exp,(t[tbs].coeff)};//put the coefficient and exponent of b
      t[ti++] = tt;//store the term in array t
      tbs++;
    }else{//if the exponent of a is greater than b

      term tt = {t[tas].exp,(t[tas].coeff)};//put the coefficient and exponent of a
      t[ti++] = tt;//store the term in array t
      tas++;
    }
  }

  while(tas<=a.end){//put the rest of unread terms of a(if left) into terms array(t)
    if(ti>=max_t){
      printf("Not enough space\n");
      ti = tti;
      return zero();
    }
      term tt = {t[tas].exp,(t[tas].coeff)};
      t[ti++] = tt;
      tas++;
  }
  while(tbs<=b.end){//put the rest of unread terms of b(if left) into terms array(t)
    if(ti>=max_t){
      printf("Not enough space\n");
      ti = tti;
      return zero();
    }
```

```
        term tt = {t[tbs].exp,(t[tbs].coeff)};
        t[ti++] = tt;
        tbs++;
    }

    polynomial tp = {tti,(ti-1)};
    p[pi++] =  tp;
    return tp;
  }

  polynomial pmul(polynomial a,polynomial b){//multiply two polynomials
    if(max_t<=0 || max_p<=0){
      printf("Initiate first by calling initiate()\n");
      return zero();
    }
    if(a.start==-1 || b.start==-1)//if a or b is a zero  polynomial
      return zero();
    if(pi>=max_p){//if polynomials array(p) is full
       printf("Not enough space\n");
       return zero();
    }

     int a_length = a.end-a.start+1;//number of terms in a
     int b_length = b.end-b.start+1;//number of terms in b
     term tt[(a_length+b_length)];//temporary terms array
     int tti = 0;//temporary terms array index
     int init_ti = ti;//save the initial terms array index
     int space = max_t-ti;//space left in the original terms array(t)

     for(int i=a.start;i<=a.end;i++){
        for(int j=b.start;j<=b.end;j++){
          term ttt = {(t[i].exp+t[j].exp),(t[i].coeff*t[j].coeff)};//new term by adding
exponents and multiplying coeffs
          //printf("ttt= %.3fxE%d\n",ttt.coeff,ttt.exp);
          int flag = 0;
          for(int k=0;k<tti;k++)//loop through temporary terms array(tt)

            if(ttt.exp == tt[k].exp){//if any term in the tt has same exponent as ttt
              tt[k].coeff += ttt.coeff;//then add the coeffs
              flag = 1;//set flag
              break;
            }

          if(flag==0){//if flag not set
            if(tti>space){//check if original terms array(t) has space for another term
              printf("Not enough space\n");
              ti = init_ti;
              return zero();
```

```
        }

        tt[tti++] = ttt;//if has space then add term into temporary array(tt)
      }

      /*for(int k=0;k<tti;k++){
        printf("tt[%d]= %.3fxE%d\n",k,tt[k].coeff,tt[k].exp);
      }*/
    }
  }

  int c = 0;
  while(c<tti){//loop through tt and store in t

    if(ti>=max_t){//check if t is full
      printf("Not enough space\n");
      ti = init_ti;
      return zero();
    }

    int max_i = -1;
    int max = -1;
    for(int i=0;i<tti;i++)
      if(max<tt[i].exp){//find the max exponent term in tt
        max = tt[i].exp;
        max_i = i;
      }

    t[ti++] = tt[max_i];//if t not full then copy max exponent term from tt to t
    c++;
    tt[max_i].exp = -1;
  }

  polynomial tp = {init_ti,(ti-1)};
  p[pi++] = tp;//update p
  return tp;
}

polynomial const_mul(float x,polynomial a){//multiply polynomial with a constant
  if(max_t<=0 || max_p<=0){
    printf("Initiate first by calling initiate()\n");
    return zero();
  }
  if(a.start==-1)//if a is a zero  polynomial
    return zero();

  int length = a.end-a.start+1;//number of terms in a
```

```
    if(length>(max_t-ti) || pi>=max_p){//check space in polynomial array(p) and terms
array(t)
      printf("Not enough space\n");
      return zero();
    }

    //update array p
    polynomial y = {ti,(ti+length-1)};
    p[pi++] = y;

    for(int i=a.start;i<=a.end;i++){
       term tt = {t[i].exp,(x*t[i].coeff)};//multiply each term with the given constant
       t[ti++] = tt;//put into t
    }

    return y;
  }

  void print(polynomial a){//print the polynomial as a function of x
    if(a.start==-1){//if a is a zero  polynomial
      printf("0\n");
      return;
    }
    for(int i=a.start;i<=a.end;i++){
       if(i==a.end)
         printf("%.3fxE%d\n",t[i].coeff,t[i].exp);
       else
         printf("%.3fxE%d + ",t[i].coeff,t[i].exp);
    }
  }
```

poly.c

