Data Structures and Algorithms
Lab Report

Titir Adhikary
001810501031

Assignment 1

1. Write a program to compute the factorial of an integer n iteratively and recursively. Check when there is overflow in the result and change the data types for accommodating higher values of inputs.

Solution Approach

1. Program a function **factorial(dtype num)**, which calculates the factorial passed to it as a parameter.
2. Program a function **check_factorial(dtype factorial_value, dtype n)** which checks based on the factorial output and the number passed in as arguments to determine whether the calculated value is a victim of overflow or not.
3. We program an infinite loop that breaks out when the value returned from **check_factorial(...)** gives a **false** value.

Algorithm

Recursive Technique

```
factorial(n)
Begin
  if n == 0 or 1 then
    return 1;
  else
    return n * factorial(n-1);
  endif
End
```

Iterative Technique

```
factorial(n)
Begin
  fact = 1
  for i = 1 to n do
     fact = fact * i
  return fact
End
```

Checking Function

```
check_factorial(factorial_value, n)
Begin
  result_for_check = factorial_value / factorial(n - 1);
  if result_for_check == n then
    return 1;
  else
    return 0;
  endif
End
```

Results

dtype = int : gives wrong result at 13! i.e. upto 12! = 0.479 million we will get correct result.

dtype = long : gives wrong result at 21! i.e. upto 20! = 2.4 million we will get correct result.

Discussions

13! = 6,227,020,800, whereas int (signed) being 32 bit gives a max value of 2^(31) - 1 = 2,147,483,647. As 13! is greater than the max range of signed int, an overflow is generated.

21! = 51,090,942,171,709,440,000, whereas long int (signed) being 64 bit gives a max value of 2^(63) - 1 = 9,223,372,036,854,775,807. As 21!is greater than the max range of signed long int, an overflow is generated.

----

2. Write a program to generate the nth Fibonacci number iteratively and recursively. Check when there is overflow in the result and change the data types for accommodating higher values of inputs. Plot the Fibonacci number vs n graph. Also generate the curve for the number of recursive function calls vs n for the recursive approach.

Solution Approach

1. Program a function **fibonacci(dtype n)**, which calculates the nth Fibonacci number passed to it as a parameter.
2. Store the values generated as per the format (n, value) in a .csv file.
3. We program an infinite loop that goes on doing step 2,and eventually breaks out when the value returned from **fibonacci(n)** gives a value which is less than 0 i.e. negative.

Algorithm

Recursive Technique

```
fibonacci(n)
Begin
  no.of_function_calls += 1;
  if (n == 1) then return a;
  else if (n == 2) then return b;
  else return fibonacci(n - 1) + fibonacci(n - 2);
End
```

Iterative Technique

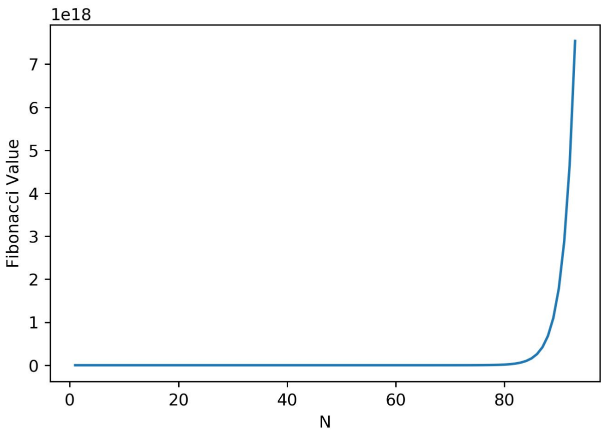
```
fibonacci(n)
Begin
  a, b = 0, 1
  for i in range(0, n) do
      a, b = b, a + b
  return a
End
```

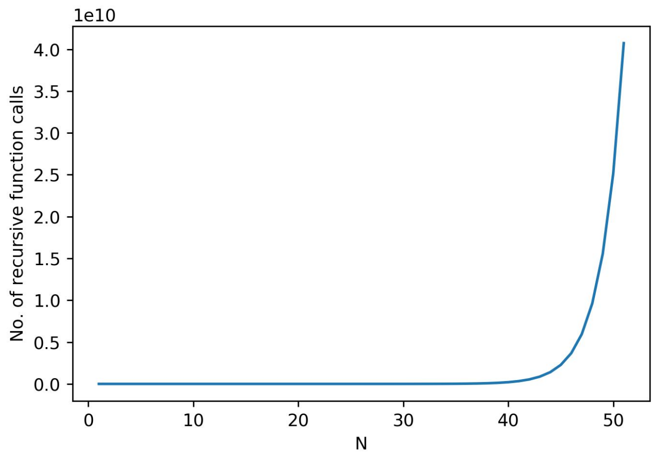
Sample Output

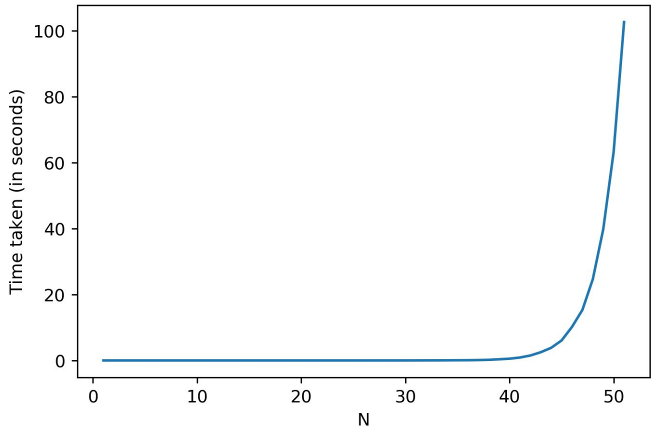
```
Fibonacci number vs n
n, value
1, 0
2, 1
3, 1
4, 2
5, 3
...

Number of Recursive Calls vs n
n, no_of_function_calls, time (in seconds)
1, 1, 0.000002
2, 1, 0.000001
3, 3, 0.000000
4, 5, 0.000001
5, 9, 0.000001
...
```

Graphs







Discussions

We find that there the no. of recursive function calls and the time taken increases exponentially with respect to n. Thus, for big enough N, the call stack may become too big and may lead to Stack Overflow. To make the processing faster, we may resort to other effective algorithms such as Dynamic Programming, which will help decrease the time to calculate nth factorial.

----

3. Write programs for linear search and binary search for searching integers, floating point numbers and words in arrays of respective types.

Algorithm

Linear Search

```
linear_search (list, value)
Begin
  for each item in the list do
      if match item == value
         return the item's location
      end if
   end for
```

```
End
```

Binary Search

```
binary_search(sorted_list, size, value)
Begin
   Set lowerBound = 1
   Set upperBound = n

   while x not found do
      if upperBound < lowerBound
         EXIT: x does not exists.
  set midPoint = lowerBound + (upperBound - lowerBound)/2
      if A[midPoint] < x then
         set lowerBound = midPoint + 1
      if A[midPoint] > x then
         set upperBound = midPoint - 1
      if A[midPoint] = x  then
         EXIT: x found at location midPoint
   end while
End
```

Discussions

In the case of float, an epsilon value EPSILON of the order 0.000001 has to be considered, so as to check for difference due to floating point imprecision.

----

4. Write a program to generate 1,00,000 random integers between 1 and 1,00,000 without repetitions and store them in a file in character mode one number per line. Study and use the functions in C related to random numbers.

Solution Approach

1. We first set a **SEED** to seed the random number generator.
2. Program a function **check_for_repetition(int arr[], int value, int new_size)** which checks whether the **value** passed is already present in the array or not. If the

output is **true,** then the value is discarded and the generator generates another value until the output is **false.**

3. We open a file **random_int.txt** in writing mode, and we continue generating and checking for repetition until we get 1,00,000 unique integers.

Algorithm

```
check_for_repetition(int arr[], int value, int new_size)
Begin
  for i in range(0, new_size) do
    if (arr[i] == value)
      return 1;
    end if
  end for
    return 0;
End

generate random integers
Begin
  while (i < SIZE) do
    new = (rand() % SIZE) + 1;
    if (check_for_repetition(arr, new, i) != 1) then
      arr[i] = new; write arr[i]; i = i + 1;
    else counter += 1;
    end if
  end while
End
```

Sample Output

```
random_int.txt
5895
42224
10010
44344
34827
20602
73202
94761
30651
```

...

Results

SEED : 1505
NO. OF REPETITIONS : 1047024
TIME TAKEN : 103.158609 seconds

SEED : 42
NO. OF REPETITIONS : 1047992
TIME TAKEN : 104.319788 seconds

Discussions

If we change the seed, the number of repetitions and the time taken will also change. Thus, the sequence of numbers generated randomly as well as other properties of the random number generator depends on the SEED value.

----

5. Write a program to generate 1,00,000 random strings of capital letters of length 10 each, without repetitions and store them in a file in character mode one string per line.

Solution Approach

1. We first set a **SEED** to seed the random number generator.
2. Program a function **check_for_repetition(char str_arr[][STRING_LENGTH + 1], char str[STRING_LENGTH + 1], int new_size)** which checks whether the **str** passed is already present in the array of strings or not. If the output is **true,** then the string is discarded and the generator generates another value until the output is **false.**
3. We open a file **random_string.txt** in writing mode, and we continue generating and checking for repetition until we get 1,00,000 unique strings.

Algorithm

```
check_for_repetition(char  str_arr[][STRING_LENGTH  +  1],  char
str[STRING_LENGTH + 1], int new_size)
Begin
  for i in range(0, new_size) do
    if (str_cmp(str_arr[i], value) == 0) then
      return 1;
    end if
  end for
    return 0;
End

generate random strings
Begin
  char str[STRING_LENGTH + 1];
  for j in range(0, STRING_LENGTH) do
    str[j] = (rand() % 26) + 65;
  end for
  str[j] = '\0';
  if (check_for_repetition(str_arr, str, i) != 1) then
    strcpy(str_arr[i], str); write str; i = i + 1;
  else counter += 1;
End
```

Sample Output

```
random_string.txt
TUQMPFUZJF
LMMEVMPCVM
XGJPNQRUOE
RXEYXIDZYO
OSCXDQKQUJ
OFTJSAQISG
BRMSXAJMMZ
...
```

Results

SEED : 1505
NO. OF REPETITIONS : 0

TIME TAKEN : 23.578571 seconds

SEED : 42
NO. OF REPETITIONS : 0
TIME TAKEN : 22.706921 seconds

Discussions

If we change the seed, the time taken will also change. Thus, the sequence of strings generated randomly as well as other properties of the random string generator depends on the SEED value.

----

6. Store the names of your classmates according to roll numbers in a text file one name per line. Write a program to find out from the file, the smallest and largest names and their lengths in number of characters. Write a function to sort the names alphabetically and store in a second file.

Solution Approach

1. We first develop a data structure **Student** that contains all the properties of the student.
2. Program two functions **get_shortest(Student *arr, int n)** and **get_longest(Student *arr, int n)** that returns the index that has the shortest name and the longest name respectively.
3. Program a function
4. Program a function **bubble_sort(Student *arr, int n)** which sorts the name of students lexicographically in ascending order.
5. We open a file **sorted-roll-numbers-names.txt** in writing mode, in which after loading the student entries using **regex**, and sorting them in ascending order lexicographically, we store the names along with their roll numbers.

Algorithm

```
typedef struct
{ char name[STRING_LENGTH]; dtype roll; } Student;

get_shortest(Student *arr, int n)
Begin
  min = strlen(arr[0].name), pos = 0, i = 0;
  for i in range(1, n) do
    current_length = strlen(arr[i].name);
    if (current_length < min) do
      min = current_length; pos = i;
    end if
  return pos;
End


get_longest(Student *arr, int n)
Begin
  max = strlen(arr[0].name), pos = 0, i = 0;
  for i in range(1, n) do
    current_length = strlen(arr[i].name);
    if (current_length > max) do
      max = current_length;
      pos = i;
    end if
  return pos;
End

bubble_sort(Student *arr, int n)
Begin
  for i in range(0, n - 1) do
    for j in range(0, (n - i - 1))
      if (strcmp(arr[j].name,arr[j + 1].name) > 0) then
        swap(arr[j], arr[j + 1]);
      end if
    end for
  end for
End
```

Sample Output

LONGEST NAME : SHUVAYAN GHOSH DASTIDAR
LENGTH : 22

SHORTEST NAME : SK SARJU
LENGTH : 7

```
sorted-roll-number-names.txt
001810501027, ABHIJIT MANDAL
001810501036, ABHISHEK PAL
001810501068, AMAN SHARMA
...
```

----

7. Take a four digit prime number P. Generate a series of large integers L and for each member Li compute the remainder Ri after dividing Li by P. Tabulate Li and Ri. Repeat for seven other four digit prime numbers keeping Li fixed.

