FILES :

1 . COUNT NEWLINE , COUNT CHARACTERS , COUNT TAB , COUNT SPACE

```
#include <stdio.h>
#include <stdlib.h>
// To count character , line , white , tab
int main() {
   FILE *fp ;
   char ch ;

   int charcount = 0 ;
   int linecount = 0 ;
   int whitecount = 0 ;
   int tabcount = 0 ;

   fp = fopen("C:\\Users\\nikhil2000\\OneDrive\\Desktop\\FILES2\\file.txt" , "r") ;
   if(fp == NULL)
   {
      printf("file not found") ;
      exit(0) ;
   }

   while((ch=getc(fp)) != EOF)
   {
      charcount++ ;
      if(ch==' ')
      {
         whitecount++ ;
      }
      if(ch == '\n')
      {
         linecount++ ;
      }
      if(ch == '\t')
      {
         tabcount++  ;
      }
   }
   fclose(fp) ;

   printf("\nCharacter Count = %d " , charcount) ;
   printf("\nWhiteSpace Count = %d " , whitecount) ;
   printf("\nline Count = %d " , linecount) ;
   printf("\ntab Count = %d " , tabcount) ;
   return 0;
}
```

2. COPY CONTENT OF ONE FILE TO ANOTHER

```
#include <stdio.h>
#include <stdlib.h>

// copy content of one file to another file
int main() {
    FILE *fp1 , *fp2 ;
    char ch ;
    fp1 = fopen("C:\\Users\\nikhil2000\\OneDrive\\Desktop\\FILES\\file1.txt" , "r") ;
    if(fp1 == NULL)
    {
        printf("File not found") ;
        exit(0) ;
    }
    fp2 = fopen("C:\\Users\\nikhil2000\\OneDrive\\Desktop\\FILES\\file2.txt" , "w") ;
    while((ch = fgetc(fp1)) != EOF)
    {
        fputc( ch , fp2 ) ;
    }
    fclose(fp1) ;
    fclose(fp2) ;
    fp2 = fopen("C:\\Users\\nikhil2000\\OneDrive\\Desktop\\FILES\\file2.txt" , "r") ;
    while((ch=fgetc(fp2)) != EOF)
    {
        printf("%c" , ch) ;
    }
    fclose(fp2) ;
}
```

3. CHECK COMMON USN AND UNIQUE USN

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int checkUSN(char temp[11], char USN[50][11], int n){
  for(int i = 0; i<n; i++){
    if(strcmp(temp, USN[i])==0){
      return 0;
    }
    return 1;
  }
}

void createFiles(){
  int n1, n2, i;
  FILE *f1, *f2;
  f1 = fopen("file1.txt", "w");
  f2 = fopen("file2.txt", "w");
  char USN[11];
  printf("Enter number of USNs in file1 :");
  scanf("%d", &n1);
    for(i = 0; i<n1; i++){
      scanf("%s", USN);
      fputs(USN, f1);
      fputs("\n", f1);
  }
  fclose(f1);
  printf("Enter number of USNs in file2 :");
  scanf("%d", &n2);                                  // writing into files
    for(i = 0; i<n2; i++){
      scanf("%s", USN);
      fputs(USN, f2);
      fputs("\n", f2);
  }

  fclose(f2);
  fflush(stdin);
}

void process_files() {
  FILE *f1, *f2, *f3, *f4;
  char commonUSN[50][11];
  int n1 = 0, k = 0;
  f1 = fopen("file1.txt", "r");
  f2 = fopen("file2.txt", "r");
  f3 = fopen("common_usns.txt", "w");
  f4 = fopen("unique_usns.txt", "w");
  char USN[50][11], temp[11];
  while(fgets(USN[n1], 11, f1) != NULL){
     n1++;

  fclose(f1);
  while(fgets(temp, 11, f2) != NULL){
    if(checkUSN(temp, USN, n1)==1){
      fputs(temp, f4);
      //strcpy(commonUSN[k++], temp);
    }
    if(checkUSN(temp, USN, n1)==0){
      //strcpy(USN[n1++], temp);
      fputs(temp, f3);
    }
}
}
    fclose(f2);
      fclose(f3);
        fclose(f4);

}

void sortUSNs(char USN[50][11], int n){
  char temp[11];
  for(int i = 0; i<n; i++){
    for(int j = 0; j<n-i-1; j++){
      if(strcmp(USN[j], USN[j+1])>0){
        strcpy(temp, USN[j]);
        strcpy(USN[j], USN[i]);
        strcpy(USN[i], temp);
      }
    }
  }
}

int main(){
  createFiles();
  process_files();
  return 0;
}
```

ORDERED LIST :