```
#include "polynomial.h"

int main(void){
 int in;
 initiate();
 do{
   printf("Enter a value (0=cancel, 1=add, 2=multiply, 3=multiply with constant): ");
   scanf("%d",&in);
   switch(in){
     case 1: {polynomial a,b;insert(&a);insert(&b);print(padd(a,b));break;}
     case 2: {polynomial a,b;insert(&a);insert(&b);print(pmul(a,b));break;}
     case 3: {polynomial a;float f = 0;
            insert(&a);
            printf("Enter a constant for multiplication: ");
```

```
            scanf("%f",&f);
            print(const_mul(f,a));
            break;}
     default: return 0;
   }
 }while(in!=0);
 return 0;
}
```

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 *n*

$P(x)=\sum a_i x^i$ for all *x* in the domain (usually real numbers, in this case each a_i is a real number $i=0$

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 a_j 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 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 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.

2. Define an ADT for Sparse Matrix.
Write C data structure representation and functions for the operations on the Sparse Matrix 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.

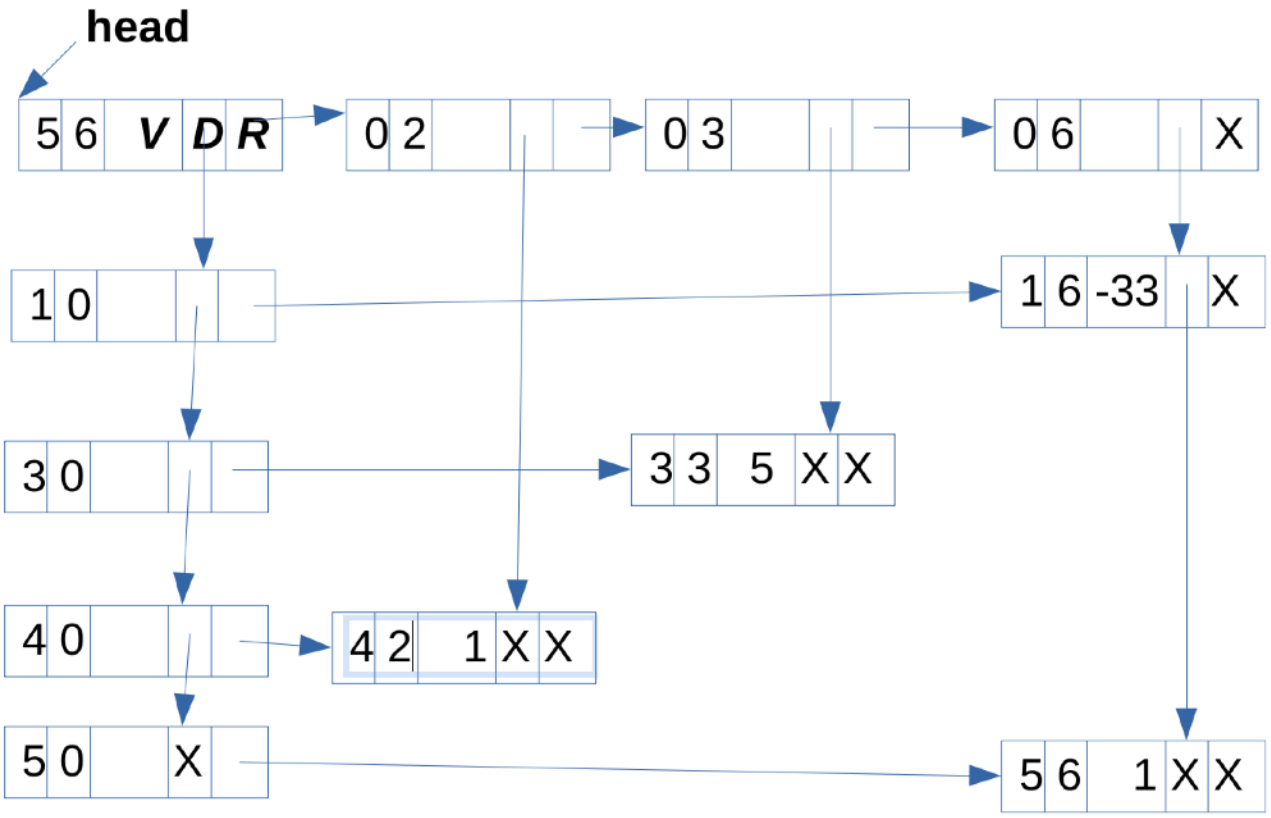
Solution Approach:

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.

0 0 0 0 0 –33 000000 005000 010000 000001

Original/source 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.

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.)

3. 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 i^{th} element
4. Store a new value (overwrite) at i^{th} position
5. Insert a new element at i^{th} 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 i^{th} 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 1^{st} position, and then repeat the process considering the sublist formed from the 2^{nd} 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.

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] }

}

4. 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 and intersection can be done on two (or more than two) 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 an array of doubly linked lists.

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_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_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 {

print_set(X){//prints the set}

5. 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)

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. Assigning one string to another

6. Print the string

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_assign(A,B){//Assign B to A

while i is less than length of B{

A[i] = B[i]

}}

string_print(X){//print the content of string X}

6. 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.

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

2. The first and last elements of the window should always be an element within the original larger array.

We simply keep sliding the window by changing the starting and ending index of within the original array and do the following calculations:

1. Find the maximum value
2. Find the minimum value
3. Find the median value
4. Find the average value
5. Print the found values

Instead of pseudo code, I have used actual C code for this problem with extensive commenting.

