**ASSIGNMENT 3**

Name: IMON RAJ

Class Roll: 002010501098

Class: BCSE II

Sem: First

Session: 2021-2022

**PROBLEM 1:**

**Problem statement:**

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

**init\_l(cur)** – initialize 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 :**

Linked list is a very popular data structure. It is a linear chain-like data structure with nodes connected to each other by pointer referencing. We are going to implement it in C.

A node has a data value and a reference to the next node

**Node { data, next }**

* **Initializing a list** is nothing but creating a head with NULL value.
* At any point if head is NULL then the list is said **empty.**
* If a node’s next reference is NULL that indicates that it is the last element or **At End Node.**
* **Insert Front** - actually inserts a node in the front of the list means before the old head. In this case we have to take care of the head’s value because it is going to change.
* **Insert After** – inserting a node after a specified node provided the prev node is not NULL.
* **Delete Front** – Here we are going to delete the old head element and update the head’s value to the closest next element.
* **Delete After** – Deleting the next node of a specified node, the previous node must not be NULL.

While implementing all these operations pointer operations are done carefully. Also when we delete a node its our responsibility to free the allocated memory for that node(No garbage collection system in C).

**Structured Pseudocode:**

**init\_l() {**

**return NULL**

**}**

**atend\_l(node) {**

**return (node->next == NULL);**

**}**

**empty\_l(head) {**

**return (head == NULL);**

**}**

**insert\_front(phead, target){** //phead is address of head

**target->next = \*phead;**

**\*phead = target;**

**}**

**insert\_after(prev, target){  
 target->next = prev->next;**

**prev->next = target;**

**}**

**delete\_front(phead){**

**temp = \*phead;**

**\*phead = \*phead->next;**

**free\_memory(temp);** // We can also return( actually done in linkedlist.h ) the node and delete elsewhere( in main program )

**}**

**delete\_after(prev){**

**temp = prev->next;**

**prev->next = temp->next;**

**free\_memory(temp);** // We can also return( actually done in linkedlist.h ) the node and delete elsewhere( in main program )

**}**

**Results:**

All the above pseudocodes are implemented in header file <linkedlist.h> . This header file is going to be used in other programs according to our needs. There is no overhead of writing the linked list main operations again. And all the other secondary operations on linked list can be implemented by using these basic functions. We can add this header file to our machines complier’s includes directory to use it globally in our machine from any path.

/\*\*\*\*AS THIS IS THE HEADER FILE THERE IS NO OUTPUT TO BE GIVEN. OUTPUTS ARE THERE WHEN WE IMPLEMENT THE FUNCTIONS\*\*\*\*/

**Discussion:**

While coding the linked list operations I faced bugs in pointer arithmetic and solved that issue. Sometimes I got into an infinite loop due to not updating the pointers value. After handling or debugging and solving these problems as a programmer my efficiency in writing clean and perfect programs increased.

**PROBLEM 2:**

**Problem statement:**

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 by developing a function Print\_list. The functions for (a) Build\_list and for (b) is Build\_list\_reverse.

**Solution Approach:**

For this problem, at the beginning we will naturally open the file and also initialize a linked list. Now we will scan the file for all the integers in it and one by one we will insert them in the list. Now, on the basis of insertion position two situations will come –

* **Build\_list\_reverse**: For each integer we scan from the file we will insert it at the front of the linked list. After insertion ends, we will get a reversed list compare to file’s data sequence.
* **Build\_list**: In this case we will always maintain a pointer at last of the linked list, and for each integer we will insert it after that last element. So in this case we get a list which has values in the same order of the file.

And **print\_list** is basically traversing the linked list just created and print its values sequentially.

**Structured Pseudocode:**

Node \* Build\_list\_reverse() {

FILE \* fp = fopen(FILENAME);

Node \* head = init\_l();

int val;

scan\_int\_from\_file(fp, val); /\*JUST SCAN AN INTEGER VALUE FROM FILE\*/

head = createnode(val);

while(scan\_int\_from\_file(fp, val)) {

Node \* newNode = createnode(val);

insert\_front(newNode, &head);

}

fclose(fp);

return head;

}

Node \* Build\_list() {

FILE \* fp = fopen(FILENAME);

Node \* head = init\_l();

int val;

scan\_int\_from\_file(fp, val); /\*JUST SCAN AN INTEGER VALUE FROM FILE\*/

head = createnode(val);

Node \* ptr = head;

while(scan\_int\_from\_file(fp, val)) {

Node \* newNode = createnode(val);

insert\_after(newNode, ptr);

ptr = ptr->next;

}

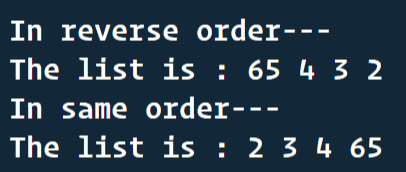
fclose(fp);

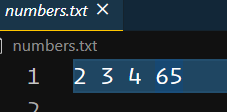
return head;

}

**Results:**

In the file there are integers and after printing the lists created by the Build\_list and Build\_list\_reverse from the file’s data, we can see that the same integers are printed in same and reverse sequence.





**Discussion:**

This program helps to understand that inserting elements in a linked list at front or at last gives different outcomes. This program needs some amount of file handling which is an important concept.

**PROBLEM 3:**

**Problem statement:**

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 is 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 :**

For this problem we have to implement various operations on linked list using the basic operations defined in the header file. In a C file we will use those newly defined operations in a menu-driven system. Let’s see the approaches followed for all those operations –

a ) Print list in same order & reverse order: For printing the list in same order we will use basic function **print\_l() from the header file.** For printing the list in reverse order we will use recursive approach in such a way that we will first call the **print\_list\_in\_reverse() function recursively** for the next node of current node and then only we will print current node’s value. The base case will be when node becomes NULL. We will call this recursive function for head node, in output reverse list will be printed.

b ) Find size of list in number of nodes: We will have a count variable initialized to 0. We will traverse the list from head until we get to a NULL value and for each Non-NULL node we will increase the count by 1 and finally return the count which is basically the size of list.

c) check whether two lists are equal: For two list we will traverse them simultaneously, each and every node of one list should be equal to the other in the same order of other and also size should also be same. So while traversing if there is any unequal node we will return false, traversal of one list ends before the other(i.e – not equal size) we return false. Otherwise, lists are equal return true. If both lists are empty return true.

d ) Search key in unordered and ordered list: Let’s say we want to find a value X in a linked list. If the linked list is ordered, we will traverse the list if we find any node’s data equal to X then we will return the node and conclude that element found, also in each iteration for ordered list we will check whether the node’s data gets bigger than X if yes then we will conclude not found. But in case of unordered list we have to scan all the elements of the list and compare with X, in case we can not find the X in list after complete traversal of the list then only we can conclude that not found.

e ) Append a list at the end of another list: If the heads of two lists are head1 and head2, then we will completely traverse the first list and then we will set the last node’s next of the first list to head2 and finally we got a list where second list(head2) is appended after first list(head1).

f) delete the nth node, last node and first node of a list: For the deletion of the first node we will just do delete\_front() already defined in the header file. For the deletion of the last node we will traverse until we reach the second last element and then we will do delete\_after(2nd\_last\_node) and last node will be deleted. For the deletion of nth node we will traverse to n-1th node making sure that the list is larger than or equal to n. And then we will delete\_after(n-1th node) and our task is done.

g ) check whether a list is ordered: We will check whether a list is ordered or not maintaining two variables ascending and descending and algorithm is similar to **ASSIGNMENT 2 Question no.-8 the array sorted test**. In this case we will traverse the linked list to the end and apply the same algo as the mentioned question.

h ) Merge two sorted list: This problem’s solution approach will be similar to **ASSIGNMENT 2 Quesiton no.-9 Merge two sorted arrays**. Here we will do the pointer arithmetic in such a way that the output list remains sorted using the similar algo as the above mentioned question.

I ) Insert a target node in the beginning, before a specified node and at the end of the list: To insert a node before a specified target, we have to link the previous node to the new node and new node to the target node. But to insert in the beginning we have to replace the address of head pointer, so we hae to pass the reference of the head pointer.

j ) Remove duplicates from a linked list (sorted and unsorted): To remove the duplicate element of a sorted list remove the same elements except the first one. For unsorted list traverse two pointer. First pointer any element and second one will find and delete the duplicates.

k ) Swap elements of a list pairwise: There will be two pointers one ahead another and they will swap once and incremented twice till end.

l ) Move last element to front of a list: Move a pointer to the last and one to the last but one. Then link the second last node to null and link the last node to the head and set the head address to the current pointer.

m ) Delete alternate nodes of a list: Make to pointer one ahead another. The second pointer will delete the node and then the first node will be incremented once and set the second pointer one ahead of the first one and repeat.14.Delete a list: Use two pointers, first one will point to the node to be deleted and second one will point to the node linked with it. First pointer will free the memory then first pointer will take the value of second pointer and the second pointer will be incremented.

o ) delete a list: For deleting a list we will use the delete\_l() which is already defined in the header file.

p ) Reverse a list: It will take three pointer pointing to three succeeding nodes. Now second will indicate the first and then first will take the value of

second and second will take the value of third and third will be incremented.

q ) Sort a list: We can swap the nodes of a list using two pointers. So now we can apply any sorting algorithm.