Solution Approach

1. We first program a function **generate_large_integer()** which generates an integer of type **long long int**.
2. Set the eight four digit prime numbers and fix the seed for the random number generator.
3. We generate a list of **LENGTH** large integers and for each large integer, we calculate the remainder for each prime number and store them in a file.

Algorithm

```
generate_large_integer()
Begin
  large_integer = 0;
  while (large_integer < LARGE_MIN) do
    for i in range(0, 5) do
         large_integer = (large_integer << 15) | (rand() & 0x7FFF);
    end for
  end while
  return large_integer & 0xFFFFFFFFFFFFFFFFULL;
End

Generating Remainder
for i in range(0, PRIMES_LENGTH) do
  prime = PRIMES[i];
  for j in range(0, LENGTH) do
```

```
   large_integer_remainders[i][j] = large_integer_array[j] %
prime;
  end for
end for
----------------------------------------------------
```

8. Convert your Name and Surname into large integers by juxtaposing integer ASCII codes for alphabet. Print the corresponding converted integer. Cut the large integer into two halves and add the two halves. Compute the remainder after dividing by the prime numbers P in problem 5.

Solution Approach

1. We first take the name on which the operations to be performed as input and store it in an integer array containing the ASCII codes of the characters.
2. Then we split the array and convert the two halves into character version of those numbers i.e. 64 becomes ‘64’.
3. We add those two halves using the function **add(char* num1, char* num2, char* sum)** and store it in the array **sum.**
4. The sum so found out, is entered into the function **mod(char* num, int a)** which gets the remainder after dividing by the number **P**.

Algorithm

```
ASCII value generation and storage
Begin
i = 0, j = 0;
while(name[i] != '\0') do
  if (name[i] != ' ') then
    sprintf(number+j, "%d",name[i]);
    j += digit(name[i]);
  end if
  i++;
end while
End
add(char *num1, char *num2, char *sum)
Begin
```

```
  str_reverse(num1); str_reverse(num2);
  itr1 = 0, itr2 = 0, itr = 0;
  carry = 0;
  while (num1[itr1] != '\0' && num2[itr2] != '\0') do
   sum[itr] = (num1[itr1] - '0') + (num2[itr2] - '0') + carry;
    carry = sum[itr]/10; sum[itr] %= 10;
    sum[itr] += '0';
    itr++; itr1++; itr2++;
  end while
  while (num1[itr1] != '\0') do
    sum[itr] = (num1[itr1]- '0') + carry;
    carry = sum[itr]/10;
    sum[itr] %= 10; sum[itr] += '0';
    itr1++; itr++;
  end while
  while (num2[itr2] != '\0')
    sum[itr] = (num2[itr2] - '0') + carry;
    carry = sum[itr]/10;
    sum[itr] %= 10; sum[itr] += '0';
    itr2++; itr++;
  end while
  sum[itr] = '\0';
  str_reverse(num1);
  str_reverse(num2);
  str_reverse(sum);
End

mod(char *str, int a)
Begin
  res = 0; i =0;
  while (str[i] != '\0')
    res = (res*10 +(str[i] - '0')) % a; i++;
  return res;
End
```

Sample Output

```
[INPUT] ENTER YOUR NAME (IN UPPERCASE): TITIR ADHIKARY
[OUTPUT] ASCII REPRESENTATION : 1133617636950818097
[OUTPUT] NUMBER 1 : 113361763
[OUTPUT] NUMBER 2 : 6950818097
[OUTPUT] SUM : 7064179860
[OUTPUT] REMAINDER : 340
```

Discussions

As the generated ASCII Representation is too big for any primitive data type to handle, we have to convert the representation into a character array and proceed accordingly.

Assignment 2

1.Define an ADT for Polynomials.
Write C data representation and functions for the operation 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

1. In a separate header file , the ADT for the polynomial is created . The ADT is defined as :
   We create two arrays : **polynomials** which is used for storing terms (containing exponents and coefficients of respective terms) and **indexes** for storing index , containing the start index and end index of the terms in the polynomials array.

   ```
   typedef struct {
        int coeff;
        int exp;
   }term;
   typedef  struct {
        int start;
        int end;
   }index;
   const int maxn = 1e5;
   int cnt; // to store the number of current polynomials
   Int tm_cnt; // to store the current number of terms
   term polynomials[maxn];
   index indexes[maxn];
   void insert(int coeff[],  int exp[] , int n);// adds a polynomial
   void update(int index); // updates a polynomial
   int degree( int index); // finds the degree of a certain polynomial
   int is_zero(int index); //returns true if polynomial is zero
   ```

```
Int coeff( int index , int exp) ; //returns coeff of a certain exp
Int mul( int in1 , int in2); // multiply two polynomials
Int add( int in1, int in2); // adds two polynomials
```

Algorithm

- **Insertion**

  Input : An array of coefficients **coeff[]** and another array of its exponents **exp[]** and **size n**
  Result: A polynomial is **inserted**

  ```
  Begin
  start = tm_cnt
  For i <- 0 to n-1:
       term t = ( coeff[ i ] , exp[ i ])
       polynomials[tm_cnt] = t
       tm_cnt += 1
  Endfor
  end = tm_cnt -1
  Index in = ( start , end)
  Indexes[cnt] = in
  cnt += 1
  End
  ```

- **Find degree**

  Input : An index **ind** of the polynomial whose degree is to be found

  ```
  Begin:
  index i = indexes[ ind ]
  term t = polynomials[ i.end ]
  return t.exp
  End
  ```

- **Add two polynomials**

Input: Two indexes in1 and in2 of two different polynomials.
Output : An added polynomial

```
Begin:
index i1 = indexes[ in1 ]
index i2 = indexes[ in2 ]

Initialize an array int vis of size i2.end  - i2.start +1 to 0
Initialize an array  term terms of size i2.end + i1.end - i1.start - i2.start +2
count =0
For i <- i1.start to i1.end:
     Flag = 0
     For j <- i2.start to i2.end:
          If ( polynomials[j].exp == polynomials[i].exp):
               Flag =1
               terms[count] = Term( polynomials[j].coeff + polynomials[i].coeff , polynomials[j].exp)
               count +=1
               vis[j] = 1
          Endif
     Endfor
     If Flag == 0:
          Terms[count] = Term( polynomials[i].coeff , polynomials[i].exp )
          count +=1
     Endif
Endfor

For j <- i2.start to i2.end:
     If ( vis[ j ] == 0) :
          Terms[count] = Term( polynomials[j].coeff , polynomials[j].exp )
          Count += 1
     Endif
```

Endfor

//Terms contains the coeff and exponents of the added polynomial

- **Multiplies two polynomials**

Input: Two indexes in1 and in2 of two different polynomials.
Output : A multiplied polynomial

```
Begin:
index i1 = indexes[ in1 ]
index i2 = indexes[ in2 ]

Initialize an array    term terms of size (i2.end-i2.start+1)*( i1.end-i1.start+1)
count =0
For i <- i1.start to i1.end:
     For j <- i2.start to i2.end:
          term t = ( polynomials[ i ].coeff * polynomials[ j ].coeff , polynomials[ i ].exp + polynomials[ j ].exp )
          Flag =0
            For k <- count  to 0:
               If ( terms[ k ].exp == t.exp ):
                    terms[k].coeff += t.coeff
                    Flag =1
               Endif
          Endfor
          If Flag == 0:
               terms[count] = t
               count += 1
          Endif
     Endfor
```

Endfor

2.Define an ADT for Sparse Matrix .
Write C data representation and functions for the operation 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

1. In a separate header file, the ADT for the sparse matrix is created . The operations on the sparse matrix include that of :
    - ***Formation*** : The sparse matrices are formed from a double dimensional array containing many zeros , and hence sparse . The sparse matrix consists of an array of triples containing the **(row , column , value)** , corresponding to the actual values in the double dimensional array.
    - ***Transpose*** : Calculates the transpose of the sparse matrix and returns a sparse matrix which can be converted to form the transpose of the actual double dimensional array.
    - ***Addition*** : Calculates the sum of two sparse matrices , and so accordingly the sum should be calculated in the double dimensional array.
    - ***Multiplication*** : Calculates the product of two sparse matrices , and so accordingly the sum should be calculated in the double dimensional array.

Representation of the ADT will be :

```
typedef struct{
int i;
int j;
int value;
}tuple;
```

void form(int arr[][]) // creates a sparse matrix from an input 2D array

tuple [] transpose(tuple arr[]) //returns transposed sparse matrix

tuple [] add(tuple arr[]) //returns added sparse matrix

tuple[] multiply(tuple a[] , tuple b[]), // returns multiplied sparse matrix

2. In another file , a menu driven program is written to convert the input 2D array into sparse matrix, and to do the above mentioned operations on it

Algorithm

- **Formation**

Input : A double dimensional array(Arr) containing more zeros of size **(m, n)**
Output : An array of triples comprising the sparse matrix, the first element being
(m , n , non-zero-count)

```
Begin :
Initialize an array of triples A
A[0] = ( m , n , non_zero_count)
cnt =0//for storing element count
for  i <- 1 to m:
     for j <- 1 to n:
          If (  Arr[i][j] != 0):
                    A[cnt] = (i , j , Arr[i][j])
               cnt+=1
     Endfor
Endfor
End
```

- **Transpose**

Input : A Sparse matrix A, A[0] =(m ,n, non_zero_count)
Output : A converted sparse matrix **B** which represents the transpose of the actual 2D array.

```
Begin:
Initialize  an  array B of same size as A
```

```
cnt=0
For i <- 1 to A[0].val:
     Triple t = ( A[i].col , A[i].row,  A[i].value)
     B[cnt] = t
     cnt+= 1
Endfor
End
```

- ***Addition***

Input : Two sparse matrices A and B
Output : Resulting added sparse matrix C

```
Begin:
assert( A[0].row == B[0].row && A[0].col == B[0].col)
Initialize a sparse matrix C of same size as A
Mark[ A[0].val  ] = {0} // Initialize to zero
cnt =0

For i <- 1 to A[0].val:
     Flag =0
     For j <- 1 to B[0].val:
          If ( A[i].row == B[j].row && A[i].col == B[j].col):
               Tuple t = ( A[i].row , A[i].col , A[i].value +
B[j].value)
               C[cnt] = t
               cnt +=1
               Flag =1
               Mark[j] =1
          Endif
     Endfor
     If flag == 0:
          C[cnt] = Tuple ( A[i].row, A[i].col , A[i].val)
          cnt +=1
     Endif
Endfor

For i <- 1 to B[0].val:
     If ( Mark[i] == 0):
          Tuple t = ( B[i].row , B[i].col , B[i].val)
          C[cnt] = t
          cnt += 1
     Endif
```

Endfor

- Multiplication

Input: Two sparse matrices A and B
Output : A resulting multiplied sparse matrix C

```
Begin
assert( A[ 0 ].col == B[0].row)

For i <- 1 to A[0].val:
     R = A[i].row
     For j <- 1 to B[0].val:
          Col = B[j].col
          Tempa = i
          Tempb = j
          Sum =0
          while( tempa < A[0].val && A[tempa].row == R &&
tempb< B[0].val  && B[tempb].col == Col):
               If (A[tempa].row < B[tempb].col):
                    tempa++
               Else if((A[tempa].row > B[tempb].col)) :
                    Tempb++
               Else:
                    sum += A[tempa++].col * b.data[tempb++].row
          If ( sum != 0):
               C[cnt] = Tuple( R , Col , sum)
               cnt += 1
          while ( j < B[0].val && B[0].col == Col):
               j+=1
     Endfor
     while ( i < A[0].val && A[0].row ==R):
               i+=1
Endfor
```

----

3. Define an ADT for List.
Write C data representation and functions for the operation on the List 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

1.In a separate header file, the ADT for the List is created . The operations on the List include that of :

- ***Insert_front*** : Inserts a node at the front of the linked list.
- ***Insert_after*** : Inserts a node after some node
- ***Delete_after*** : Deletes a node after some node
- ***Delete_front :*** Deletes the front node of the linked list .
- ***Sort :*** Returns the nodes of the linked list in sorted order.