```
#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

#define type(T) _Generic( (T), char: " %c ", int: "%d ", float: "%.3f ", default: "%s")

//generic printing function for basic data types

#define print(a,size,s)\

  printf("%s",s);\

  for(int i=0;i<size;i++)\

     printf(type(a[i]),a[i]);\

  printf("\n");\


void max(int * a,int size,int *b,int k){//max filter, k is the window size, a is the array, b is the window, size is the size of the array a

  int i = 0,j = 0;

  int max = -1;

  int t = k;

  while(i<t){//find max element in the current window

    if(a[i]>max)
```

```
      max = a[i];
    i++;
    if(i==t){//if whole current window has been traversed
      b[j++] = max;//save max element of each window of size k in b
      if(t+1<=size){//if there is space left to slide in array a
        t++;//slide the window
        i = j;//update index of sub-array/window b
        max = -1;//reset value of max, to start searching for max value in the current window again
      }
    }
  }
}

void min(int * a,int size,int *b,int k){//min filter, k is the window size, a is the array, b is the window, size is the size of the array a
  int i = 0,j = 0;
  int min = INT_MAX;
  int t = k;
  while(i<t){//find min element in the current window
    if(a[i]<min)
      min = a[i];
    i++;
    if(i==t){//if whole current window has been traversed
      b[j++] = min;//save min element of each window of size k in b
      if(t+1<=size){//if there is space left to slide in array a
        t++;//slide the window
        i = j;//update index of sub-array/window b
```

```
        min = INT_MAX;//reset value of min, to start searching for min value in the current window again
      }
    }
  }
}

/*function for using in qsort()*/
int cmp (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

void median(int *a,int size,int *b,int k){//median filter, k is the window size, a is the array, b is the window, size is the size of the array a
  /* sort the window elements for each window position in a and store the middle/ median values in b */
  for(int i=0;i<(size-k+1);i++){
     int c[k];int z = 0;
     for(int j=i;j<(i+k);j++)
        c[z++] = a[j];
     qsort(c, k, sizeof(int), cmp);
     //print(c,k);
     b[i] = c[(k/2)];
  }
}

void avg(int *a,int size,float *b,int k){//average filter, k is the window size, a is the array, b is the window, size is the size of the array a
  /* find average of the values for each window position in a and store in b */
```

```
  for(int i=0;i<(size-k+1);i++){
     float sum = 0;
     for(int j=i;j<(i+k);j++)
        sum += a[j];
     b[i] = sum/k;
  }
}

int main()
{
 int a[] = {4, 5, 1, 13, 3, 25, 27, 18, 10, 3, 4, 9};
 int size = sizeof(a)/sizeof(a[0]);
 print(a,size,"original: ")
 //printf("size=%d\n",size);
 int k = 3;
 int b[size-k+1];
 float c[size-k+1];
 max(a,size,b,k);
 print(b,size-k+1,"max: ");
 min(a,size,b,k);
 print(b,size-k+1,"min: ");
 median(a,size,b,k);
 print(b,size-k+1,"median: ");
 avg(a,size,c,k);
 print(c,size-k+1,"average: ");
}
```

7. 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/ window centered at position 1, 1).

1. We have to fill the missing elements(near the boundaries) with zeroes.
2. In the given solution we have used the sum of all the values of each product matrix(formed by multiplying the submatrix/window and the smaller matrix) as each single element in the output matrix

Instead of pseudo code, I have used actual C code for this problem with extensive commenting.