**C code :**

Node \* Build\_list() { /\*to create a list\*/

printf("\nHow many elements: ");

int n;

scanf("%d", &n);

printf("\nEnter the elements of the list: ");

Node \* head = init\_l();

int val;

scanf("%d", &val);

head = createnode(val);

Node \* ptr = head;

while(n>1) {

scanf("%d", &val);

Node \* newNode = createnode(val);

insert\_after(newNode, ptr);

ptr = ptr->next;

n--;

}

return head;

}

void print\_list\_in\_reverse(Node \* head) { /\* this is the function for printing the list in reverse \*/

if(head==NULL){

return;

}

print\_list\_in\_reverse(head->next);

printf("%d-->", head->data);

}

int size\_of\_list(Node \* head) { /\*size of a list\*/

Node \* ptr = head;

if(ptr==NULL){

return 0;

}

int count = 0;

while(ptr!=NULL){

count++;

ptr = ptr->next;

}

return count;

}

Node \* delete\_nth\_node(Node \*\* phead, int n) { /\* delete nth node of a list\*/

int i;

Node \* temp, \* ptr;

if((\*phead)==NULL){

printf("\nThe list is empty!!");

return NULL;

}

ptr = \*phead;

if(n==1){

return delete\_front(phead);

}

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

{

if((temp=ptr->next)==NULL){

printf("\nInvalid n!!");

return NULL;

}

ptr = temp;

}

return delete\_after(ptr);

}

Node \* delete\_last\_node(Node \* head){ /\*delete last node of list\*/

Node \* temp;

if(head==NULL){

printf("\nThe list is empty!!");

return NULL;

}

if(atend\_l(head)){

return head;

}

temp = head->next;

while(!atend\_l(temp)){

head = temp;

temp = temp->next;

}

return delete\_after(head);

}

int isList\_ordered(Node \* head) { /\*check if list is ordered \*/

int ascending=0, descending=0, ordered=1;

Node \* ptr = head, \* ptr\_later = head->next;

do{

if(ptr->data > ptr\_later->data){

descending = 1;

} else if(ptr->data < ptr\_later->data){

ascending = 1;

}

if(ascending\*descending) {

ordered = 0;

break;

}

ptr = ptr->next;

ptr\_later = ptr\_later->next;

}while(!atend\_l(ptr));

return ordered;

}

void Reverse\_list(Node \*\* phead){ /\*reverse a list\*/

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

}

Node \* previous = NULL;

Node \* current = \*phead;

Node \* later = \*phead;

while(current!=NULL){

later = current->next;

current->next = previous;

previous = current;

current = later;

}

\*phead = previous;

}

void sort\_list(Node \*\* phead){ /\*to sort a list\*/

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

}

int temp;

Node \* ptr1 = \*phead;

Node \* ptr2 = (\*phead)->next;

while(ptr1!=NULL){

ptr2 = ptr1->next;

while(ptr2!=NULL){

if(ptr2->data < ptr1->data){

temp = ptr2->data;

ptr2->data = ptr1->data;

ptr1->data = temp;

}

ptr2 = ptr2->next;

}

ptr1 = ptr1->next;

}

}

void swap\_element\_pairwise(Node \*\* phead){

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

}

int temp;

Node \* ptr1 = \*phead;

Node \* ptr2 = (\*phead)->next;

while(ptr2!=NULL){

temp = ptr2->data;

ptr2->data = ptr1->data;

ptr1->data = temp;

ptr1 = ptr1->next->next;

if(ptr2->next==NULL){

break;

}

ptr2 = ptr2->next->next;

}

}

void delete\_alternate\_node(Node \*\* phead){

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

}

int temp;

Node \* ptr1 = \*phead;

while(ptr1!=NULL && !atend\_l(ptr1)){

delete\_after(ptr1);

ptr1 = ptr1->next;

}

}

void insert\_before(Node \*\* phead, int val, int val2){ /\*to insert a node before a specified node in list\*/

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

}

Node \* ptr1 = \*phead;

Node \* ptr2 = ptr1->next->next;

Node \* newNode = createnode(val);

while(ptr2 != NULL){

if(ptr1->next->next->data == val2){

insert\_after(newNode, ptr1);

break;

}

ptr1 = ptr1->next;

ptr2 = ptr1->next->next;

}

}

void insert\_last(Node \*\* phead, int val){

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

}

Node \* ptr1 = \*phead;

Node \* newNode = createnode(val);

while(!atend\_l(ptr1)){

ptr1 = ptr1->next;

}

insert\_after(newNode, ptr1);

}

void move\_last\_to\_first(Node \*\* phead){

if((\*phead)==NULL){

printf("\nThe list is empty..\n");

return;

Node \* ptr1 = \*phead, \*ptr2;

while(!atend\_l(ptr1)){

ptr2 = ptr1;

ptr1 = ptr1->next;

}

ptr2->next = NULL;

ptr1->next = (\*phead);

\*phead = ptr1;

}

void check\_equal\_lists(Node \* h1, Node \* h2){

if(h1==NULL || h2==NULL){

printf("\nThe lists are not equal..");

return;

}

Node \* ptr1 = h1, \* ptr2 = h2;

while(ptr1!=NULL && ptr2!=NULL){

if(ptr1->data!=ptr2->data){

printf("\nNot equal lists...");

return;

}

ptr1 = ptr1->next;

ptr2 = ptr2->next;

}

if(ptr1==NULL && ptr2==NULL){

printf("\nEqual lists...");

}else{

printf("\nNot equal lists...");

}

}

void append\_second\_after\_first(Node \* h1, Node \* h2){

if(h1==NULL){

printf("\nThe first list is empty...");

return;

}

Node \* ptr = h1;

while(!atend\_l(ptr)){

ptr = ptr->next;

}

ptr->next = h2;

}

Node \* mergeLinkedList(Node \* head1, Node \* head2){

Node \*resultHead, \*resultTail, \*temp;

resultHead = resultTail = NULL;

while(1){

if(head1 == NULL){

resultTail->next = head2;

break;

}

if(head2 == NULL) {

resultTail->next = head1;

break;

}

/\* Check whether current node of

which Linked list is smaller\*/

if(head1->data <= head2->data){

temp = head1;

head1 = head1->next;

} else {

temp = head2;

head2 = head2->next;

}

/\*Add smaller node to result linked list \*/

if(resultHead == NULL){

resultHead = resultTail = temp;

} else {

resultTail->next = temp;

resultTail = temp;

}

resultTail->next = NULL;

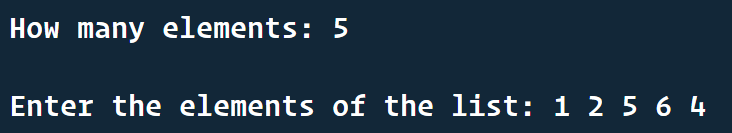
}

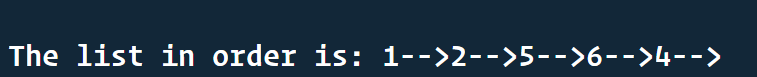
return resultHead;

}

**Results:**

This program takes one list and do some specified operations on it and also it takes two lists and do some operations which involves both of the lists(merging, equality checking, appending etc.).