Representation of the ADT will be :

```
typedef struct node{
     int data;
     node * next ;
} node;

void  insert_front( node **head);
void  delete_front( node **head);
void delete_after( node *target, node **head);
void insert_after ( node * target , node**head );
void  sort( node** head);
void reverse( node **h);
```

Algorithm

- **Is_empty**

```
void is_empty(node *head){
     return head == NULL
}
```

- **Insert_front**

```
void insert_front( node ** head , node* target ){
     target -> next = (*head);
     *head = target;
}
```

- **Insert_after**

```
void insert_after( node *target , node * prev ){
     target -> next = prev -> next;
     prev -> next = target;
}
```

- **Delete_front**

```
void delete_front( node **head ){
     If ( is_empty(*head)) return;
     *head = (*head)-> next;
}
```

- **Delete_after**

```
void delete_after( node *prev   ){
     If ( prev -> next == NULL ) return;
     prev -> next = prev -> next -> next;
}
```

- **Sort**

```
void sort( node **head){
node *temp = *head;
while ( temp != NULL){
     node *temp2= temp;
     while ( temp2 != NULL){
          if ( temp2 -> data < temp -> data){
               swap( temp2 -> data , temp -> data);
          }
          temp2 = temp2 -> next;
     }
```

```
        temp = temp -> next;
    }
}
```
----

4. Define an ADT for Set.
Write C data representation and functions for the operation on the Set 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

1.In a separate header file, the ADT for the Set is created . The operations on the List include that of :

- ***insert*** : Inserts a new element inside the set
- ***delete*** : Deletes an element from the set
- ***exists*** : Checks whether an element is present inside the set or not.
- ***size :*** Returns the size of the set.

Representation of the ADT will be :

```
const int maxn = 1e5;
int arr[maxn]; // Array based implementation of the set
int cnt;


Void insert(  int data);
Void delete( int data);
int exists( int data);
Int size();
```

ALGORITHM

- **insert**

```
void insert( int data ){
     if (arr[data]) return;
     arr[data] =1;
     cnt +=1
}
```

- **delete**

```
void insert( int data ){
     If ( !arr[data]) return;
     arr[data] = 0;
     cnt -= 1;
}
```

- **exists**

```
int  exists( int data ){
     return arr[data] == 1;
}
```

- **size**

```
int size( ){
     return cnt;
}
```

----

5. Define an ADT for String.
Write C data representation and functions for the operation on the String in a Header file, with array as the basic 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

1. In a separate header file , the ADT for the string is created . The operations on the List include that of :
   - ***_init_*** : Creates a new string of given size.
   - ***concatenate*** : Appends a second string at the end of the first string.
   - ***at :*** Returns the character at a particular index.
   - ***replace :*** Replaces the character at a particular index.
   - ***substring :*** Extracts a substring from starting index to ending position.
   - ***show :*** Displays the string.
   - ***destruct :*** Frees the pointer holding the string array.

Framework

The string structure will have two associated variables:
- maxlen: the maximum number of characters that the string array can hold. (global variable)
- size: which will contain the size of string array.
- *str: which will be a pointer to the string array.

Algorithm

- **Creating a new string**

  Input : A pointer to a structure for string, **s** and the string **size n.**
  Result: A new string of size n.

  Begin
  If (size > maxlen)
      Display "Memory is insufficient to store the string"
  for i : 0 to size - 1
      accept "ch" from user
      s -> str[i++] = ch
  end for
  End

- **Concatenate**

  Input : Two pointers to structures for two strings, **s1** and **s2**
  Result: A new string formed by concatenating the given two strings.

  Begin
  temp = (s1 -> size) + (s2 -> size) ;
  if (temp > maxlen)
  Display "New string can't be created due to memory insufficiency"
  for i : s1 -> size to temp - 1:
  s1 -> str[i] = s2 -> str[i - (s1 -> size)] ;
  end for
  s1 -> size = temp
  End

- **Reference pointer**

  Input : A pointer to a structure for string, namely **s,** and the index, **ind.**
  Result: The character at index **ind**.

  Begin
  if (ind > size)
  Display "Array index out of bounds !!!"
  else
  return s -> str[ind] ;
  End

- **Change character at a particular position**

  Input : A pointer to a structure for string, namely **s,** the position, **pos,** and the
  character, **ch,** to be replaced with.

  Begin
  if (pos > size)
  Display "Array index out of bounds !!!"
  else
  return s -> str[pos] = ch ;

End

- **Obtaining substring between two indices**

  Input : A pointer to a structure for string, **s** and the starting and ending
  positions, **start** and **end,** respectively.
  Result : The required substring.

  ```
  Begin
  If ( start < 0 ) OR ( end >= size )
       Display "Array index out of bounds !!!"
       exit(0) ;
  _init_( string * s1 , end - start + 1 ) ;
  else
       for i : 0 to (end - start)
            s1 -> str[i] = s -> str[ i + start ] ;
       end for
  end if
  return s1;
  End
  ```

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 odd integer like 3,5,7. Explain what you will do with boundary conditions.

Solution Approach

1. Separate functions are created for minimum, average, maximum and median and are called independently to calculate the output array for each of this operations.
2. For the output array to be of a size equal to the input array , the input array has to padded with lengths (k-1)/2 in both ends of the input array. The array is padded with **0**.

3. ***Maximum :*** To calculate the maximum output array using the filter , the filter is slid along the array with a stride of 1 , and a total of n elements are obtained in the output array.
4. ***Minimum :*** To calculate the minimum output array using the filter , the filter is slid along the array with a stride of 1 , and a total of n elements are obtained in the output array.
5. ***Average :*** To calculate the average output array using the filter , the filter is slid along the array with a stride of 1 , and a total of n elements are obtained in the output array.
6. ***Median :*** To calculate the median output array using the filter , the filter is slid along the array with a stride of 1 , and a total of n elements are obtained in the output array.

Algorithm

- **PAD**

Input : An Input array arr[] of size n , and k representing the size of filter

Output : The input array padded with 0 on both sides of length (k-1)/2

```
Begin:
Initialize an array output[] of size n + k-1 =0
cnt =(k-1)/2
For i <- cnt to n +  cnt -1:
     Output[ i ] = arr[ i - cnt ]
Endfor
return output
End
```

- **Maximum**

Input : An Input array arr[] of size n , and k representing the filter size

Output : An array of size n , constituting the maximum of filter size in the array for each slide.

```
Begin:
Initialize an array output[] of size n + k-1 =0
output = pad( input ) // the input array is padded to form size n +  k - 1
For i <- 0 to n-1:
     mx =0
     For j <- 0 to k-1:
          mx = max(  mx,  arr[ j + i ] )
     Endfor
     output[ j ] = mx
Endfor
End
```

- **Minimum**

Input : An Input array arr[] of size n , and k representing the filter size

Output : An array of size n , constituting the minimum of filter size in the array for each slide.

```
Begin:
Initialize an array output[] of size n + k-1 =0
output = pad( input ) // the input array is padded to form size n +  k - 1
For i <- 0 to n-1:
     mn =0
     For j <- 0 to k-1:
          mn = min(  mx,  arr[ j + i ] )
     Endfor
     output[ j ] = mn
Endfor
End
```

- **Average**

Input : An Input array arr[] of size n , and k representing the filter size

Output : An array of size n , constituting the average of filter size in the array for each slide.

```
Begin:
Initialize an array output[] of size n + k-1 =0
output = pad( input ) // the input array is padded to form size n + k - 1
For i <- 0 to n-1:
     mx =0
     For j <- 0 to k-1:
          mx  += arr[ j + i ] )
     Endfor
     output[ j ] = mx / k
Endfor
End
```

- **Median**

Input : An Input array arr[] of size n , and k representing the filter size

Output : An array of size n , constituting the median of filter size in the array for each slide.

```
Begin:
Initialize an array output[] of size n + k-1 =0
output = pad( input ) // the input array is padded to form size n + k - 1
For i <- 0 to n-1:
     Initialize an array temp[] of size k
     For j <- 0 to k-1:
          temp[ j ] =   arr[ j + i ]
     Endfor
     sort the temp array
     output[ j ] = temp[ k/2 ]
Endfor
End
```

Discussion

The boundaries of the array for sliding the window is handled by padding the elements with 0. This ensures the output array after applying the filter to be of size equal to the size of the original array.

----

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 position within the large matrix. Explain how you will handle the boundary values.

Solution Approach

- Create and initialize two square matrices, namely **mat** and **sub_mat**.
- Let temp = (dimension of **sub_mat** + 1) / 2. Then, pad the matrix **mat** on all four sides with **temp** rows / columns. Increase **mat**'s dimensions while declaring the matrix.
- Multiply the padded **mat** with **sub_mat** and place the result at the position in **mat** corresponding to the central position of **sub_mat**.
- The output will be the new **mat** after discarding the padded rows / columns.

  The problem will essentially require three functions:

  - fill_data(int **mat, int dim) : to initialize the matrices with values entered by the user.
  - out_of_range(int i) : to check if a particular index is out of the accessible range of rows / columns.
  - multiply(int mat[][], int sub_mat[][]) : to multiply the two matrices

Algorithm

- **Check if a particular index is out of range :** takes the index **ind** as argument

  ```
  Begin:
  N = dimension of mat
  if ( ind < 0 ) OR ( ind > N )
       return TRUE
  else
       return FALSE
  End
  ```

- **Multiplies two matrices**

  Input: Two matrices of integer type
  Output : The multiplied matrix

  ```
  Begin:
  N = dimension of mat
  M = dimension of sub_mat
  mid = ( M - 1 ) / 2 ;

  for row : 0 to N
  for col : 0 to N
       val = 0 ;
       for sub_col : 0 to M
            index = row - mid + sub_col ;
            if out_of_range(index) : FALSE
                 out [ row ] [ col ] := out [ row ] [ col ]
  + mat [ index ] [                                     col
  ] * sub_mat [ mid ] [ sub_col ] ;
            end if
       end for ( sub_col )
  end for ( row , column )
  End
  ```

Discussion

The sub matrix has to be of an odd number of dimensions to get the centre of the matrix. The given algorithm takes O ($n^2 * m$) and quadratic amount of auxiliary memory.

----

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

Solution Approach

- The given array is traversed to search for variations in the data stored. It has been assumed that the array contains only integer type values.
- Two variables are created, namely "*asc*" and "*des*", to store info regarding whether the array is sorted in ascending or descending order.
- Initialize both the variables to 0.
- The array is then traversed till the semi-last element taking two elements at a time.
- If the first element is smaller than the second element, then set asc = 1, else des = 1 if it is greater. [No change in case of equality]
- After all the elements have been checked, if:
    - asc = 0 & des = 0, all the array elements are equal.
    - asc = 1 & des = 0, array is in ascending order.
    - asc = 0 & des = 1, array is in descending order.
    - asc = 1 & des = 1, array is unsorted.

Framework

The array will have two associated variables:

- size: which will contain the size of the array.
- *arr: which will a pointer to the array header.

The functions associated with the array are:

- get_dim(): to get the size of the array from user.
- get_data(): to get the array elements from the user.
- det_order(): to get the order of the sorted array.

Algorithm

```
● int get_dim ( void ) {
      int temp;     // to store value accepted from user
      return temp;
  }

● void get_data ( int *arr, int size ) {
      for i : 1 to size
      accept *(arr + i) }
● void det_order ( int *arr, int n) {
      int asc = des = 0;
      for i : 1 to (size - 1)
          if ( arr[ i ] < arr[ i + 1] )
          asc = 1;
          else if ( arr[ i ] > arr[ i + 1])
          des = 1;
          if ( asc = 1 & des = 1 )
          break for loop;
      end for
      if ( asc = 0 & des = 0)
          display " all the array elements are equal "
      if ( asc = 1 & des = 0)
          display " the array elements are in ascending order "
      if ( asc = 0 & des = 1)
          display " the array elements are in descending order "
      if ( asc = 1 & des = 1)
          display " array is unsorted "
  }
```

Results

This algorithm yielded efficient results with linear time complexity. It has been observed that for random cases, it wasn't even necessary to traverse the whole array to get the result. It further minimizes the checking within the loop to just three verifications, thereby giving a O(3*n) worst case complexity.

----

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

Solution Approach

The only function required is Merge (int * s1, int size1, int * s2, int size2) which takes in two sorted arrays and their sizes as arguments from the testing function, and merges them into a single sorted array. The Merge function will operate as follows :

1. Declare two pointers for both arrays and make them point to the first element of each array. Create a third array and set it to NULL.
2. Traverse both the arrays simultaneously until one of them reaches the end.
3. Get the values pointed by each pointer. Push the smaller value into the third array and increment its corresponding pointer by one.
4. After one pointer reaches the end of its array, push all the elements of the other array into the third one, in order of their scanning.

Algorithm

- **Merge the two sorted arrays:**

Input : Two arrays containing integer elements and variables containing their respective sizes.

Output : A third array containing elements of both the arrays in sorted order.