```
#include<stdio.h>

#include<stdlib.h>

#define ledge 5 //using 5 instead of 128 because inputting so many values is unviable

#define sedge 3 //we use size 3 for the smaller matrix, we can easily show with size 5 also

#define size ((ledge)+2*((sedge)/2)) //for calculating the edge size of the padded larger matrix


void print(int sz,int x[][sz],char* const s){

  printf("%s\n",s);

  for(int k=0;k<sz;k++){

     for(int m=0;m<sz;m++)

        printf("%d ",x[k][m]);

    printf("\n");
```

```
  }
}


// utility function for multiplying 2 smaller matrices and returning the total sum of each indiviual products
int mul(int a[][sedge],int b[][sedge]){
  int tot = 0;


  for(int i=0;i<sedge;i++){
     int sum = 0;//stores the sum of the elements of each row of the product matrix of a*b
     for(int j=0;j<sedge;j++){
        sum += a[i][j]*b[j][i];
     }
     tot += sum;//stores the sum of the sums of rows
  }
//  printf("tot=%d\n",tot);
  return tot;//return the total sum of the elements of the product matrix
}


/* main algorithm */
void mmul(int a[][size],int b[][sedge],int c[][ledge]){
  for(int i=0;i<ledge;i++){
     for(int j=0;j<ledge;j++){
        int d[sedge][sedge];
        for(int k=0;k<sedge;k++){
           for(int m=0;m<sedge;m++){
```

```
            d[k][m] = a[i+k][j+m];//save the smaller matrix of required size from the bigger matrix

//              printf("%d ",d[k][m]);

          }

  //        printf("\n");

       }

       c[i][j] = mul(b,d);//fill up each index of the final required matrix with the total sum of the elements of each product matrix.

    }

  }

}


int main()

{

 //printf("%d\n",size);

 int a[size][size];   /*= {0, 0, 0, 0, 0, 0, 0,

                        0, 1, 2, 3, 4, 5, 0,

                        0, 6, 7, 8, 9,10, 0,

                        0,11,12,13,14,15, 0,

                        0,16,17,18,19,20, 0,

                        0,21,22,23,24,25, 0,

                        0, 0, 0, 0, 0, 0, 0};*///sedge/2 size padding around the matrix given to that boundary windows can be filled easily

 /* filling up the larger matrix to look like the matrix above */

 int padding = sedge/2;

 int k = 1;

 for(int i = 0; i < size; i++){

    for(int j = 0; j < size; j++){
```

```
        if(i>=ledge || j>=ledge || i<padding || j<padding)
                a[i][j] = 0;
        else
                a[i][j] = k++;
    }
}
 print(size,a,"Larger matrix:");

 /* smaller matrix */
 int b[sedge][sedge] = {1,2,3,
                        4,5,6,
                        7,8,9};
 int c[ledge][ledge];
 mmul(a,b,c);
 print(ledge,c,"Final matrix:");
}
```

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

Solution Approach:

We check every consecutive element in the array to find out the sorting order of the array

Pseudocode:

```
sort_check(int *a,int size){
  int asc=0,desc=0;
  for(int i=1;i<size;i++){
     if(a[i]>=a[i-1])
```

```
    asc++;//increment asc if next element is greater than the previous element
   else
    desc++;//increment dsc if next element is smaller than the previous element
 }

 if(asc==size-1)//if asc is equal to the number of elements-1
   then print that the array is sorted in ascending order

 if(desc==size-1)//if asc is equal to the number of elements-1
  then print that the array is sorted in ascending order

 otherwise print that the array is in random order
}
```

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

Solution Approach:

We can simply use the algorithm used for merging arrays in merge sort.

1. We compare the elements which are at the front in the two arrays,
2. a. For ascending sorting, the smaller value is stored in the output array first
   b. For descending sorting, the larger value is stored in the output array first
3. Do step 1 again

Pseudocode:

/********* Function can't be used for strings, as comparing strings is quite different from comparing numericals. But, we can easily achieve string array merging by writing a function which compares string character by character (like strcmp() in c)

```
and using that instead of comparison operators for comparing array elements
**********/

int k,i,j;

my_merge(x,y,data,x_size,y_size)
  k=0,i=0,j=0; /* indexes for data,x,y */
  while(i<x_size && j<y_size){
       if(x[i]>y[j]) /* if x[i] is greter put x[i] in data first and vice versa */
         data[k++] = y[j++];
       else
         data[k++] = x[i++];
  }
 /* empty out the rest of x or x if left*/
  while(i<x_size)
     data[k++] = x[i++];
  while(j<y_size)
     data[k++] = y[j++];
```