after sorting ---



after swapping paiwise nodes ---



after moving last element to the front of the list ---



after deleting alternate nodes ---



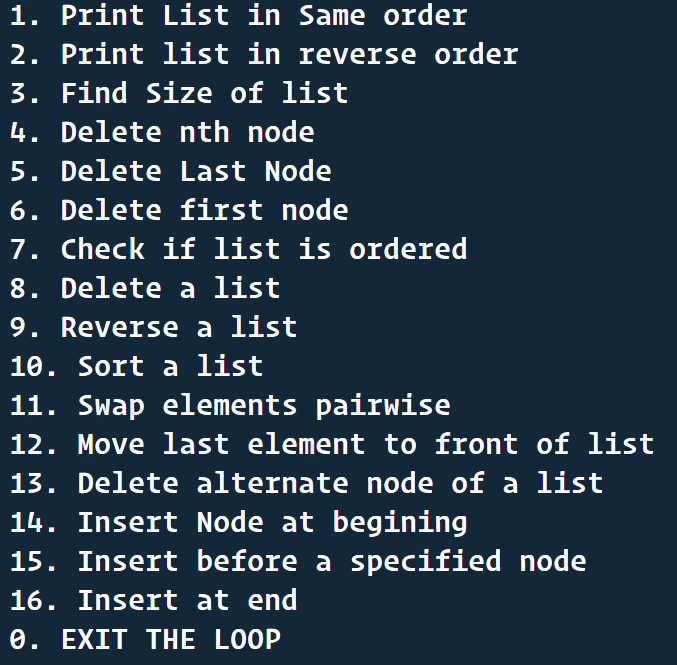
after insert front ---



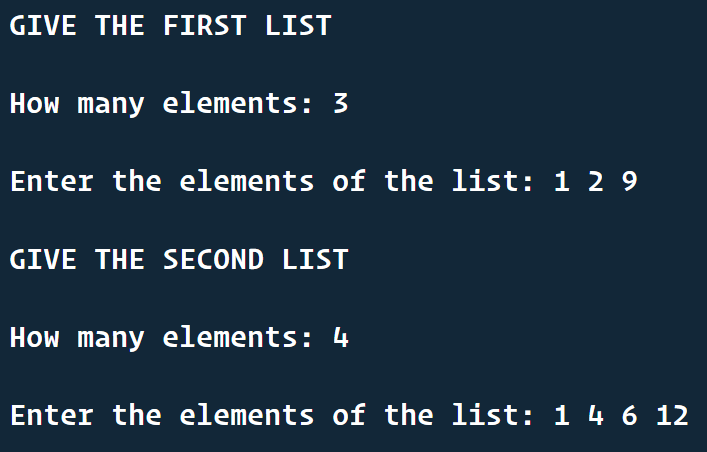
insert end –

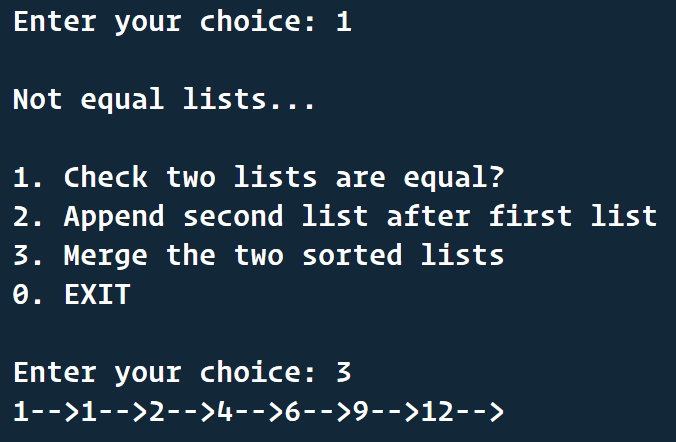


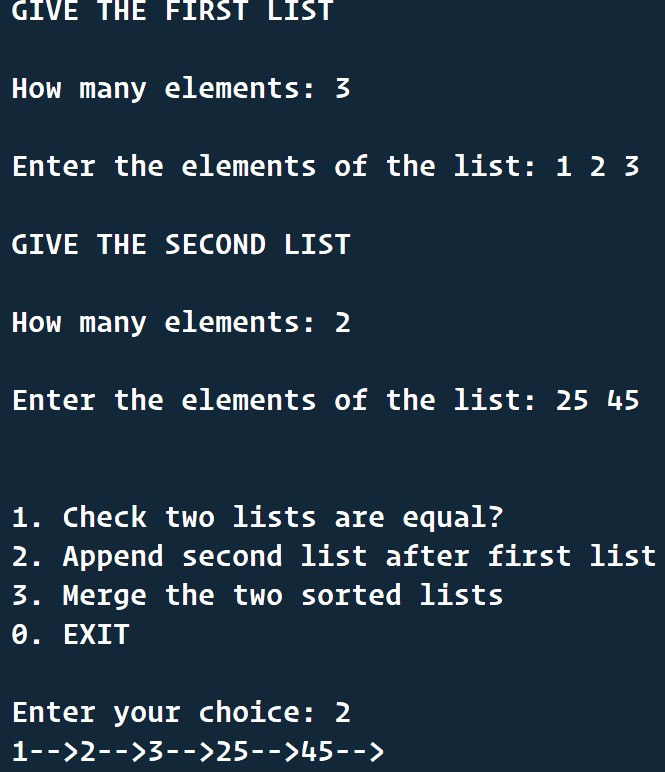
all operations that can be done –



For two linked list operations ---







**Discussion:**

This program helps to create some advanced functions using the simple functions of the header file.

**PROBLEM 4:**

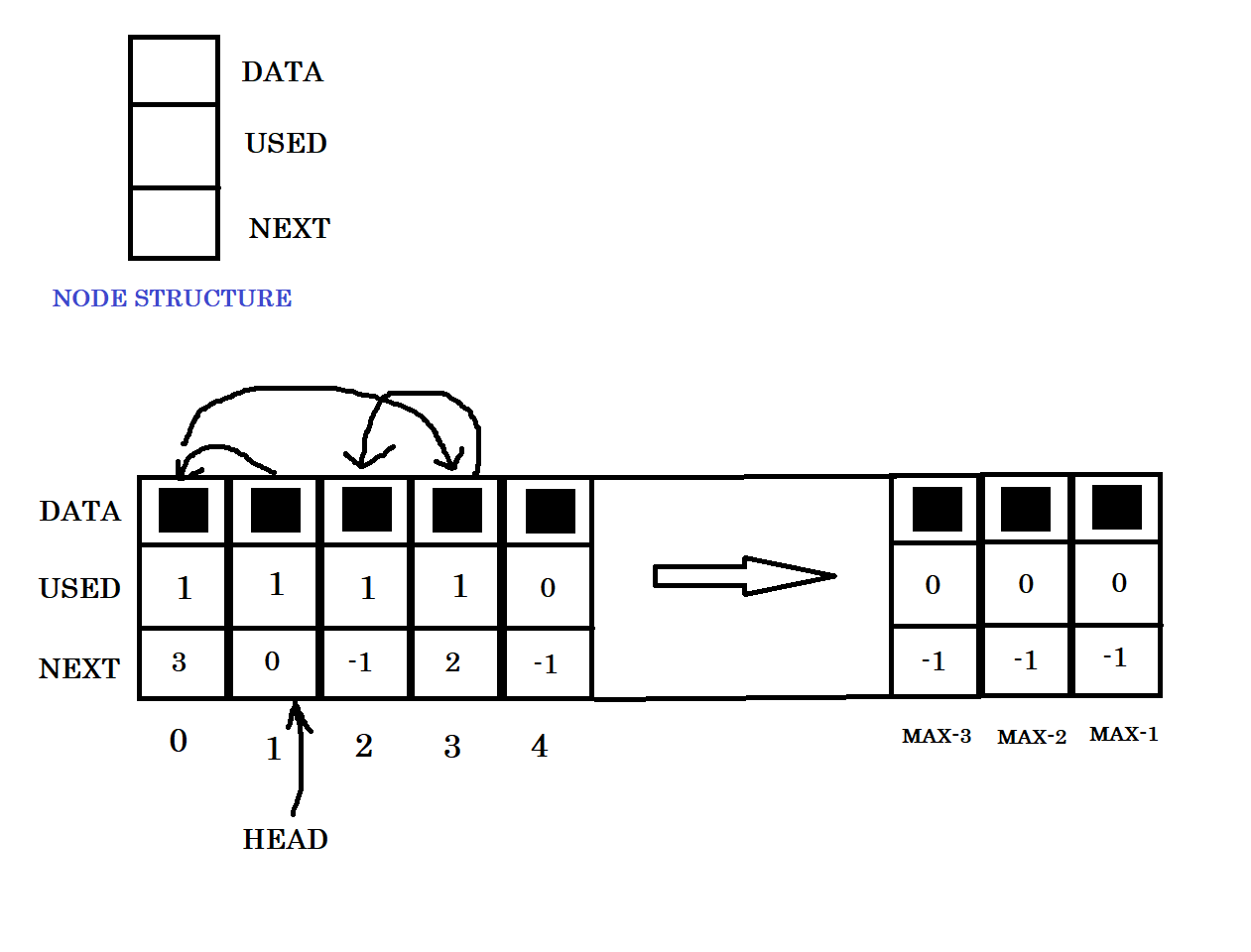
**Problem statement:**

Write all the above operations of Single Linked List for the implementation using array. You need to develop Build\_list and Build\_list\_reverse, as well as Print\_list

**Solution Approach :**

We have to replicate linked list using array. So, we will use an approach mentioned below –

1. Create a node structure which will contain three ints – **data** to store actual data, **next** to store index(address) of next node, **used** to maintain used and unused nodes. By default next will be -1 (like NULL) and used will be 0 (like free memory which can be allocated for a node). Deleting a node basically makes used variable to 0 means memory is free and again can be used.
2. We will create an array of nodes all of them will be unused (used=0).
3. Now we will do the header file operations same as linked list using the above mentioned concept.
4. Creating a new node is basically finding a node whose used = 0 and making used = 1 and deletion will make it 0 again. The pointer arithmetic in normal linked list will be similar to the array element index(address) arithmetic.



The algorithms to do the operations will be similar to singly linked list algorithms, only implementation process will be based on array.