*int * Merge : takes int * arr1, int size1, int * arr2, int size2*
Create a third array of size = size1 + size2
ptr1 = ptr2 = ptr3 = 0

```
while: ( ptr1 < size1 ) & ( ptr2 < size2 )
     if ( arr1[ptr1] <= arr2[ptr2] )
          arr3[ptr3++] = arr[ptr1]
          ptr1 := ptr1 + 1
     else
          arr3[ptr3++] = arr[ptr2]
          ptr2 := ptr2 + 1
     end if
end while

if ( ptr1 != size1 ) : copy all the elements of arr1 from
ptr1 to size1
if ( ptr2 != size2 ) : copy all the elements of arr2 from
ptr2 to size2
                  into the third array
Return * arr3;      // returning the third array
```

Discussion

This is a very popular technique and is commonly used in Merge Sort which implements a "divide and conquer" framework. This process makes full use of the advantage that the input arrays are already sorted, thereby providing excellent form of stable sort.

Assignment 3

1. Implement the following functions of ADT Linked List using singly linked list as a header file:

init_l(cur) – initialise a list
empty_l(head) – boolean function to return true if list pointed to by head is empty
atend_l(cur) – boolean function to return true if cur points to the last node in the list
insert_front(target, head) – insert the node pointed to by target as the first node of the list pointed to by head
insert_after(target, prev) – insert the node pointed to by target after the node pointed to by prev
delete_front(head) – delete the first element of the list pointed to by head
delete_after(prev) – delete the node after the one pointed to by prev

Solution Approach

We need to create the following functions and add them to a header file so that we can use SLL ADT in other programs. There will be an initialization function which creates and allocates space for a node, a check function to find whether the node is at the end of the list(if the next pointer of the node is null), and other functions to manipulate the before and after pointers and add/remove nodes.

Algorithm

- **init_l(curr):** uses malloc to allocated memory for the new node of size Node.
- **empty_l(head):**

    - *if(head == null)*
      => list is empty
- **at_end(head):**
    - *if(the next pointer of head is null)*
      => the node is at the end
- **insert_front(target, head):**

  if(head == null)

      head = target

  else

      target->next = head

      target = head

- **insert_after(target, prev):**

      if prev == null

      target->next = prev

      else

      node* temp = prev->next

      prev->next = target

      target->next = temp

- **delete_after(prev):**

      if prev == null

      abort

      if prev->next == null

      abort

      else

      node* temp = prev->next

prev = prev->next->next

delete(temp)

Question 2: Read integers from a file and arrange them in a linked list (a) in the order they are read, (b) in reverse order. Show the lists by printing.

Solution Approach
For storing the numbers in the same order they are read we just need to store every number at the end of the linked list everytime.For storing the numbers in reverse order we need insert the number before head node of the list everytime.

Algorithm

- **Read integers from list**
  Input: Filename and a bool flag to determine the order
```
readIntegers(char* filename, int order)
     begin:
          verify the file exists or not
          open the file in read mode
          node* forward = init_l()
          node* reverse = init_l()
          value = read a integer from the file & store
          node* curr = createnode(value)
          if order = 1//denotes forward storing
               insert_after(curr,&forward)
               prev = curr
          else
               if order = -1//denotes reverse storing
                    insert_front(curr,&reverse)
               endif
          endif
          close the file
     end
```

General Discussion
Everytime we work with dynamic memory allocation we need to take care of the malloc error, memory leakage and memory size such kind of things. We also should check the existence of the file that we are given.

Question 3

Implement the following functions in a menu-driven C program using the data structure operation of Singly Linked List in the header file developed in problem 1:

a) print a list (i) in the same order, (ii) in the reverse order.

b) find the size of a list in number of nodes

c) check whether two lists are equal

d) search for a key in (i) an unordered list, (ii) an ordered list(Return the node if key found and delete the node from original list)

e) append a list at the end of another list.

f) delete the nth Node, last node and first node of a list.

g) check whether a list is ordered

h) merge two sorted lists

i) insert a target node in the beginning, before a specified node and at the end of the list (sorted and unsorted).

j) remove duplicates from a linked list (sorted and unsorted)

k) swap elements of a list pairwise

l) move last element to front of a list

m) delete alternate nodes of a list

n) rotate a list

o) delete a list.

p) reverse a list.

q) sort a list.

Solution Approach

We have to achieve the above functionalities using the header file created for singly linked list.So we need to create separate function in order to achieve that

```
int return_size(node*);//just to show null error
void create_a_list();
```

```
void print_a_list();
void print_a_particular_list(int);
void print_all_list();
void print_in_reverse_main();
void print_in_reverse(node*);
void find_number_of_nodes();
void compare_two_lists();
void delete();
void delete_alternate_nodes();
void search_a_key();
```

Algorithm

- **create_a_list()**
```
begin
     int user_input = /*take input from user*/
     node* head = init_l(user_input)
     if head == NULL
          raise error
     else
          return head
end
```

- **Print a list**
  - **In same order**
    INPUT: head node of the list which is to be printed
```
begin
     node* head;
     if head = NULL
          raise empty error
     else
          while head -> next != NULL
               print head->data
               head = head->next
          end while
     endif
end
```
  - **In reverse order**
    Input: head node of the list that is to be printed

    print_in_reverse(head)

```
        begin
            print_in_reverse(head->next)
            print head
        end
```

- **Size of a List**
  Input: head node of the list whose number of nodes is to be determined

```
    size_of_a_list(node* head)
    begin
        count = 0
        while head -> next != NULL
            count = count + 1
        end while
        count = count + 1
        return count
    end
```

- **Check Weather the lists are equal or not**
  Input: head nodes of two lists which are to be compared

```
compare_two_lists(node* head1, node* head2)
    begin
        size1 = size_of_a_list(head1)
size2 = size_of_a_list(head2)
if head1 = 0 and head2 = 0
    print equal
else if head1 != head2
    print unequal
else
    for i=0 to i<head1 in steps of 1
        if head1->data != head2->data
            print unequal
            break
        else
            head1 = head1->next
            head2 = head-> next
        endif
    end for
    print equal
end if
```

- **Search for a key**
  Input: head node of the list and the key value to be searched

```
search_for_a_key(node* head, int key)

     begin:

     while atend_l(temp)
          if head->data = key

          node* temp = head
          while temp != NULL
               if temp->next->data = key
                    print found
                    return temp->next
                    delete_after(temp)
               else
                    print key not found
               endif
          end while
     end
```

- **Append a list at the end of another list**
  Input: head nodes of two lists

```
append_two_lists(node* head1, node* head2)
     begin:
          node* temp = head1
          while temp != NULL
               temp = temp->next
          end while
          temp->next = head2
     end
```

- **Delete nth node**
  Input: The head node and the position of the node in that list

```
delete_nth_node(node* head, int n)
     begin:
          size = size_of_a_list(head)
          if n > size
               print invalid number of node
```

```
          else if size = 0
               print list has no element
          else
               for i = 0 to n-1 in steps of 1 do
                    head = head->next
                    delete_after(head)
               end for
          endif
     end
```

- **Delete last node**
  Input: Delete the last node of the given list

```
delete_last_node(node* head)
     begin:
          while head->next != NULL
               delete_after(head)
          end while
     end
```

- **Delete first node**
  Input: head node of the list

```
delete_first_node(node* head)
     begin:
          delete_front(head)
     end
```

- **Check if the list is ordered or not**
  Input: head node of the list to be checked

```
     begin:
          flag = 0
          size = size_of_a_list(head)
          if size = 0 or size = 1 or size = 2
               print list is sorted
          else
               if head->data > head->next->data
                    /*go for descending order checking*/
                    while head->next != NULL
                         if head->data < head->next->data
                              flag = 1
                              print Unordered
                              break
```

```
                    end if
                end while
            if flag = 0
                print ordered
            end if

            else
                if head->data < head->next->data
                /*go for descending order checking*/
                while head->next != NULL
                    if head->data > head->next->data
                        flag = 1
                        print Unordered
                        break
                    end if
                end while
                if flag = 0
                    print ordered
                end if
            end if
        end if
    end
```

- **Swap elements of a list**
  Input: two nodes whose values are to be swapped

```
swap(node* a,node* b)
    begin:
        int temp = a->data
        a->data = b->data
        b->data = a->data
    end
```

- **Sort a list**
  Input: head node of the list to be sorted and an integer to determine the way to sort(ascending or descending)

```
sort(node* head,int order)
    begin:
        size = size_of_a_list(head)
        if size =0 or size = 1 or size = 2
            print already sorted
```

```
          return
     else
          swapped = 0
          node* ptr
          node* lptr = NULL
          do
               swapped = 0
               ptr = head
               while ptr->next != NULL
                    if order = 1
                      if ptr->data > ptr->next->data
                         swap(ptr,ptr->next)
                         swapped = 1
                    endif
               elseif
                    if order = -1
                        if ptr->data<ptr->next->data
                              swap(ptr,ptr->next)
                              swapped = 1
                         endif
                    endif
                    ptr = ptr->next
               endif
               lptr = ptr
          while swapped = 1
          endwhile
     endif
end
```

- **Merge two sorted list**
  Input: head node for the two sorted list

```
merge_two_list(node* head1,node* head2)
     begin:
          node** final
          if empty_l(head1) = true
               *final = head2
               return final
          elseif empty_l(head2) = true
               *final = head1
               return final
          else
```

```
            order = 1
            if head1->data > head1-> data
                order = -1 //denotes descending order
            else
                order = 1 //denotes ascending order
            endif
        endif
        *final = head1
        append_two_lists(*final,head2)
        sort(final,order)
    end
```

- **Move last element to the front**
  Input:head node of the list

```
move_to_the_first(node* head)
    begin:
        node* temp = head
        while temp->next != NULL
            temp = temp->next
        end while
        temp->next->next = head
        temp->next = NULL
    end
```

- **Reverse a list**
  Input: head node of the list to be reversed

```
reverse_a_list(node* head)
    begin:
        if empty_l(head) = true
            print list is empty
            return
        else
            size = size_of_a_list(head)
            if size = 1
                return// list already reveerse
            else
                node* curr = head
                node* prev = init_l()
                while curr != NULL
                    next = current->next
                    current->next = prev
```

```
                        prev = current
                        current = next
                    end while
                *head = prev
                endif
            endif
        end
```

General Discussion

The first question that arises that why should we use linked list in the first place. Well the answer is it has some benefits over arrays. Linked lists can grow in size(theoretically infinitely but practically it’s limited by the memory size of the heap memory). Moreover Insertion and deletion in any linked list takes constant time whereas in the array it takes O(n) time.The only problem with linked list is that searching and sorting in linked list takes O(n) and O(n^2) time respectively.In array it is way more faster.

4. Write all the above operations of Single Linked List for the implementation using array.

Solution Approach

We need to achieve all the functionalities of the linked list but using only arrays. For this we need to take a 2D array with two fields namely data field and cursor field.Data field contains the data corresponding to every node. The cursor field somewhat does the job of pointers in a trivial linked list.Every cursor field contains the index of next linked node in that list.We need an another variable named available which always points to the next available node present in the array.

for achieving those functionalities we need to define these functions:

```
int check_size();
int is_valid(int);//return true if the list is valid
int size_of_a_list(int);
void new_list();
```

```
void display_one_list();
void display_all_list();
void display();
void display_size();
void insert_a_new_data();
void enter_before_head(int,int);
void enter_at_a_position(int,int);
void enter_at_end(int,int);
void delete_something();
void delete_first_element(int);
void delete_nth_element(int);
void delete_last_element(int);
void driver_size_of_a_list();
void compare_two_lists();
void merge_two_lists();
void reverse_print_a_list();
```

Algorithm:

```
struct node
{
        int data;
        int next;
};

struct node s[max];//initialize the array of data
int head[head_count];//contains the index of head of the
int list_no = 0;
int size = max; // contain no of available nodes
int available = 0;
```

- **Create a new list**

```
begin:
     if list_no >= head_count or size <= 0
          print not enough space
          return
     else
          head[list_no] = available
          list_no = list_no + 1
          for i = 0 to num in steps of 1 do
```

```
                temp = input()
                s[available.data] = temp
                if i!= num-1
                     available = s[available].next
                elseif i = num-1
                     papa = s[available].next
                     s[avialable].next = -1
                     available = papa
                endif
           end for
      endif
end
```

- **Display a list**

  Input: index of the head node of the list

```
begin:
     index = head[a]//a is the index of head node
     while s[index].next != -1
          print s[index].data
          index = s[index].next
     end while
     if s[index].next == -1
          print s[index].data
     endif
end
```

- **Insert a new data to a list**

  Input: head index(=a) of the list and the data(=data) to be entered

  - **Insert before head**

```
          begin:
               s[available].data = data
               temp = s[available].next
               s[availble].next = head[a]
               head[a]= available
               available = temp
```

```
end
```