**Structured Pseudocode:**

FOR THE ARRAYLINKEDLIST HEADER FILE:

typedef struct node

{

int data;

int next; // default -1 means null

int used; // default 0 when not in use or when deleted node

} ALLNode;

typedef struct array\_linked\_list

{

nodeAddress head;

ALLNode array[MAX\_LINKED\_LIST\_SIZE];

} ArrayLinkedList;

ArrayLinkedList \*prepare\_linked\_list()

{

ArrayLinkedList \*plist = (ArrayLinkedList \*)malloc(sizeof(ArrayLinkedList));

int i;

for (i = 0; i < MAX\_LINKED\_LIST\_SIZE; i++)

{

plist->array[i].data = 0;

plist->array[i].used = 0;

plist->array[i].next = -1;

}

plist->head = -1;

return plist;

}

nodeAddress createnode(int value, ArrayLinkedList \*plist)

{

nodeAddress temp = -1;

int i;

for (i = 0; i < MAX\_LINKED\_LIST\_SIZE; i++)

{

if (plist->array[i].used == 0)

{

temp = i;

plist->array[i].data = value;

plist->array[i].next = -1;

plist->array[i].used = 1;

break;

}

}

if (temp == -1)

{

printf("\nList is full");

}

return temp;

}

int empty\_l(ArrayLinkedList \*plist)

{

return (plist->head == -1);

}

int atend\_l(nodeAddress node, ArrayLinkedList \*plist)

{

return (plist->array[node].next == -1);

}

void insert\_front(ArrayLinkedList \*plist, nodeAddress target)

{

if (target == -1)

{

printf("\nList full!!!");

return;

}

else

{

plist->array[target].next = plist->head;

plist->head = target;

}

}

void insert\_after(nodeAddress prev, ArrayLinkedList \*plist, nodeAddress target)

{

if (target == -1)

{

printf("\nList full!!!");

return;

}

plist->array[target].next = plist->array[prev].next;

plist->array[prev].next = target;

}

void delete\_after(nodeAddress prev, ArrayLinkedList \*plist)

{

nodeAddress temp = plist->array[prev].next;

plist->array[prev].next = plist->array[temp].next

plist->array[temp].used = 0;

plist->array[temp].next = -1;

}

void delete\_front(ArrayLinkedList \*plist)

{

nodeAddress temp = plist->head;

plist->head = plist->array[plist->head].next;

plist->array[temp].used = 0;

plist->array[temp].next = -1;

}

void print\_l(ArrayLinkedList \*plist)

{

nodeAddress temp = plist->head;

printf("\n");

while (temp != -1)

{

printf("%d -> ", plist->array[temp].data);

temp = plist->array[temp].next;

}

}

void delete\_l(ArrayLinkedList \*plist)

{

int i;

for (i = 0; i < MAX\_LINKED\_LIST\_SIZE; i++)

{

plist->array[i].used = 0;

plist->array[i].next = -1;

}

plist->head = -1;

}

THE MAIN C FILE:

ArrayLinkedList \*plist;

void Build\_list()

{

printf("\nHow many elements: ");

int n;

scanf("%d", &n);

printf("\nEnter the elements of the list: ");

int val;

scanf("%d", &val);

nodeAddress head = createnode(val, plist);

insert\_front(plist, head);

nodeAddress ptr = head;

nodeAddress newNode;

while (n > 1)

{

scanf("%d", &val);

newNode = createnode(val, plist);

insert\_after(ptr, plist, newNode);

ptr = newNode;

n--;

}

}

void print\_list\_in\_reverse(nodeAddress head)

{ // this is the function for printing the list in reverse

if (head == -1)

{

return;

}

print\_list\_in\_reverse(plist->array[head].next);

printf("%d-->", plist->array[head].data);

}

int size\_of\_list()

{

nodeAddress ptr = plist->head;

if (ptr == -1)

{

return 0;

}

int count = 0;

while (ptr != -1)

{

count++;

ptr = plist->array[ptr].next;

}

return count;

}

void delete\_nth\_node(int n)

{

int i;

nodeAddress temp, ptr;

if (plist->head == -1)

{

printf("\nThe list is empty!!");

return;

}

ptr = plist->head;

if (n == 1)

{

delete\_front(plist);

return;

}

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

{

if ((temp = plist->array[ptr].next) == -1)

{

printf("\nInvalid n!!");

return;

}

ptr = temp;

}

delete\_after(ptr, plist);

}

void delete\_last\_node()

{

nodeAddress temp, temp2;

if (plist->head == -1)

{

printf("\nThe list is empty!!");

return;

}

if (atend\_l(plist->head, plist))

{

delete\_front(plist);

}

temp = plist->array[plist->head].next;

while (!atend\_l(temp, plist))

{

temp2 = temp;

temp = plist->array[temp].next;

}

delete\_after(temp2, plist);

}

int isList\_ordered()

{

int ascending = 0, descending = 0, ordered = 1;

nodeAddress ptr = plist->head, ptr\_later = plist->array[plist->head].next;

do

{

if (plist->array[ptr].data > plist->array[ptr\_later].data)

{

descending = 1;

}

else if (plist->array[ptr].data < plist->array[ptr\_later].data)

{

ascending = 1;

}

if (ascending \* descending)

{

ordered = 0;

break;

}

ptr = plist->array[ptr].next;

ptr\_later = plist->array[ptr\_later].next;

} while (!atend\_l(ptr, plist));

return ordered;

}

void Reverse\_list()

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

nodeAddress previous = -1;

nodeAddress current = plist->head;

nodeAddress later = plist->head;

while (current != -1)

{

later = plist->array[current].next;

plist->array[current].next = previous;

previous = current;

current = later;

}

plist->head = previous;

}

void sort\_list()

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

nodeAddress ptr1 = plist->head;

nodeAddress ptr2 = plist->array[plist->head].next;

while (ptr1 != -1)

{

ptr2 = plist->array[ptr1].next;

while (ptr2 != -1)

{

if (plist->array[ptr2].data < plist->array[ptr1].data)

{

temp = plist->array[ptr2].data;

plist->array[ptr2].data = plist->array[ptr1].data;

plist->array[ptr1].data = temp;

}

ptr2 = plist->array[ptr2].next;

}

ptr1 = plist->array[ptr1].next;

}

}

void swap\_element\_pairwise()

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

nodeAddress ptr1 = plist->head;

nodeAddress ptr2 = plist->array[plist->head].next;

while (ptr2 != -1)

{

temp = plist->array[ptr2].data;

plist->array[ptr2].data = plist->array[ptr1].data;

plist->array[ptr1].data = temp;

ptr1 = plist->array[plist->array[ptr1].next].next;

if (plist->array[ptr2].next == -1)

{

break;

}

ptr2 = plist->array[plist->array[ptr2].next].next;

}

}

void delete\_alternate\_node()

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

nodeAddress ptr1 = plist->head;

while (ptr1 != -1 && !atend\_l(ptr1, plist))

{

delete\_after(ptr1, plist);

ptr1 = plist->array[ptr1].next;

}

}

void insert\_before(int val, int val2)

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

nodeAddress ptr1 = plist->head;

nodeAddress ptr2 = plist->array[ptr1].next;

nodeAddress newNode = createnode(val, plist);

int data;

while (ptr2 != -1)

{

data = plist->array[plist->array[ptr1].next].data;

if (data == val2)

{

insert\_after(ptr1, plist, newNode);

break;

}

ptr1 = plist->array[ptr1].next;

ptr2 = plist->array[ptr1].next;

}

}

void insert\_last(int val)

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

nodeAddress ptr1 = plist->head;

nodeAddress newNode = createnode(val, plist);

while (!atend\_l(ptr1, plist))

{

ptr1 = plist->array[ptr1].next;

}

insert\_after(ptr1, plist, newNode);

}

void move\_last\_to\_first()

{

if ((plist->head) == -1)

{

printf("\nThe list is empty..\n");

return;

}

nodeAddress ptr1 = plist->head, ptr2;

while (!atend\_l(ptr1, plist))

{

ptr2 = ptr1;

ptr1 = plist->array[ptr1].next;

}

plist->array[ptr2].next = -1;

plist->array[ptr1].next = (plist->head);

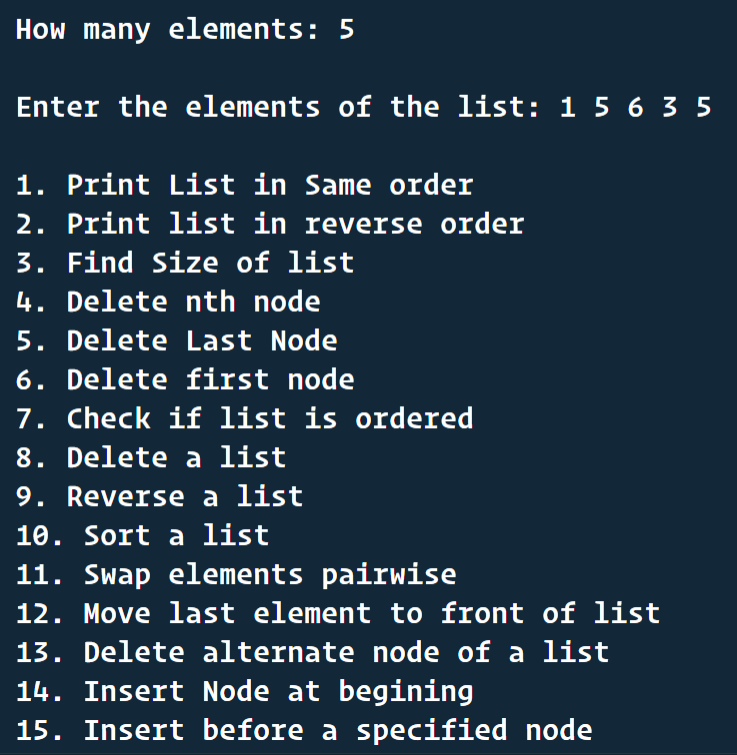
plist->head = ptr1;

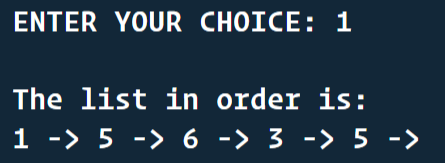
}

**Results:**

The results and outcomes are same as a normal singly linked list results, though the internal representation of data here is different. Nonetheless, as the linked list is based on array here, so, the two linked list operations like merging two lists and appending a list to the end etc. can not be normally implemented.

**HERE THE OUTPUT WILL BE COMPLETELY SAME AS THE SINGLY LINKED LIST OUTPUT, DIFFERENCE WILL BE IN THE INTERNAL IMPLEMENTATION. BUT OUTPUT WILL BE SAME AS THIS ARRAY LINKED LIST IS ALSO REPLICATING A SINGLY LINKED LIST. THAT’S WHY, ONLY TWO SCREEN SHOTS ARE GIVEN –**





**Discussion:**

Here we have used a static array for this linked list implementation that’s why if the linked list less number of nodes, memory wastage is there and also if the linked list wants to grow beyond the static array size, there no way to handle it.

**PROBLEM 5:**

**Problem statement:**

Repeat problems 1 and 3 for a circular single linked list, doubly linked list and circular doubly linked list. You need to develop Build\_list and Build\_list\_reverse, as well as Print\_list for each case.

**Solution Approach :**

So, here we have to implement the operations for doubly, circular singly and circular double linked lists. So, in the question no 1, we have written the pseudo code for a singly linked list. For the variations of linked lists mentioned in this question, the approach to apply the operations are similar. here we need to take care of these extra things –

#) In a doubly linked list, each node consists of two pointers, prev and next. During implementation, we need to change these pointers’ value.

#) In a circular singly linked list, the tail node is again pointed to head. So, here we need to remember that the end node of list means (node->next == head).

#) In a circular doubly linked list, we need to take care of the previous two points together.

**Structured Pseudocode:**

DOUBLY LINKED LIST HEADER FILE:

// defining the Node of a linked list using structure

typedef struct dllnode

{

int data;

struct dllnode \*prev;

struct dllnode \*next;

} DLLNode;

// create a new Node with integer data

DLLNode \* createDLLnode(int data) {

DLLNode \* newnode = (DLLNode \*)malloc(sizeof(DLLNode));

if (newnode == NULL) {

return NULL;

}

newnode->data = data;

newnode->prev = NULL;

newnode->next = NULL;

return newnode;

}

// initializing a new list

DLLNode \* init\_l()

{

return NULL;

}

// check if the list is empty

int empty\_l(DLLNode \*head)

{

return (head == NULL);

}

// check if the pointer is in the last of list

int atend\_l(DLLNode \*cur)

{

if (cur == NULL)

{

return 0;

}

else

{

return (cur->next == NULL);

}

}

// delete first element of the list

DLLNode \* delete\_front(DLLNode \*\*phead) {

if (\*phead == NULL){

return NULL;

}

DLLNode \* temp = \*phead;

\*phead = (\*phead)->next;

if(\*phead){

(\*phead)->prev = NULL;

}

temp->next = NULL;

return(temp);

}

// delete an element after a particular node

DLLNode \* delete\_after(DLLNode \*previous){

if(previous == NULL || previous->next == NULL){

return NULL;

}

previous = previous->next;

previous->prev->next = previous->next;

if(previous->next){

previous->next->prev = previous->prev;

}

previous->next = NULL;

previous->prev = NULL;

return previous;

}

// insert a new node at the begining of the list

void insert\_front(DLLNode \*target, DLLNode \*\*phead) {

if (target == NULL || \*phead == NULL){

return;

}

target->next = \*phead;

(\*phead)->prev = target;

\*phead = target;

}

// insert a new node after a specific node

void insert\_after(DLLNode \*target, DLLNode \*previous) {

if (target == NULL || previous == NULL){

printf("\nrrr");

return;

}

target->next = previous->next;

target->prev = previous;

previous->next = target;

if(target->next){

target->next->prev = target;

}

}

// print the complete list

void print\_l(DLLNode \* head) {

if (head == NULL) {

printf("\nList is empty");

return;

}

DLLNode \* cur = head;

while (cur != NULL)

{

printf("%d-->", cur->data);

cur = cur->next;

}

}

// delete the complete list

void delete\_l(DLLNode \*\* phead) {

if((\*phead) == NULL){

return;

}

DLLNode \* temp = NULL;

while(!empty\_l(\*phead)) {

temp = delete\_front(phead);

free(temp);

temp = NULL;

}

}

CIRCULAR SINGLY LINKED LIST HEADER FILE

// defining the CLLNode of a linked list using structure

typedef struct CLLNode

{

int data;

struct CLLNode \*next;

} CLLNode;

// create a new CLLNode with integer data

CLLNode \* createCLLNode(int data) {

CLLNode \* newCLLNode = (CLLNode \*)malloc(sizeof(CLLNode));

if (newCLLNode == NULL) {

return NULL;

}

newCLLNode->data = data;

newCLLNode->next = newCLLNode;

return newCLLNode;

}

// initializing a new list

CLLNode \* init\_l()

{

return NULL;

}

// check if the list is empty

int empty\_l(CLLNode \*head)

{

return (head == NULL);

}

// check if the pointer is in the last of list

int atend\_l(CLLNode \*cur, CLLNode \* head)

{

if (cur == NULL)

{

return 0;

}

else

{

return (cur->next == head);

}

}

// delete an element after a particular CLLNode

CLLNode \* delete\_after(CLLNode \*prev, CLLNode \*\* phead){

if(prev == NULL || prev->next == NULL){

return NULL;

}

CLLNode \* temp;

if(atend\_l(prev, \*phead)){

temp = \*phead;

if(temp->next == temp){

\*phead = NULL;

return temp;

}

(\*phead) = (\*phead)->next;

prev->next = \*phead;

temp->next = temp;

return temp;

}else{

temp = prev->next;

prev->next = temp->next;

temp->next = temp;

return temp;

}

// delete first element of the list

CLLNode \* delete\_front(CLLNode \*\*phead) {

if (\*phead == NULL){

return NULL;

}

CLLNode \* temp = \*phead;

while(!atend\_l(temp, \*phead)){

temp = temp->next;

}

return delete\_after(temp, phead);

}

// insert a new CLLNode at the begining of the list

void insert\_front(CLLNode \*target, CLLNode \*\*phead) {

if (target == NULL || \*phead == NULL){

return;

}

CLLNode \* temp = \*phead;

while(!atend\_l(temp, \*phead)){

temp = temp->next;

}

temp->next = target;

target->next = \*phead;

\*phead = target;

}

// insert a new CLLNode after a specific CLLNode

void insert\_after(CLLNode \*target, CLLNode \*prev) {

if (target == NULL || prev == NULL){

return;

}

target->next = prev->next;

prev->next = target;

}

// print the complete list

void print\_l(CLLNode \* head) {

if (head == NULL) {

printf("\nList is empty");

return;

}

CLLNode \* cur = head;

int i=0

while (cur != head || i==0)

{

printf("%d-->", cur->data);

cur = cur->next;

i++;

}

}

// delete the complete list

void delete\_l(CLLNode \*\* phead) {

if((\*phead) == NULL){

return;

}

CLLNode \* temp = NULL;

while(!empty\_l(\*phead)) {

temp = delete\_front(phead);

free(temp);

temp = NULL;

}

}

CIRCULAR DOUBLY LINKED LIST HEADER FILE:

// defining the CDLLNode of a linked list using structure

typedef struct CDLLNode

{

int data;

struct CDLLNode \*next;

struct CDLLNode \*prev;

} CDLLNode;

// create a new CDLLNode with integer data

CDLLNode \*createCDLLNode(int data)

{

CDLLNode \*newCDLLNode = (CDLLNode \*)malloc(sizeof(CDLLNode));

if (newCDLLNode == NULL)

{

return NULL;

}

newCDLLNode->data = data;

newCDLLNode->next = newCDLLNode;

newCDLLNode->prev = newCDLLNode;

return newCDLLNode;