- **Insert at a particular position**

```
begin:
     temp = s[head[a]].next
     for i=1 to pos-1 in steps of 1 do
          temp = s[temp].next
     end for
     s[available].data = data
     store = s[available].next
     s[temp].next = available
     s[available].next = temp1
     available = store
end
```

- **insert at end**

```
begin:
     temp = s[head[a]].next
     while s[temp].next != 1
          temp = s[temp].next
     end while
     s[available].data = data
     store = s[availble].next
     s[available].next = available
     available = store
end
```

- **Delete a data from the list**
    - **Delete first element**

```
begin:
      s[head[a]].data = '\0';
      int store = s[head[a]].next
     s[head[a]].next = available
     available = head[a]
     head[a] = store
     size = size+1
end
```

- **Delete nth element**

```
begin:
     start = head[a]
     temp = s[start].next
     for i=1 to n-1 in steps of 1 do
          temp = s[temp].next
     end for
     temp1 = s[temp].next
     size++
     s[temp].next = available
     available = temp
     for i=1 in steps of 1 do
          if s[start].next = temp
               s[start].next = temp1
               break
          else
               start = s[start].next
          endif
     end for
end
```

- **Delete last element**

```
begin:
     store = s[head[a]].next
     while s[s[store].next].next = -1
                store = s[store].next
     end while
     s[s[store].next].next = available
     available = s[store].next;
     s[store].next = -1
     size++
end
```

- **Determine size of a list**

Input: the index of the list head

```
begin:
     count = 1
     store = s[head[choice]].next
     while s[store].next != -1
          count = count + 1
```

```
            store = s[store].next
        end while
        count = count + 1
        return count
    end
```

- **Compare two lists**

  Input: head index of the two lists to be compared

```
    begin:
        /*verity both the lists*/
        if isvalid(head1) and isvalid(head2) = false
            print INVALID LIST
            return
        else
            len1 = size_of_a_list(first)
            len2 = size_of_a_list(second)
            count=0
            if len1 != len2
                print UNEQUAL
                return
            else
                int start1 = head[first]
                int start2 = head[second]
                while
                    if      s[start1].next!=-1      &&
                s[start2].next != -1
                        print UNEQUAL
                        count = 1
                        break
                    else
                        start1 = s[start1].next
                         start2 = s[start2].next
                    endif
                end while
                if count = 0
                    print EQUAL
                endif
            endif
        endif
    end
```

- **Merge two lists**
  Input: head index of the two lists to be merged

```
begin:
     /*check validity of the lists*/
     store = head[first]
     while s[store].next != -1
           store = s[store].next
     end while
     s[store].next = head[second]
     //update head array
     second++
     while second < max
           head[second-1] = head[second];
           second++
     end while
end
```

- **Reverse print a list**
  Input: the index of the list head

```
begin:
     len = size_of_a_list(choice)
     temp[len],i=0
     start = head[choice]
     while s[start].next != -1
          temp[len-i-1] = s[start].data
          start = s[start].next
          i++
     end while
     s[start].data = temp[0]
     for i=0 to len in steps of 1 do
          print temp[i]
     end for
     print 'NULL'
end
```

General Discussion:
This is an unorthodox representation of a singly linked list.

There no direct benefit of this type of representation. But this program uses stack memory of the system whereas normal linked list uses heap part of the memory.The memory requirement for both the representation are almost same.

5. Repeat problems 1 and 3 for a circular single linked list, doubly linked list and circular doubly linked list.

Solution Approach

The approach for doubly linked list is almost the same, except that there's a new node pointing to the previous node which will make traversal easier
The approach for circular linked list means, the last pointed (called the tail pointer) will point to the first node and this will also speed up traversal in some cases

Algorithm(Doubly Linked List):

- init_l(curr): *Allocate space for a new node using malloc*
- empty_l(curr): *Check if curr is null or not, if yes then empty.*
- at_end(curr): *Check if the forward pointer is null, if yes, then the node is the end or else not.*
- insert_front(target, head):
  target->next = head
  head->prev = target
- insert_after(target, previous):
  node temp = prev->next*
  previous->next = target
  target->next = temp
  temp->prev = target
  target->prev = previous
- delete_front(head):
  - *node* temp = head*
  - *head = head->next*
  - *head->prev = null*
  - *delete(temp)*
- delete_after(previous):
  - *node* temp = previous->next*

- *previous->next = previous->next->next*
- *if(previous->next == null)*
  end
- *else*
- *node* nextNode = previous->next*
- *nextNode->prev = previous*
- *delete(temp)*

- Print the list in the same order
  - *while(!empty_l(head))*
    print(head)
    head = head->next
- Print the list in the reverse order
  - *while(!at_end(head))*
    head = head->next
  - *while(!empty_l(head)*
    print(head)
    head = head->prev

- Find the size
  - *while(!empty_l(head))*
    count++
- Check if two lists are equal
  - *if(empty_l(head1) && empty_l(head2))*
    return true
  - *bool check = true*
  - *else if(size_l(head1) == size_l(head2))*
    while(!empty(head1))
    if(head1 != head2)
    check = false
    return false
    head1 = head1->next
    head2 = head2->next

  else return false

- Search for the node in an unordered/ordered list:
  - *It will take the same time to search in ordered and unordered linked list, because it takes the same time to traverse to any node*
  - *while(!empty_l(head))*
    if(head == toBeSearchedNode)

```
        node* previous = head.prev;
        delete_after(previous)
        return true

return false
```

- Append a list at the end of another list
  - *head1, head2 => pointers to the two lists*
  - ```
    while(!at_end(head1))
        head1 = head1->next
    head1->next = head2
    ```
- Delete the first node, the last node, and the nth node of a list
  - *Deleting the first node*
    ```
        delete_front(head)
    ```
  - *Deleting the nth node*
    ```
        int count = 1;
        while(count == n || empty_l(head))
            count++;
            if(count == n)
                node* previous = head->prev
                delete_after(previous)
                return
        if(empty_l(head))
            >> there are less than n nodes
    ```
  - *Deleting the last node*
    ```
        while(at_end(head))
            head = head->next
        delete_after(head->prev)
    ```
- Check whether the list is ordered
  - *int states(0 implies all equal, 1 implies increasing and -1 implies that its decreasing*
  - ```
    if(size_l(head) == 1)
        return 0
    else
        find_state(head)
            if(head->ele == head->next->ele)
                state = 0
            else if(head->ele < head->next->ele)
                state = 1
            else state = -1
    ```

```
head = head->next
while(!empty_l(head))
     newstate = find_state(head)
     if(newstate == 0)
          continue
     else if(newstate == state)
          continue
     else
          >>it's not ordered
          >>return
>>its ordered and print the state
end
```

- Merge two sorted Lists

```
head3 is the new list.
while(!empty_l(head1) && !empty_l(head2))
     if(head1->ele < head2->ele)
          insert_after(head1, head3)
          head1->next
     else if(head2->ele < head2->ele)
          insert->after(head2, head3)
          head2->next
     else
          insert->after(head1, head3)
          insert->after(head2, head3)
          head1->next
          head2->next
```

- Insert a target node at the beginning, before a specified node and at the end of the list
    - *while(!at_end(head1)*
      head1 = head1->next
      head1->next = target
      target->prev = head1
    - *head1 is the list*
      node target
    - *insert_front(target, head) //inserts at the beginning*
      node ptr is a pointer to the specific node*
      insert_after(target, ptr->prev)

- Sort a linked list

```
head1 is a pointer to the head node of the list
```

temph1 = head1
temph2 = head1->next
while(!empty_l(head1))
    head2 = head1
    while(!empty_l(head2->next))
        ele1 = head2->ele
        ele2 = head2->next->ele
        if(ele1 > ele2)
            swap(ele1, ele2)
        head2 = head2->next
    head1 = head1->next

- Remove duplicates from unsorted/unsorted list
    - *Sort the list(if unsorted)*
    - *head1 is the list*

    ```
    while(!empty_l(head1->next))
        if(head1->ele == head1->next->ele)
            delete_after(head1)
        head1 = head1->next
    ```
- Swap elements of a list pairwise
    - *Node* end_ptr, front_ptr*
      front_prt = head
    - *while(!at_end(head))*

    ```
        head = head->next
    end_ptr = head
    while(front_ptr != end_ptr || end_ptr->next!=end_ptr)
        swap(front_ptr, end_ptr)
        front_ptr = front_ptr->next
        end_ptr = end_ptr->prev
    ```
- Delete the alternate nodes of a list

    ```
    delete_front(head)
    while(!empty_l(head) || !at_end(head))
        delete_after(head)
        head = head->next
        if(head->next == null)
            end
        else
            head = head->next
    ```
- Rotate a list n times
    - *for(int i = 1 ; i <= n ; increment i by 1)*
      node endptr, frontptr*

```
node* dummy = endptr
while(!empty_l(frontptr))
    node* temp = frontptr
    frontptr->ele = dummy->ele
    dummy->ele = temp->ele
    frontptr=frontptr->next

endofwhile
```

- Delete a list
  ```
  while(!at_end(head))
      head = head->next
  while(!empty_l(head))
      delete_after(head->prev)
  ```
- Reverse a list: *swap elements of the list pairwise*

Algorithm(Circular List)

- init_l(curr)
  - *allocate memory for curr, which will be used in the list*
- *empty_l(head)*
  - ```
    if(head==null)
        return true
    return false
    ```
- *at_end(curr, tail)*
  - ```
    if(curr == tail)
        return true
    return false
    ```
- *insert_front(target, tail)*
  ```
  Node* temp = tail.prev
  temp->next = target
  target->prev = temp
  target->next = tail
  tail->prev = target
  ```
- *insert_after(target, tail)*
  - *similar to insert_after of doubly linked list*
- *delete_after(tail)*

- Node temp = tail->next
  - tail = tail->next
  - delete(taili)
- print_list(tail)
```
Node* temp = tail
while(temp != tail)
     temp = temp->next
     print(temp)
print(temp
```
- print_list_reverse(tail)
```
node* temp = tail
while(temp != tail)
     print(temp)
     temp = temp->prev
```

- size_l(tail)
```
node* temp = tail
print(temp)
temp = temp->next
count++
while(temp! = tail)
     count++;
     temp = temp->prev
```
- equal(tail1, tail2)
```
if(tail1 == tail2 && tail1 == null)
     return true
else
if(size_l(tail1) != size_l(taile2)
     return false
else
     node* n = tail
     n = n->next
     while(n != tail)
          if(equal nodes)
               return true
     return false
```
- search_for_a_key(tail, key) :
```
node* n = tail->next
while(n != tail)
     if(n->ele == key)
```

```
            return true
    if(tail->ele == key)
        return true
    return false
```

- ordered_or_not(tail):
    - int states(0 implies all equal, 1 implies increasing and -1 implies that its decreasing
    - if(size_l(tail) == 1)

```
        return 0
    else
        find_state(tail)
            if(tail->ele == tail->next->ele)
                state = 0
            else if(tail->ele < tail->next->ele)
                state = 1
            else state = -1
    head = head->next
    while(!at_end(tail))
        newstate = find_state(head)
        if(newstate == 0)
            continue
        else if(newstate == state)
            continue
        else
            >>it's not ordered
            >>return
    >>its ordered and print the state
    end
```

Question 6

Implement an application to find out the Inverted Index of a set of text files.

Program Approach:

The structure which will hold the elements is as follows ::

```
++++++++++++++++++++++
| prev       |        |
 ++++++++++++++ docu_  +
 | word       |        |
 ++++++++++++++ name   +
 | next       |        |
 ++++++++++++++++++++++
```

Pre-Requisites ::

1. The document should have all small letters

The algorithm is as follows ::

1. Take 50 elements from the document in an array
2. Sort them out and then merge them in the list
3. While doing so maintain the name of the document containing them
4. Also while adding an element check if it is already present or not. If so just append the document containing it.

In format document, we take a file and converted each character
Into lower case characters. We will omit the non-alphanumeric characters with the help of this function. Ultimately, we'll write the output into a binary file
We will use radix sort to sort the strings in the documetn.
Also we'll take at most 50 words at once and place them in the in a place at their proper position.

sort_strings :: To sort the elements
comp_strings :: To compare the strings
create_node :: To create the node with proper initialization
place_elem :: To place the elements in their respective
rem_duplicate :: To remove the duplicate element in the list
sort_strings:: Explanation We will sort on the basis of the following priority order
SENTINAL_CHAR < Lowercase_char < Digits

- In place elem, we'll follow this algorithm

- 1. If the head is NULL, then the list contains no elements. So add elements unjudiciously. Also since these words will be coming the same document, just check for duplicates.
- If so ignore the word.
- Else, if the element is less than the head, then make that at the first using insert_front function
- Else use insert_after to insert before the element immediately greater than element,provided that the element is not present already. In such a case use append_doc.