}

// initializing a new list

CDLLNode \*init\_l()

{

return NULL;

}

// check if the list is empty

int empty\_l(CDLLNode \*head)

{

return (head == NULL);

}

// check if the pointer is in the last of list

int atend\_l(CDLLNode \*cur, CDLLNode \*head)

{

if (cur == NULL)

{

return 0;

}

else

{

return (cur->next == head);

}

}

// delete an element after a particular CDLLNode

CDLLNode \*delete\_after(CDLLNode \*prev, CDLLNode \*\*phead)

{

if (prev == NULL || prev->next == NULL)

{

return NULL;

}

CDLLNode \*temp;

if (atend\_l(prev, \*phead))

{

temp = \*phead;

if (temp->next == temp)

{

\*phead = NULL;

return temp;

}

(\*phead) = (\*phead)->next;

prev->next = \*phead;

(\*phead)->prev = prev;

temp->next = temp;

temp->prev = temp;

return temp;

}

else

{

temp = prev->next;

prev->next = temp->next;

temp->next->prev = prev;

temp->next = temp;

temp->prev = temp;

return temp;

}

}

// delete first element of the list

CDLLNode \*delete\_front(CDLLNode \*\*phead)

{

if (\*phead == NULL)

{

return NULL;

}

CDLLNode \*temp = \*phead;

while (!atend\_l(temp, \*phead))

{

temp = temp->next;

}

return delete\_after(temp, phead);

}

// insert a new CDLLNode at the begining of the list

void insert\_front(CDLLNode \*target, CDLLNode \*\*phead)

{

if (target == NULL)

{

return;

}

if (\*phead == NULL)

{

\*phead = target;

}

CDLLNode \*temp = \*phead;

while (!atend\_l(temp, \*phead))

{

temp = temp->next;

}

temp->next = target;

target->prev = temp;

target->next = \*phead;

(\*phead)->prev = target;

\*phead = target;

}

// insert a new CDLLNode after a specific CDLLNode

void insert\_after(CDLLNode \*target, CDLLNode \*prev)

{

if (target == NULL || prev == NULL)

{

return;

}

target->next = prev->next;

target->prev = prev;

prev->next = target;

target->next->prev = target;

}

// print the complete list

void print\_l(CDLLNode \*head)

{

if (head == NULL)

{

printf("\nList is empty");

return;

}

CDLLNode \*cur = head;

int i = 0;

while (cur != head || i == 0)

{

printf("%d-->", cur->data);

cur = cur->next;

i++;

}

}

// delete the complete list

void delete\_l(CDLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

return;

}

CDLLNode \*temp = NULL;

while (!empty\_l(\*phead))

{

temp = delete\_front(phead);

free(temp);

temp = NULL;

}

}

CIRCULAR SINGLY OTHER OPERATIONS:

CLLNode \*Build\_list()

{

printf("\nHow many elements: ");

int n;

scanf("%d", &n);

printf("\nEnter the elements of the list: ");

CLLNode \*head = init\_l();

int val;

scanf("%d", &val);

head = createCLLNode(val);

CLLNode \*ptr = head;

while (n > 1)

{

scanf("%d", &val);

CLLNode \*newCLLNode = createCLLNode(val);

insert\_after(newCLLNode, ptr);

ptr = ptr->next;

n--;

}

return head;

}

void print\_list\_in\_reverse(CLLNode \*head, CLLNode \*ptr)

{ // this is the function for printing the list in reverse

if (ptr->next == head)

{

printf("%d-->", ptr->data);

return;

}

print\_list\_in\_reverse(head, ptr->next);

printf("%d-->", ptr->data);

}

int size\_of\_list(CLLNode \*head)

{

CLLNode \*ptr = head;

if (ptr == NULL)

{

return 0;

}

int count = 0;

while (ptr->next != head)

{

count++;

ptr = ptr->next;

}

return count + 1;

}

CLLNode \*delete\_nth\_CLLNode(CLLNode \*\*phead, int n)

{

int i;

CLLNode \*temp, \*ptr;

if ((\*phead) == NULL)

{

printf("\nThe list is empty!!");

return NULL;

}

ptr = \*phead;

if (n == 1)

{

return delete\_front(phead);

}

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

{

if ((temp = ptr->next) == \*phead)

{

printf("\nInvalid n!!");

return NULL;

}

ptr = temp;

}

return delete\_after(ptr, phead);

}

CLLNode \*delete\_last\_CLLNode(CLLNode \*\*phead)

{

int size = size\_of\_list(\*phead);

return delete\_nth\_CLLNode(phead, size);

}

int isList\_ordered(CLLNode \*head)

{

int ascending = 0, descending = 0, ordered = 1;

CLLNode \*ptr = head, \*ptr\_later = head->next;

if (!ptr\_later)

{

return 1;

}

do

{

if (ptr->data > ptr\_later->data)

{

descending = 1;

}

else if (ptr->data < ptr\_later->data)

{

ascending = 1;

}

if (ascending \* descending)

{

ordered = 0;

break;

}

ptr = ptr->next;

ptr\_later = ptr\_later->next;

} while (!atend\_l(ptr, head));

return ordered;

}

void Reverse\_list(CLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

CLLNode \*previous = NULL;

CLLNode \*current = \*phead;

CLLNode \*later = \*phead;

do

{

later = current->next;

current->next = previous;

previous = current;

current = later;

} while (current != \*phead);

current->next = previous;

\*phead = previous;

}

void sort\_list(CLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

CLLNode \*ptr1 = \*phead;

CLLNode \*ptr2 = (\*phead)->next;

do

{

ptr2 = ptr1->next;

do

{

if (ptr2->data < ptr1->data)

{

temp = ptr2->data;

ptr2->data = ptr1->data;

ptr1->data = temp;

}

ptr2 = ptr2->next;

} while (ptr2 != \*phead);

ptr1 = ptr1->next;

} while (!atend\_l(ptr1, \*phead));

}

void swap\_element\_pairwise(CLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

CLLNode \*ptr1 = \*phead;

CLLNode \*ptr2 = (\*phead)->next;

do

{

temp = ptr2->data;

ptr2->data = ptr1->data;

ptr1->data = temp;

ptr1 = ptr1->next->next;

ptr2 = ptr2->next->next;

} while (ptr1 != \*phead && ptr2 != \*phead);

}

void delete\_alternate\_CLLNode(CLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

CLLNode \*ptr1 = \*phead;

do

{

delete\_after(ptr1, phead);

ptr1 = ptr1->next;

} while (ptr1 != \*phead && !atend\_l(ptr1, \*phead));

}

void insert\_before(CLLNode \*\*phead, int val, int val2)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

CLLNode \*ptr1 = \*phead;

CLLNode \*ptr2 = ptr1->next;

CLLNode \*newCLLNode = createCLLNode(val);

do

{

if (ptr1->next->data == val2)

{

insert\_after(newCLLNode, ptr1);

break;

}

ptr1 = ptr1->next;

ptr2 = ptr1->next;

} while (ptr2 != \*phead);

}

void insert\_last(CLLNode \*\*phead, int val)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

CLLNode \*ptr1 = \*phead;

CLLNode \*newCLLNode = createCLLNode(val);

while (!atend\_l(ptr1, \*phead))

{

ptr1 = ptr1->next;

}

insert\_after(newCLLNode, ptr1);

}

void move\_last\_to\_first(CLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

CLLNode \*ptr1 = \*phead;

while (!atend\_l(ptr1, \*phead))

{

ptr1 = ptr1->next;

}

\*phead = ptr1;

}

void check\_equal\_lists(CLLNode \*h1, CLLNode \*h2)

{

if (h1 == NULL || h2 == NULL)

{

printf("\nThe lists are not equal..");

return;

}

CLLNode \*ptr1 = h1, \*ptr2 = h2;

do

{

if (ptr1->data != ptr2->data)

{

printf("\nNot equal lists...");

return;

}

ptr1 = ptr1->next;

ptr2 = ptr2->next;

} while (ptr1 != h1 && ptr2 != h2);

if (ptr1 == h1 && ptr2 == h2)

{

printf("\nEqual lists...");

}

else

{

printf("\nNot equal lists...");

}

}

void append\_second\_after\_first(CLLNode \*h1, CLLNode \*h2)

{

if (h1 == NULL)

{

printf("\nThe first list is empty...");

return;

}

CLLNode \*ptr = h1;

while (!atend\_l(ptr, h1))

{

ptr = ptr->next;

ptr->next = h2;

ptr = h2;

while (!atend\_l(ptr, h2))

{

ptr = ptr->next;

}

ptr->next = h1;

}

DOUBLY LINKED LIST OTHER OPERATIONS:

DLLNode \*Build\_list()

{

printf("\nHow many elements: ");

int n;

scanf("%d", &n);

printf("\nEnter elements of the list: ");

DLLNode \*head = init\_l();

int val;

scanf("%d", &val);

head = createDLLnode(val);

DLLNode \*ptr = head;

while (n > 1)