Algorithm:

```
typedef struct doc_{
    char doc_name[MAX_WORD_SIZE];
    struct doc_* next_doc;
}doc;

typedef struct node_tag{
    struct node_tag* prev;
    char word[MAX_WORD_SIZE];
    doc* docs;
    struct node_tag* next;
}node;

void format_doc(FILE* fp, char* name){
    FILE* f_new = fopen(name,"w");
    fseek(fp,0,SEEK_SET);
    char c = '\0';
    while(!feof(fp)){
            char str[MAX_WORD_SIZE] = "\0";
            int str_index = 0;

            while(isalnum(c=fgetc(fp))){
                if(c>='A' && c<='Z') c = c - 'A' + 'a';
                str[str_index++] = c;
                }

            while(str_index>0 && str_index<MAX_WORD_SIZE-1)
                str[str_index++] = SENTINAL_CHAR;
            str[str_index] = '\0';
```

```
            if(str_index != 0){
                //printf("In format_doc we're writing ::
%s\n",str);
                fwrite(str,sizeof(char),MAX_WORD_SIZE,f_new);
            }
    }
    fclose(f_new);
    return;
}
void sort_strings(char str[STR_ARRAY_LEN][MAX_WORD_SIZE]){
    for(int i=MAX_WORD_SIZE-1;i>=0; i--){
        int count[37] = {0};
        for(int j=0; j<STR_ARRAY_LEN; j++){
            if(isdigit(str[j][i])) count[str[j][i]-'0'+27]++;
            else if(isalpha(str[j][i])) count[str[j][i] -
'a'+1]++;
            else if(str[j][i] == SENTINAL_CHAR) count[0]++;
        }

        int pos[37] = {0};
        for(int m=1;m<37;m++){
            pos[m]=pos[m-1]+count[m-1];
        }

        char temp[STR_ARRAY_LEN][MAX_WORD_SIZE];

        for(int k=0; k<STR_ARRAY_LEN; k++){
            if(isdigit(str[k][i]))
strcpy(temp[pos[str[k][i]-'0'+27]++],str[k]);
            else if(isalpha(str[k][i]))
strcpy(temp[pos[str[k][i]-'a'+1]++], str[k]);
            else strcpy(temp[pos[0]++],str[k]);
        }

        for(int l=0; l<STR_ARRAY_LEN; l++)
            strcpy(str[l],temp[l]);
    }
return;
}

int comp_strings(char str1[MAX_WORD_SIZE], char
str2[MAX_WORD_SIZE]){
```

```
    for(int i=0; i<MAX_WORD_SIZE-1; i++){
        if(str1[i] == SENTINAL_CHAR || str2[i] ==
SENTINAL_CHAR){
            if(str2[i]!= SENTINAL_CHAR) return IS_LESSER;
            else if(str1[i] != SENTINAL_CHAR) return
IS_GREATER;
            else continue;
        }
        else if(isalpha(str1[i]) && isalpha(str2[i])){
            if(str1[i]>str2[i])
                return IS_GREATER;
            else if(str1[i]<str2[i]) return IS_LESSER;
            else continue;
        }
        else if(isdigit(str1[i]) || isdigit(str2[i])){
            if(!isdigit(str2[i])) return IS_GREATER;
            else if(!isdigit(str1[i])) return IS_LESSER;
            else continue;
        }
    }

    return IS_EQUAL;
}

doc* create_doc(const char doc_name[MAX_WORD_SIZE]){
    doc* doc_head = (doc*)malloc(sizeof(doc));
    strcpy(doc_head->doc_name, doc_name);
    doc_head->next_doc = NULL;
    return doc_head;
}

void append_doc(node* elem, char doc_name[MAX_WORD_SIZE]){
    doc* doc_tmp = elem->docs;
    while(doc_tmp != NULL){
        if(strcmp(doc_tmp->doc_name,doc_name) == IS_EQUAL)
            return;
        if(doc_tmp->next_doc != NULL) doc_tmp =
doc_tmp->next_doc;
        else break;
    }
    doc_tmp->next_doc = create_doc(doc_name);
}
```

```
node* create_node(const char str[MAX_WORD_SIZE],const char
doc_name[MAX_WORD_SIZE]){
    node* temp = (node *)malloc(sizeof(node));

    temp->prev = NULL;
    strcpy(temp->word,str);
    temp->docs = create_doc(doc_name);
    temp->next = NULL;

    return temp;
}
void insert_front(node** head, char str[MAX_WORD_SIZE],char
doc_name[MAX_WORD_SIZE]){
    node* temp = create_node(str, doc_name);
    (*head)->prev = temp;
    temp->next = *head;
    *head = temp;
}

void insert_after(node* elem, char str[MAX_WORD_SIZE],char
doc_name[MAX_WORD_SIZE]){
    node* temp = create_node(str,doc_name);
    temp->next = elem->next;
    temp->prev = elem;
    if(temp->next!=NULL) temp->next->prev = temp;
    elem->next = temp;

    return;
}

void place_elem(node** head, char
str[STR_ARRAY_LEN][MAX_WORD_SIZE], char
doc_name[MAX_WORD_SIZE]){
    char duplicate[MAX_WORD_SIZE] ="\0";
    strcpy(duplicate,str[0]);

    int start_index = 0;

    while(str[start_index][0] == SENTINAL_CHAR)
        start_index++;
```

```
    if(*head == NULL){
        *head = create_node(str[start_index],doc_name);
        node* curr = *head;
        for(int i=start_index+1; i<STR_ARRAY_LEN; i++){
            if(comp_strings(duplicate, str[i]) != IS_EQUAL){
                insert_after(curr,str[i],doc_name);
                strcpy(duplicate,str[i]);
                curr = curr->next;
            }
            else continue;
        }
    }
    else{
        if(comp_strings(str[start_index],(*head)->word) ==
IS_LESSER)
            insert_front(head,str[start_index++],doc_name);
            node* curr = *head;

            for(int i=start_index; i<STR_ARRAY_LEN; i++){
                while(comp_strings(str[i],curr->word) ==
IS_GREATER){
                    if(curr->next != NULL)
                        curr = curr->next;
                    else break;
                }
                if(comp_strings(str[i],curr->word) ==
IS_GREATER)
                    insert_after(curr,str[i],doc_name);
                else if(comp_strings(str[i],curr->word) ==
IS_EQUAL)
                    append_doc(curr,doc_name);
                else insert_after(curr->prev,str[i],doc_name);
            }
    }
    return;
}

void print_dict(node* head){
    while(head!= NULL){
        for(int i=0; i<MAX_WORD_SIZE; i++)
            if(head->word[i]!=SENTINAL_CHAR)
printf("%c",head->word[i]);
```

```
            else printf(" ");
        printf("\t");

        doc* tmp = head->docs;
        while(tmp != NULL){
            printf("<---->%s",tmp->doc_name);
            tmp = tmp->next_doc;
        }
        printf("\n");
        head = head->next;
    }
    return;
}

void i_STR_list(char str_arr[STR_ARRAY_LEN][MAX_WORD_SIZE]){
    for(int i=0; i<STR_ARRAY_LEN; i++){
        for(int j=0; j<MAX_WORD_SIZE-1; j++)
            str_arr[i][j] = SENTINAL_CHAR;
        str_arr[i][MAX_WORD_SIZE-1] = '\0';
    }
    return;
}

void dict_add_from(char doc_name[MAX_WORD_SIZE], node** head){
    FILE* fp;
    fp = fopen(doc_name,"r");
    if(fp == NULL){
        printf("No such file exists, skipping....\n");
        return;
    }

    format_doc(fp,"formatted");
    fclose(fp);

    fp = fopen("formatted", "r");

    char str[MAX_WORD_SIZE] = "\0";
    char str_arr[STR_ARRAY_LEN][MAX_WORD_SIZE];
    int i = 0;
    int fread_c = 0;
    i_STR_list(str_arr);
```

```
    /* when we try to control the loop with feof(fp), *
     * IT DOESN'T WORK!!!!!! Hence do the same with   *
     * fread()                                         */

    while((fread_c =
fread(str,sizeof(char),MAX_WORD_SIZE,fp))){
        strcpy(str_arr[i%STR_ARRAY_LEN], str);
        i++;

        if(i%STR_ARRAY_LEN == 0 && i!=0) {
            sort_strings(str_arr);
            place_elem(head, str_arr, doc_name);
            i_STR_list(str_arr);
        }
    }


/**************************************************************
*******
    * It may happen that the str_arr is not empty neither is a
multiple *
    * of STR_ARRAY_LEN however the file has came to end of it.
Thus,    *
    * then in that case, we will not be adding the elements in
str_arr  *
    * to the dict we're creating. Taking care of that corner
case       *

**************************************************************
*******/

    if(i%STR_ARRAY_LEN != 0 && i!=0) {
        sort_strings(str_arr);
        place_elem(head, str_arr, doc_name);
        i_STR_list(str_arr);
    }

    //print_dict(*head);
    return;
}

void search_rec(char query[MAX_WORD_SIZE], node* head){
```

```
    static node* prev_req = NULL;
    if(prev_req == NULL) prev_req = head;

    char formatted_q[MAX_WORD_SIZE];
    int encountered = 0;
    for(int i=0; i<MAX_WORD_SIZE-1; i++){
        if(query[i] == '\0') encountered = 1;

        if(!encountered) formatted_q[i] = query[i];
        else formatted_q[i] = SENTINAL_CHAR;
    }

    if(comp_strings(formatted_q, prev_req->word) ==
IS_LESSER){
        while(comp_strings(formatted_q, prev_req->word) !=
IS_EQUAL){
            if(prev_req->prev != NULL)
                prev_req = prev_req->prev;
            else{

printf("______________________________________________________
\n");
                printf("Your query cannot be found in DB\n");

printf("______________________________________________________
\n");
                return;
            }
        }
    }
    else if(comp_strings(formatted_q, prev_req->word) ==
IS_GREATER){
        while(comp_strings(formatted_q, prev_req->word) !=
IS_EQUAL){
            if(prev_req->next != NULL)
                prev_req = prev_req->next;
            else{

printf("______________________________________________________
\n");
                printf("Your query cannot be found in DB\n");
```

```
printf("______________________________________________________
\n");
                return;
            }
        }
    }

    doc* temp = prev_req->docs;
    int i=1;

printf("______________________________________________________
\n");
    printf("%s\n",query);
    while(temp != NULL){
        printf("%d. %s\n",i++,temp->doc_name);
        temp = temp->next_doc;
    }

printf("______________________________________________________
\n");
}


int main(int argc, char **argv){
    node* head = NULL;
    for(int i=1; i<argc; i++){
        dict_add_from(argv[i], &head);
    }

    print_dict(head);

    search_rec("fox", head);
    search_rec("will", head);
    search_rec("me", head);
    return 0;
}
```

8. Write an application for adding, subtracting and multiplying very large numbers using linked lists.

Solution Approach

Store the numbers digit by digit in the linked list. And then multiply each digit of the linked list with the remaining nodes of the linked list to get the final result

Algorithm

```
input the two numbers x1, x2
node *head1, *head2 are the two linked lists
subroutine: insertNumber(head, x)
     for(each number : x)
          while(x != 0)
               dig = x%10
               insert(head, x)
               x = x/10
subroutine: insertNumberAtEnd(head, num)
     while(!at_end(head))
          head = head->next
     head->ele = head->ele + mult % 10
     mult = mult/10
     while(mult != 0)
          head->next = new Node(mult%10)
          mult = mult/10
insert(head1, x1)
insert(head2, x2)
node *head3 = final result
while(!empty(head1))
     ele1 = head1->ele
     while(!empty(head2))
          ele2 = head2->ele
          mult = ele1 * ele2
          insertNumberAtEnd(head3, mult)
```

Solution Approach

Add corresponding list values to each other. In case of the carry, add it to the next node.