{

scanf("%d", &val);

DLLNode \*newDLLNode = createDLLnode(val);

insert\_after(newDLLNode, ptr);

ptr = ptr->next;

n--;

}

return head;

}

void print\_list\_in\_reverse(DLLNode \*head)

{ // this is the function for printing the list in reverse

if (head == NULL)

{

return;

}

print\_list\_in\_reverse(head->next);

printf("%d-->", head->data);

}

int size\_of\_list(DLLNode \*head)

{

DLLNode \*ptr = head;

if (ptr == NULL)

{

return 0;

}

int count = 0;

while (ptr != NULL)

{

count++;

ptr = ptr->next;

}

return count;

}

DLLNode \*delete\_nth\_DLLnode(DLLNode \*\*phead, int n)

{

int i;

DLLNode \*temp, \*ptr;

if ((\*phead) == NULL)

{

printf("\nThe list is empty!!");

return NULL;

}

ptr = \*phead;

if (n == 1)

{

return delete\_front(phead);

}

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

{

if ((temp = ptr->next) == NULL)

{

printf("\nInvalid n!!");

return NULL;

}

ptr = temp;

}

return delete\_after(ptr);

}

DLLNode \*delete\_last\_DLLnode(DLLNode \*head)

{

DLLNode \*temp;

if (head == NULL)

{

printf("\nThe list is empty!!");

return NULL;

}

if (atend\_l(head))

{

return head;

}

temp = head;

while (!atend\_l(temp))

{

head = temp;

temp = temp->next;

}

return delete\_after(head);

}

int isList\_ordered(DLLNode \*head)

{

int ascending = 0, descending = 0, ordered = 1;

DLLNode \*ptr = head, \*ptr\_later = head->next;

do

{

if (ptr->data > ptr\_later->data)

{

descending = 1;

}

else if (ptr->data < ptr\_later->data)

{

ascending = 1;

}

if (ascending \* descending)

{

ordered = 0;

break;

}

ptr = ptr->next;

ptr\_later = ptr\_later->next;

} while (!atend\_l(ptr));

return ordered;

}

void Reverse\_list(DLLNode \*\*phead)

{

DLLNode \*current, \*temp;

current = (\*phead);

while (!atend\_l(current))

{

temp = current->next;

current->next = current->prev;

current->prev = temp;

current = temp;

}

current->next = current->prev;

current->prev = NULL;

(\*phead) = temp;

}

void sort\_list(DLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

DLLNode \*ptr1 = \*phead;

DLLNode \*ptr2 = (\*phead)->next;

while (ptr1 != NULL)

{

ptr2 = ptr1->next;

while (ptr2 != NULL)

{

if (ptr2->data < ptr1->data)

{

temp = ptr2->data;

ptr2->data = ptr1->data;

ptr1->data = temp;

}

ptr2 = ptr2->next;

}

ptr1 = ptr1->next;

}

}

void swap\_element\_pairwise(DLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

DLLNode \*ptr1 = \*phead;

DLLNode \*ptr2 = (\*phead)->next;

while (ptr2 != NULL)

{

temp = ptr2->data;

ptr2->data = ptr1->data;

ptr1->data = temp;

ptr1 = ptr1->next->next;

if (ptr2->next == NULL)

{

break;

}

ptr2 = ptr2->next->next;

}

}

void delete\_alternate\_DLLnode(DLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

int temp;

DLLNode \*ptr1 = \*phead;

while (ptr1 != NULL && !atend\_l(ptr1))

{

delete\_after(ptr1);

ptr1 = ptr1->next;

}

}

void insert\_before(DLLNode \*\*phead, int val, int val2)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

DLLNode \*ptr1 = \*phead;

DLLNode \*ptr2 = ptr1->next;

DLLNode \*newDLLNode = createDLLnode(val);

while (ptr2 != NULL)

{

if (ptr1->next->data == val2)

{

insert\_after(newDLLNode, ptr1);

break;

}

ptr1 = ptr1->next;

ptr2 = ptr1->next;

}

}

void insert\_last(DLLNode \*\*phead, int val)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

DLLNode \*ptr1 = \*phead;

DLLNode \*newDLLNode = createDLLnode(val);

while (!atend\_l(ptr1))

{

ptr1 = ptr1->next;

}

insert\_after(newDLLNode, ptr1);

}

void move\_last\_to\_first(DLLNode \*\*phead)

{

if ((\*phead) == NULL)

{

printf("\nThe list is empty..\n");

return;

}

DLLNode \*ptr1 = \*phead;

while (!atend\_l(ptr1))

{

ptr1 = ptr1->next;

ptr1->prev->next = NULL;

ptr1->prev = NULL;

ptr1->next = (\*phead);

(\*phead)->prev = ptr1;

\*phead = ptr1;

}

void check\_equal\_lists(DLLNode \*h1, DLLNode \*h2)

{

if (h1 == NULL || h2 == NULL)

{

printf("\nThe lists are not equal..");

return;

}

DLLNode \*ptr1 = h1, \*ptr2 = h2;

while (ptr1 != NULL && ptr2 != NULL)

{

if (ptr1->data != ptr2->data)

{

printf("\nNot equal lists...");

return;

}

ptr1 = ptr1->next;

ptr2 = ptr2->next;

}

if (ptr1 == NULL && ptr2 == NULL)

{

printf("\nEqual lists...");

}

else

{

printf("\nNot equal lists...");

}

}

void append\_second\_after\_first(DLLNode \*h1, DLLNode \*h2)

{

if (h1 == NULL)

{

printf("\nThe first list is empty...");

return;

}

DLLNode \*ptr = h1;

while (!atend\_l(ptr))

{

ptr = ptr->next;

}

ptr->next = h2;

}

DLLNode \*mergeLinkedList(DLLNode \*head1, DLLNode \*head2)

{

DLLNode \*resultHead, \*resultTail, \*temp;

resultHead = resultTail = NULL;

while (1)

{

/\* \*/

if (head1 == NULL)

{

resultTail->next = head2;

break;

}

if (head2 == NULL)

{

resultTail->next = head1;

break;

}

/\* Check whether current DLLnode of

which Linked list is smaller\*/

if (head1->data <= head2->data)

{

temp = head1;

head1 = head1->next;

}

else

{

temp = head2;

head2 = head2->next;

}

/\*Add smaller DLLnode to result linked list \*/

if (resultHead == NULL)

{

resultHead = resultTail = temp;

}

else

{

resultTail->next = temp;

resultTail = temp;

}

resultTail->next = NULL;

}

return resultHead;

}

**Results:**

*Here we are implementing same operations for doubly, circular doubly and circular singly linked lists. For each case internal implementation will be different but after running the program the terminal output is totally same as all the operations are same. Outputs are given for a singly linked list in question 03 and 02. Same output will come here also.*

**Discussion:**

For this problem, some small points must be remembered. Circular linked list’s last element means whose, next is pointing to head. For doubly linked list one node can delete himself which is not possible for singly linked list. Reversing doubly linked list takes less temporary pointer variables compared to singly linked list. After writing functions for same operations for each type of linked list, practice and debugging power increases.

**PROBLEM 6:**

**Problem statement:**

Implement an application to find out the Inverted Index of a set of text files, given a set of keywords. Create a set of 6 text files having the keywords in different positions in the text files. The keywords may occur multiple times in a file. The inverted index file will list the key words along with the filenames in which they occur and how many times they occur in that file.

**Solution Approach:**

We will create six text files with keywords in it. And in the program, we will pass our desired keywords to find and count from these text files. Then for each text file we will open it read mode and scan it word by word, and for each word we will compare with the keywords. We will maintain the counts of the keywords for each file and print them in the output inverted\_index.txt file. For comparison we will use string library function strcmp().

**C code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define number\_of\_keywords 6

int keywords\_count[number\_of\_keywords];

FILE \*renew;

void make\_inverted\_index(char \*filename, char \*\*keywords)

{

FILE \*fp = fopen(filename, "r");

FILE \*fp2 = fopen("inv\_index.txt", "a");

int i;

if (!fp || !fp2)

printf("\nFile doesn't exist");

char str[20];

for (i = 0; i < number\_of\_keywords; i++)

{

keywords\_count[i] = 0;

}

while (!feof(fp))

{

fscanf(fp, "%s", str);

for (i = 0; i < number\_of\_keywords; i++)

{

if (strcmp(str, keywords[i]) == 0)

{

keywords\_count[i]++;

}

}

fprintf(fp2, "\n\n\nFOR FILE %s ::---", filename);

for (i = 0; i < number\_of\_keywords; i++)

{

fprintf(fp2, "\n%s - %d", keywords[i], keywords\_count[i]);

fclose(fp);

fclose(fp2);

}

int main()

{

char \*\*keywords = (char \*\*)malloc(number\_of\_keywords \* sizeof(char \*));

int i;

for (i = 0; i < number\_of\_keywords; i++)

{

keywords[i] = (char \*)malloc(10 \* sizeof(char));

}

for (i = 0; i < number\_of\_keywords; i++)

{

printf("\nEnter the %dth keyword: ", i + 1);

fflush(stdin);

gets(keywords[i]);

}

renew = fopen("inv\_index.txt", "w");

fclose(renew);

make\_inverted\_index("f1.txt", keywords);

make\_inverted\_index("f2.txt", keywords);

make\_inverted\_index("f3.txt", keywords);

make\_inverted\_index("f4.txt", keywords);

make\_inverted\_index("f5.txt", keywords);

make\_inverted\_index("f6.txt", keywords);

for (i = 0; i < number\_of\_keywords; i++)

{

free(keywords[i]);

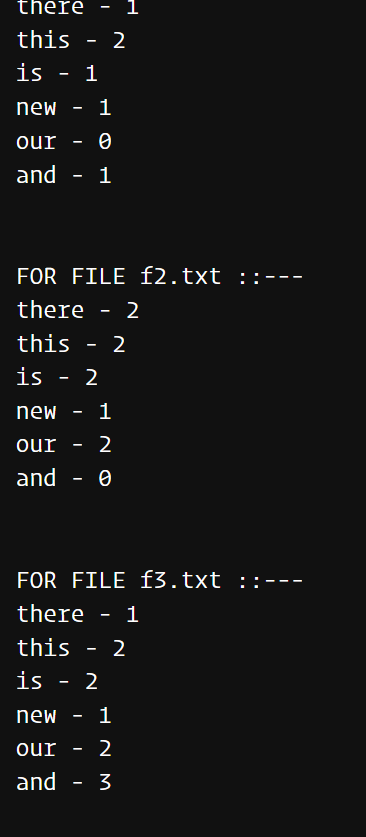
}

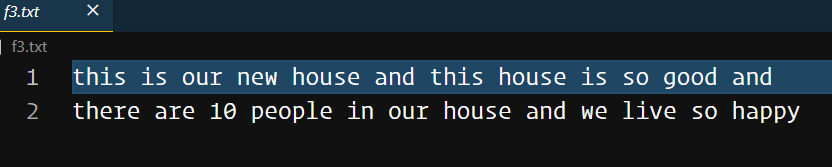
free(keywords);

}

**Results:**

For this problem we have made six text files with keywords in them and also, we specify the keywords we want to search for in the running program and then program creates another text file which will contain frequencies for each keyword in each file.





**Discussion:**

This program helps to understand file handling, string operations and functional programming as we are passing the file names as argument in a function. One thing is taken care of – in each file opening checking the file pointer helps us to understand any file opening failed or not.

**PROBLEM 7:**

**Problem statement:**

Write an application for adding, subtracting and multiplying very large numbers, say more than 70-digit integers, using (a) arrays and (b) linked lists to represent the large integers.

**Solution Approach:**

For this problem we have implemented large numbers using array and we can add two large integers.

We have actually created a structure type with an array of shorts which is going to store digits of a large integer. Initializing a number actually makes all digits zero. While taking a large integer as input we take it in a string first. Then we catch each character digit and convert into short and store in the array in a correct order. During add operation, we start to add from right side of two arrays also considering the carry for next step and store result in another Number type.

**Pseudocode:**

typedef struct number

{

short arr[MAX\_SIZE];

} Number;

Number init\_number()

{

Number n;

int i;

for (i = 0; i < MAX\_SIZE; i++)

{

n.arr[i] = 0;

}

return n;

}

Number take\_number()

{

Number n = init\_number();

printf("Enter Number: ");

char str[MAX\_SIZE];

fflush(stdin);

gets(str);

int len = strlen(str);

int i;

for (i = 0; i < len; i++)

{

n.arr[MAX\_SIZE - len + i] = (short)(str[i]) - 48;

}

return n;

}

Number add\_numbers(Number n1, Number n2)

{

int i, rem = 0;

Number out = init\_number();

for (i = MAX\_SIZE - 1; i >= 0; i--)

{

out.arr[i] = n1.arr[i] + n2.arr[i] + rem;

rem = out.arr[i] / 10;

out.arr[i] %= 10;

}

return out;

}

void print\_number(Number n)

{

int i;

for (i = 0; i < MAX\_SIZE; i++)

{

if (n.arr[i] != 0)

{

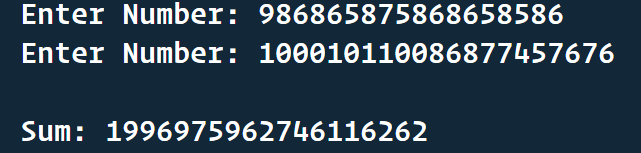
printf("%d", n.arr[i]);

}

}

}

**Results:**

****

**Discussion:**

This program helps us to do arithmetic of large integers which can not be stored even in long long data type.

**PROBLEM 8:**

**Problem statement:**

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

**Solution Approach:**

We are taking input for coordinates of the polygon clockwisely. For every point in the second polygon, we sum the areas of all triangles formed by edges in the first polygonal and this point. If this is equal to the area of the first polygon, then the point lies inside or on this polygon. Proceeding in this manner, this can be found for all points in the second polygon. This is again repeated for the first polygon. However for intersection, one polygon must have a vertex in the other polygon.

**Pseudo code:**

struct point {

float x

float y

point\* next

};

Standard ADT operations for linked list have been used

1. distance (point\* p1, point\* p2)

begin

return sqrt ((p1->x-p2->x)^2 + (p1->y-p2->y)^2)

end

2. get\_area (point\* p1, point\* p2, point\* p3) -> area of a triangle

begin

s1 = distance (p1, p2)

s2 = distance (p2, p3)

s3 = distance (p3, p1)

semi = (s1+s2+s3)/2

return sqrt (semi\*(semi-s1)\*(semi-s2)\*(semi-s3))

end

3. get\_poly\_area (point\* p, point\* poly, int\* flag) -> sum of area of triangle

formed by polygon poly and point p

begin

prev = poly

curr = poly->next

ret = 0

while !atend\_l (prev)

val = get\_area (p, prev, curr)

if val == 0

\*flag = 1

endif

ret += val

end while

val = get\_area (p, prev, poly)

if val==0

\*flag = 1

endif

ret += val

end

main ()

begin

point\* poly1 = First polygon of size n

point\* poly2 = Second polygon of size n

point\* dummy\_node1 = centroid of first polygon

point\* dummy\_node2 = centroid of second polygon

flag1 =0, flag2 = 0

temp1 = poly1, temp2 = poly2

area1 = get\_poly\_area (dummy\_node1, poly1, &flag1)

area2 = get\_poly\_area (dummy\_node2, poly2, &flag2)

count1 = 0, count2 = 0

for i=0 to n-1

val = get\_poly\_area (temp2, poly1, &flag1)

if val!=area1

count1++

endif

temp2 = temp2->next

endfor

for i=0, i<n, i++

val = get\_poly\_area (temp1, poly2, &flag2)

if val!=area2

count2++

endif

temp1 = temp1->next

endfor

if count1 == n and count2 == n

return not intersect //All points outside each other

else if count1==0 and count2==n and flag1==0

return touching //Second inside first

else if count2==0 and count1==n and flag2 ==0

return touching //First inside second

else if (count1==0 and count2==n) or (count2==n and count1==0)

return not intersecting //One contained in another

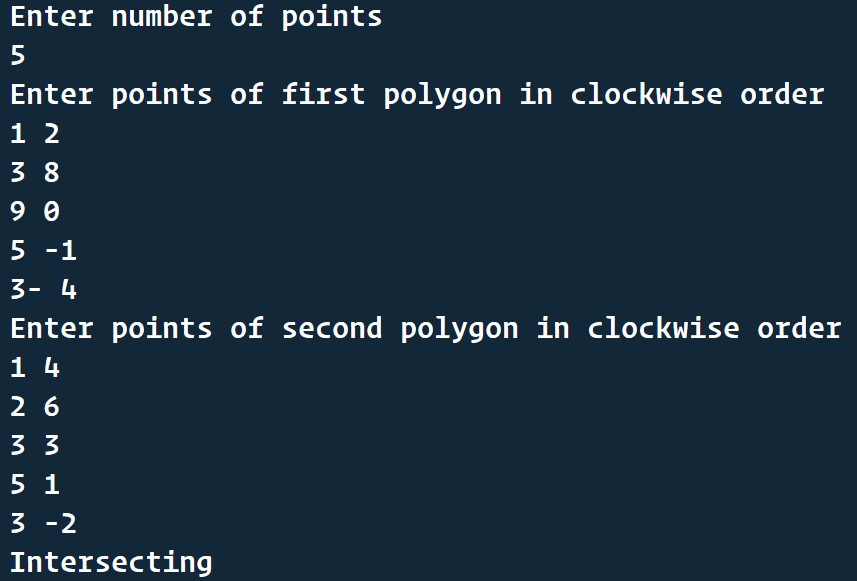
else

return Intersecting

endif

end

**Results:**

****

**Discussion:**

This program helps us to understand the approach to solve geometrical problems.