Algorithm

```
x1, x2 be the numbers
head1, head2 are the corresponding lists for the two
numbers
insertNumber(head1, x1)
insertNumber(head2, x2)//subroutine calls
head3 is the third list
head3->ele = 0
while(!empty_l(head1) && !empty_l(head2))
     int element = head1->ele + head2->ele
     head3->ele += element%10
     if(element/10 == 0)
          head3 = head3->next
          continue
     else
          head3 = head3->next
          head3->ele = element/10

     head1 = head1->next
     head2 = head2->next
if(!empty_l(head1))
     head3->ele += head1->ele%10
     if(head1->ele/10 == 0)
               head3 = head3->next
               continue
          else
               head3 = head3->next
               head3->ele = element/10
if(!empty(head2))
     //same as with head1
```

Program Approach for subtracting two numbers

Subtract 9 from all the numbers of the subtractend. Then add the numbers. If you have a carry at the end, then the result is the final list without, the last node. Else, the answer is the negative of the list value

Algorithm

```
int x1, x2
node* head1, head2 be the two linked list
insert(head1, x1)
insert(head2, x2)
//we are going to do head1 - head2
while(!empty_l(head2))
     head2->ele += 9
     head2 = head2->next
int size1 = size_l(head1)
head3 = add(head1, head2)
if(size_l(head3) == size1)
     >>answer is negative of the list value
else
     while(at_end(head->next))
          head = head->next
     delete_after(head)
     >>answer is the list value
```

9.Given two polygons, find out whether they intersect or not.

Program Approach:

The underlying concept for this problem is to devide a polygon into some number of triangles and add up their area. If two polygons intersect there must be a vertex of one polygon lying into the bonundary of the other polygon. we connect all the other vertices of the other polygon with that particular points. Hence the other polygon get devided into n number of triangles(Where n is the nunber of sides of that polygon).
If we add up the areas of the triangles it should be equal to the area of the actual polygon.Otherwise the areas won't match up.

Algorithm:

```
typedef struct point_tag
{
```

```
     float x;
     float y;
     struct point_tag* next;
}point;

begin:
     point* create(float x, float y)
          point* p = allocate memory for a point objecte
          p->x = x
          p->y = y
          p->next = NULL
     int ret_sign(point* p, point* p1,point* p2)
          f = 0
          f = (p->y - p1->y)*(p2->x - p1->x) - (p->x -
p1->x)*(p2->y - p1->y)
          if f > 0
               return 1
          else return 0
     int check_inside(poimt *p, point* p_head)
          check_in = 1
          point* prev = p_head
          point* curr = p_head->next
          if prev = NULL and curr = NULL curr->next = NULL
               return 1
          do
               check_in = check_in * ret_sign(p,prev,curr)
               prev = curr
               curr = curr->next
          while prev = p_head
          end while
          return check_in
          if  check_in = 1
               print INTERSECTED
          else
               print NOT INTERSECTED
end
```

Assignment 4A

1. Implement a greedy function to implement the coin change problem with coins of denominations 1p, 5p, 10p, 25p and 50p. You need to minimize the number of coins for a change. The input to the function is the amount in paise to be changed and the output is a string containing the denominations and the numbers against each denomination. Call the function from the main program. How would test the correctness of your program?

Solution Approach

1. Program a function **greedy(int amount)**,which calculates the minimal number of coins required for the change **amount** passed as a parameter.
2. In the function we pass an array containing the coin denominations **denominations[] = {50, 25, 10, 5, 1},** and we execute a while loop over this array thereby implementing the greedy algorithm.

Algorithm

Greedy Algorithm

```
greedy(amount)
Begin
  denominations[] = {50, 25, 10, 5, 1};
  number[5] = {0}; size = 5; i = 0;
  while (i < size) do
    while(amount >= denominations[i]) do
      amount = amount - denominations[i];
      number[i]++;
    end while
    i++;
end while
  for (i = 0; i < size; i ++)
          display (denomination, number of coins)

End
```

Sample Output

```
greedy(76)
Rs. 50 - 1, Rs. 25 - 1, Rs. 10 - 0, Rs. 5 - 0, Rs. 1 - 1
```

Discussions

Greedy algorithm makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution. Thus, although may not give us the correct solution, it will lead to a solution close to the correct solution. This algorithm helps in getting to a ‘good-enough’ solution faster than other techniques.

To test the correctness, we have to use the technique of Random Testing in which we implement our algorithm and also, implement a reference algorithm that we know to be correct (e.g., one that exhaustively tries all possibilities and takes the best). In certain number of cases, Random Testing will tell us the number of times our Greedy Algorithm was correct or incorrect.

----

2. Write a program to solve the n-queens problem. Choose the data structure you will use to solve the problem. Run the program for 100, 1000 and 10000 queens. Note the time required in each case.

Solution Approach

1. Make a caller function solveNQ() which calls a recursive function solveNQUtil() which will go for next column having a column fixed in the previous stage the safety of a room is checked by isSafe() function
2. If there is no safe place in the column then backtrack and for the next place and then again that column
3. If all columns are covered then it has a solution, otherwise no solution available

Algorithm

1) Start in the leftmost column

```
2) If all queens are placed
    return true
3) Try all rows in the current column.
   Do following for every tried row.
    a) If the queen can be placed safely in this row
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing the queen in [row, column] leads to
       a solution then return true.
    c) If placing queen doesn't lead to a solution then
       unmark this [row, column] (Backtrack) and go to
       step (a) to try other rows.
3) If all rows have been tried and nothing worked,
   return false to trigger backtracking.
```

Sample Output

```
N = 10
     1 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 1 0 0
     0 1 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 1 0
     0 0 0 0 0 1 0 0 0 0
     0 0 1 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 1
     0 0 0 1 0 0 0 0 0 0
     0 0 0 0 0 0 1 0 0 0
     0 0 0 0 1 0 0 0 0 0
```

Discussion

The time taken for the program for a 10x10 grid is 0m0.019s. The time for 100x100 is _ (tested for 14m8.582s with no answer) . The time for 1000x1000 is _ (isnt brave enough to run on local machine). So as we can see the time complexity rises exponentially.

----

3. Write a program to solve the "Rat-in-a-Maze" problem for various dimensions of the maze. How do you propose to present the output for a large maze?

Solution Approach

1. Make a caller function solveMaze() which calls a recursive function solveMazetUtil() which will go horizontally and if this does not lead to a solution then backtrack and move down
2. If there is no solution print no solution

Algorithm

If destination is reached
print the solution matrix
Else
a) Mark current cell in solution matrix as 1.
b) Move forward in the horizontal direction and recursively check if this
move leads to a solution.
c) If the move chosen in the above step doesn't lead to a solution
then move down and check if this move leads to a solution.
d) If none of the above solutions works then unmark this cell as 0
(BACKTRACK) and return false.

Discussion

This is also a recursive search and so exponentially increasing. The output will be got exhaustively and for any path not leading a solution will result in backtracking and proceed. For a large output i propose a linked list with the cell numbers attached. Like- start ---> (1,2) →(1,3).... etc

Output

For input
1 0 0 0
1 1 0 1

```
0 1 0 0
1 1 1 1
```

Output

```
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1
```

Using linked list

(1,1) → (2,1) → (2,2) → (3,2) → (4,2) → (4,3) → (4,4)

---

4. Implement the following sorting algorithms with their different variants for files of integer and strings:

Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort and Heap sort.

Tabulate the timing analysis data as discussed in the lab and plot the curves with the timing data to ascertain the time complexity of the algorithms.

Algorithm

Bubble sort

```
Sorted = 0
If not sorted
Sorted = 1
Loop from i = 1 to n-2
     If arr[i] > arr[i+1]
          swap(arr+i, arr+i+1)
          Sorted = 0
```

Insertion sort

```
Loop from i = 1 to n-1.
```

a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Selection sort

```
Loop from i = 1 to n-1
       a) Pick the max in arr[0...n-i] swap with a[n-i]
```

Merge sort

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
          middle m = (l+r)/2
     2. Call mergeSort for first half:
          Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
          Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
          Call merge(arr, l, m, r)
```

Quick sort

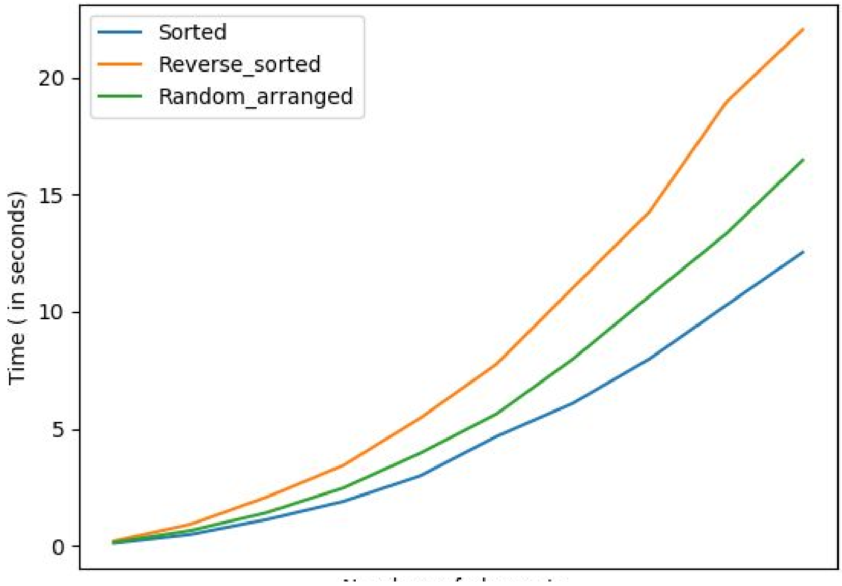
```
quickSort(arr[], low, high)
    if (low < high)
        pi = partition(arr, low, high);
             quickSort(arr, low, pi - 1);  // Before pi
             quickSort(arr, pi + 1, high); // After pi
```

Heap sort

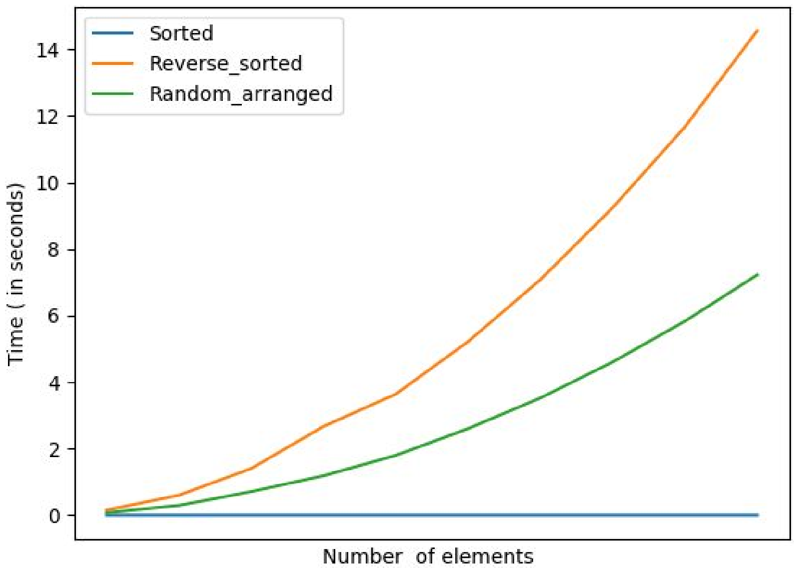
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than

Discussion

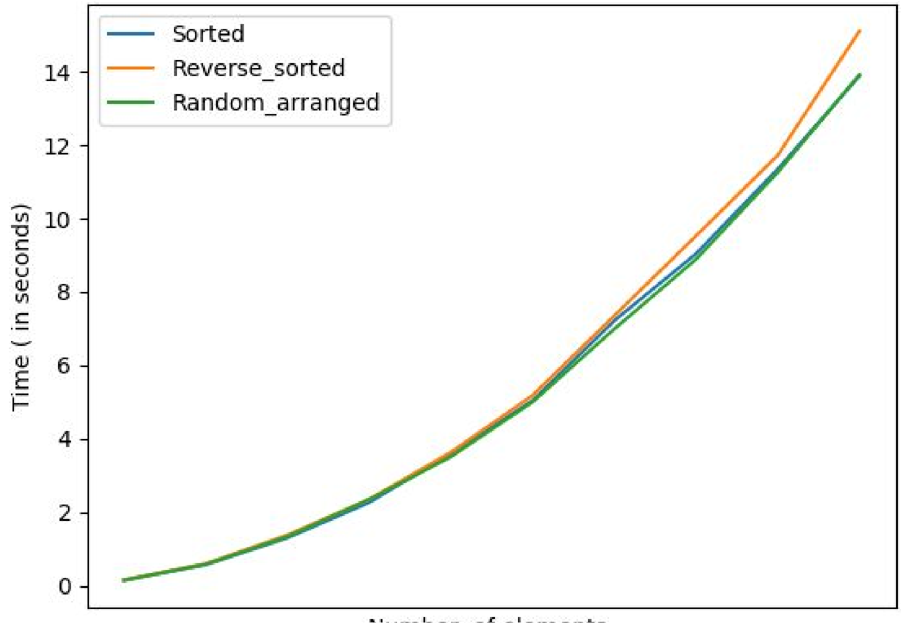
- The worst case scenario for **bubble sort** is O(n^2) the best case scenario is O(n) and space complexity is O(1).



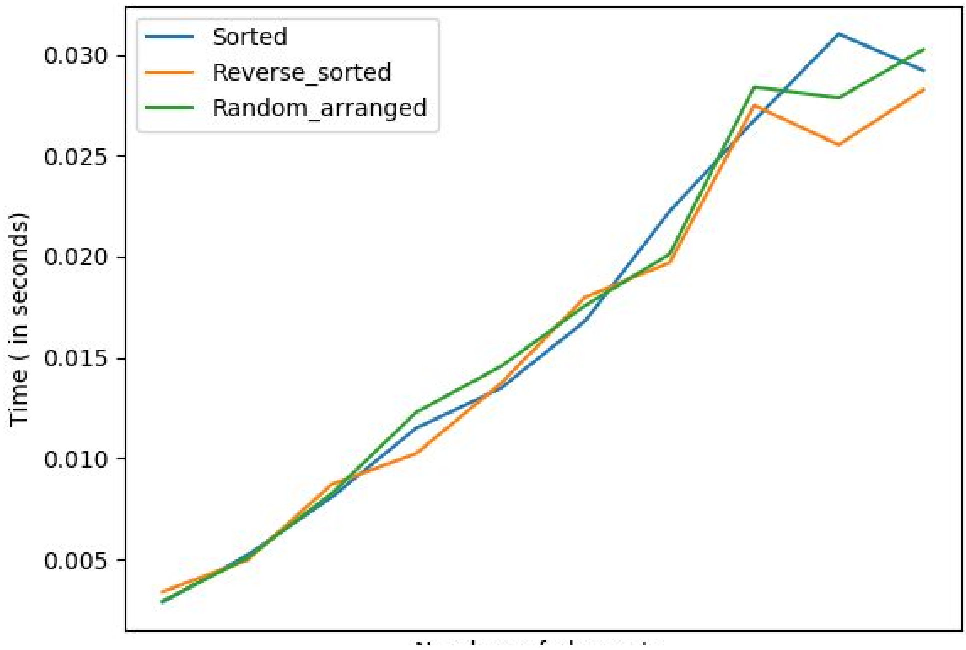
- The **insertion sort** takes O(n^2) in the worst case scenario. O(1) space and O(n) in the minimum case when the array is sorted.



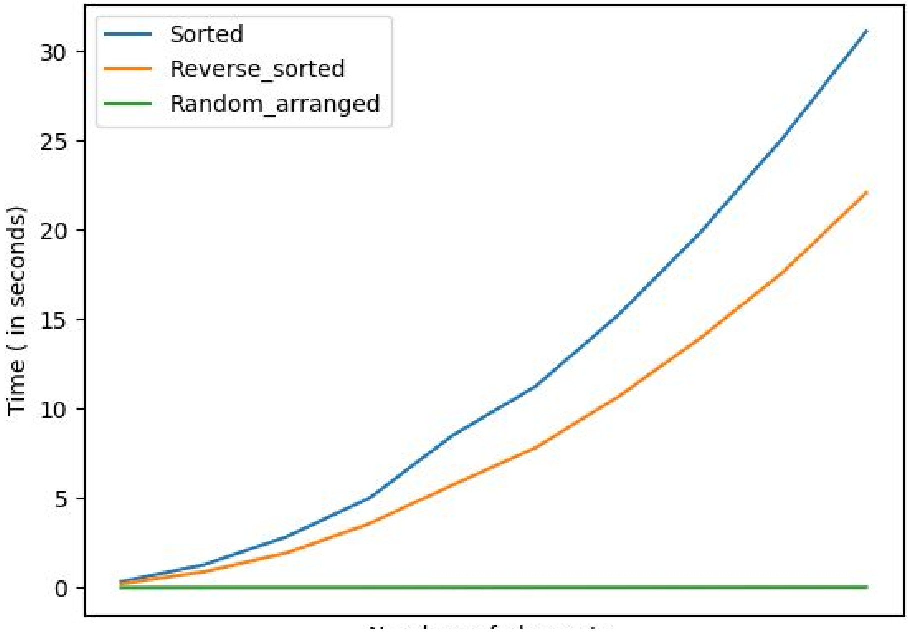
- The **selection sort** takes O(n^2) in the worst case scenario. O(1) space and it does not take more than O(n) swaps so it is useful when memory write is costly.



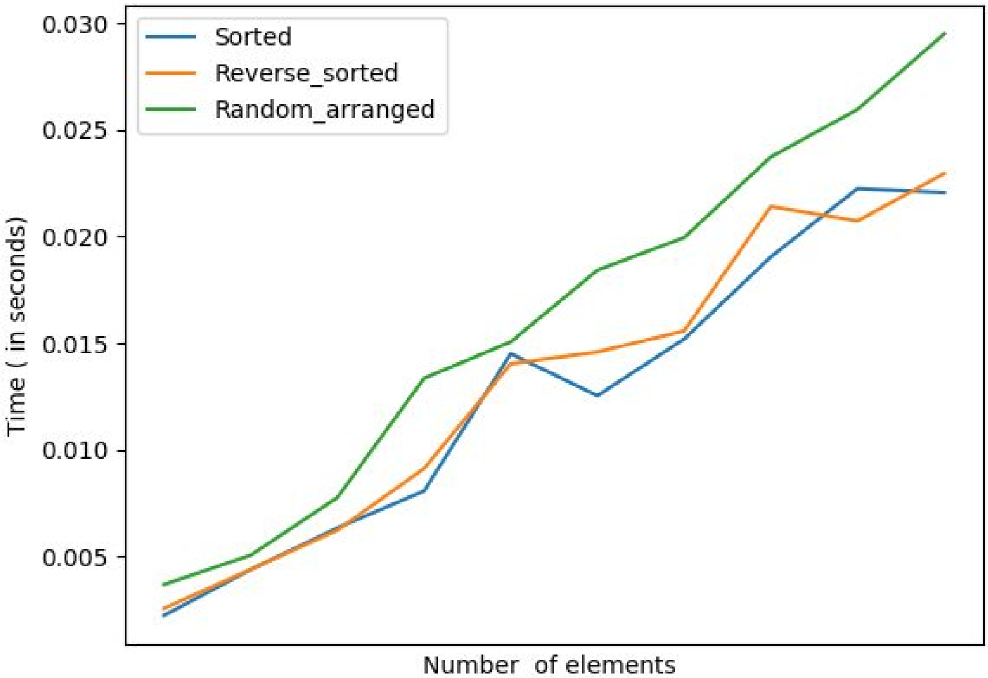
- The **merge sort** takes O(nlogn) in all scenario. It takes O(n) auxiliary space and it utilizes divide and conquer approach.



- The worst case scenario for **quick sort** is O(n^2) and average and best case scenario is O(nlogn).



- In **heap sort** the time complexity for heapify is O(logn) the time complexity to build heap is O(n) overall time complexity O(nlogn).



----

5. a. Implement the ADT Stack using array and linked list in separate header files containing support for the following :-

1. Print the elements of the stack from top to bottom.
2. Print the elements bottom to top.
3. Check whether two stacks are equal or not.

Solution Approach

1. **create (Stack **S)** - Creates the stack
2. **isEmpty(Stack *S)** - Checks whether the stack is empty or not
3. **isFull(Stack *S)** - Checks whether the stack is full or not
4. **getTop(Stack *S)**- Gets the value placed on top of the stack
5. **display(Stack *S)** - Displays the stack from top to bottom
6. **displayReverse(Stack *S)** - Displays the stack from bottom to top
7. **push(Stack *S, int element)** - Pushes an element to the stack
8. **pop(Stack *S)** - Pops an element from the stack
9. **checkEquality(Stack *S1, Stack *S2)** - Checks whether the two stacks are equal or not

Algorithm

```
typedef struct Node
{
  int data;  struct Node* next;
} Node;

typedef struct Stack
{
  Node* top;   int nodes;
} Stack;

create (Stack **S)
Begin
  *S = allocate_memory()
  if (*S == NULL) then
```

```
    display(error);
    exit(0);
  end if
  (*S)->top = NULL; (*S)->nodes = 0;
End

isEmpty(Stack *S)
Begin
  return (S->top == NULL);
End

isFull(Stack *S)
Begin
  return (S->nodes >= MAX_LENGTH);
End

getTop(Stack *S)
Begin
  return (S->top)->data;
End

display(Stack *S)
Begin
  if (isEmpty(S)) then
    display("Stack is Empty");
    return;
  end if
  Node* cur = S->top;
  while(cur != NULL) do
    display(node_data);
    cur = cur->next;
  end while
End

show_reverse(Node *node)
Begin
  if (node->next == NULL) then
    return;
  else
    node = node->next;
    show_reverse(node);
    display(node_data);
```

```
  end if
End

display_reverse(Stack *S)
Begin
  if(isEmpty(S)) then
    display(“Stack is Empty”);
    return;
  else
    show_reverse(S->top);
    display(Top Node Data);
  end if
End

push(Stack *S, int element)
Begin
  if (isFull(S)) then
    display(“Stack is Full”);
    return;
  else
    Node *new = allocate_memory();
    new->data = element;
    new->next = S->top;
    S->top = new;
    (S->nodes)++;
  end if
End


pop(Stack *S)
Begin
  if(!isEmpty(S)) then
    Node *cur = S->top;
    int val = cur->data;
    S->top = cur->next;
    (S->nodes)--;
    free(cur);
    return val;
  else
    display(“Stack is Empty”);
    return NONE;
End
```

```
checkEquality(Stack *S1, Stack *S2)
Begin
  if (isEmpty(S1) || isEmpty(S2))
    display("Stack is Empty");
    return -1;
  if (S1->nodes != S2->nodes)
    display("Stack is not of Equal Length");
    return -1;
  flag = 1; Node* cur1 = S1->top;
  Node* cur2 = S2->top;
  while(cur1 != NULL || cur2 != NULL) do
    if (cur1->data != cur2->data) then
      flag = 0; break;
    end if
    cur1 = cur1->next; cur2 = cur2->next;
  end while
  return flag;
End
```

5. b. Using the above header files, implement the following :-
   1. Conversion of Infix to Postfix Expressions.
   2. Evaluation of Postfix Expressions.

Solution Approach

1. We implement a function **prec(char c)** which checks for the order of precedence of operators and gives a value **true** or **false** whether the order is fulfilled or not.
2. Program a function **infixToPostfix(char s[])** which does the necessary conversion from infix expressions to postfix expressions.
3. Program a function **evaluatePostfix(char exp[])** which does the required evaluation for the postfix expressions.

Algorithm

```
int prec(char c)
Begin
  if(c == '^') then
```

```
    return 3;
  else if(c == '*' || c == '/') then
    return 2;
  else if(c == '+' || c == '-') then
    return 1;
  else
    return -1;
  end if
End

infixToPostfix(char s[])
Begin
  Stack *S; create(&S);
  push(S, '$');
  l = length(s);
  ctr = 0;



  for(int i = 0; i < l; i++) do
     if((s[i] >= 'a' && s[i] <= 'z')||(s[i] >= 'A' && s[i] <=
'Z')) then
        output_string[ctr++] = s[i];
      else if(s[i] == '(') then
        push(S, '(');
      else if(s[i] == ')') then
        while(getTop(S) != '$' && getTop(S) != '(')
          c = getTop(S);
          pop(S);
          output_string[ctr++] = c;
        end while
        if(getTop(S) == '(') them
          c = getTop(S);
          pop(S);
        end if
    else
        while(getTop(S) != '$' && prec(s[i]) <=
prec(getTop(S)))
          c = getTop(S);
          pop(S);
          output_string[ctr++] = c;
         push(S, s[i]);
```

```
  end for
  while(getTop(S) != '$')
    char c = getTop(S);
    pop(S);
    output_string[ctr++] = c;
  output_string[ctr] = '\0';
  display(output_string);
End
```

```
evaluatePostfix(char exp[])
Begin
  Stack *S; create(&S);
  for (i = 0; exp[i]; ++i) do
    if (isdigit(exp[i])) then
      push(S, exp[i] - '0');
    else
      val1 = pop(S); val2 = pop(S);
      switch (exp[i])
        case '+': push(S, val2 + val1); break;
        case '-': push(S, val2 - val1); break;
        case '*': push(S, val2 * val1); break;
        case '/': push(S, val2/val1); break;
    end if
  return pop(S);
End
```

Sample Output

```
infixToPostfix("a+b*(c^d-e)^(f+g*h)-i")
Ans: abcd^e-fgh*+^*+i-

evaluatePostfix("231*+9-")
Ans: -4
```