```
/* Program to insert in a sorted list */
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    int data;
    struct Node *next;
};
//************SORTED INSERT************

void sortedInsert(struct Node **head_ref, struct Node *new_node) {
    struct Node *current;

    if (*head_ref == NULL || (*head_ref)->data>= new_node->data) {
        new_node->next = *head_ref;
        *head_ref = new_node;
    } else {
        current = *head_ref;
        while (current->next != NULL && current->next->data < new_node->data) {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
//************ CREATE NODE************
struct Node *newNode(int new_data) {
    /* allocate node */
    struct Node *new_node
            = (struct Node *) malloc(
                    sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;
    new_node->next = NULL;

    return new_node;
}
//************PRINT LIST************
void printList(struct Node *head) {
    struct Node *temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
//************ DRIVER CODE************
int main() {
    /* Start with the empty list */
    struct Node *head = NULL;
    struct Node *new_node = newNode(5);
    sortedInsert(&head, new_node);
    new_node = newNode(10);
    sortedInsert(&head, new_node);
    new_node = newNode(7);
    sortedInsert(&head, new_node);
    new_node = newNode(3);
    sortedInsert(&head, new_node);
    new_node = newNode(1);
    sortedInsert(&head, new_node);
    new_node = newNode(9);
    sortedInsert(&head, new_node);
    printf("\n Created Linked List\n");
    printList(head);

    return 0;
}
```

RANDOM LIST OR SINGLY LINKED LIST :

```
#include <stdio.h>
#include <stdlib.h>
#define ISEMPTY printf("\nEMPTY LIST:");
struct node
{
   int value;
   struct node *next;
};

struct node* create_node(int);
void insert_node_first();
void insert_node_last();
void insert_node_pos();
void sorted_ascend();
void delete_pos();
void search();
void update_val();
void display();
void rev_display(struct node *);

typedef struct node snode;
snode *newnode, *ptr, *prev, *temp;
snode *first = NULL, *last = NULL;

int main()
{
   int ch;
   char ans = 'Y';

   while (1)
   {
      printf("\n---------------------------------\n");
      printf("\nOperations on singly linked list\n");
      printf("\n---------------------------------\n");
      printf("\n1.Insert node at first");
      printf("\n2.Insert node at last");
      printf("\n3.Insert node at position");
      printf("\n4.Sorted Linked List in Ascending Order");
      printf("\n5.Delete Node from any Position");
      printf("\n6.Update Node Value");
      printf("\n7.Search Element in the linked list");
      printf("\n8.Display List from Beginning to end");
      printf("\n9.Display List from end using Recursion");
      printf("\n10.Exit\n");
      printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
      printf("\nEnter your choice");
      scanf("%d", &ch);

      switch (ch)
      {
         case 1:
            printf("\n...Inserting node at first...\n");
            insert_node_first();
```

```
                break;
            case 2:
                printf("\n...Inserting node at last...\n");
                insert_node_last();
                break;
            case 3:
                printf("\n...Inserting node at position...\n");
                insert_node_pos();
                break;
            case 4:
                printf("\n...Sorted Linked List in Ascending Order...\n");
                sorted_ascend();
                break;
            case 5:
                printf("\n...Deleting Node from any Position...\n");
                delete_pos();
                break;
            case 6:
                printf("\n...Updating Node Value...\n");
                update_val();
                break;
            case 7:
                printf("\n...Searching Element in the List...\n");
                search();
                break;
            case 8:
                printf("\n...Displaying List From Beginning to End...\n");
                display();
                break;
            case 9:
                printf("\n...Displaying List From End using Recursion...\n");
                rev_display(first);
                break;
            case 10:
                printf("\n...Exiting...\n");
                return 0;
                break;
            default:
                printf("\n...Invalid Choice...\n");
                break;
        }

    }
    return 0;
}

snode* create_node(int val)
{
    newnode = (snode *)malloc(sizeof(snode));
    if (newnode == NULL)
    {
        printf("\nMemory was not allocated");
        return 0;
    }
    else
    {
        newnode->value = val;
        newnode->next = NULL;
        return newnode;
    }
}
//************ INSERT NODE FIRST ************
void insert_node_first()
{
    int val;

    printf("\nEnter the value for the node:");
    scanf("%d", &val);
    newnode = create_node(val);
    if (first == last && first == NULL)
    {
        first = last = newnode;
        first->next = NULL;
        last->next = NULL;
    }
    else
    {
        temp = first;
        first = newnode;
        first->next = temp;
    }
    printf("\n----INSERTED----");
}
//************ INSERT NODE LAST ************

void insert_node_last()
{
    int val;

    printf("\nEnter the value for the Node:");
    scanf("%d", &val);
    newnode = create_node(val);
    if (first == last && last == NULL)
    {
        first = last = newnode;
        first->next = NULL;
        last->next = NULL;
    }
    else
    {
        last->next = newnode;
        last = newnode;
        last->next = NULL;
    }
    printf("\n----INSERTED----");
}
```

//************ INSERT NODE AT POS ************

```
void insert_node_pos()
{
   int pos, val, cnt = 0, i;

   printf("\nEnter the value for the Node:");
   scanf("%d", &val);
   newnode = create_node(val);
   printf("\nEnter the position ");
   scanf("%d", &pos);
   ptr = first;
   while (ptr != NULL)
   {
      ptr = ptr->next;
      cnt++;
   }
   if (pos == 1)
   {
      if (first == last && first == NULL)
      {
         first = last = newnode;
         first->next = NULL;
         last->next = NULL;
      }
      else
      {
         temp = first;
         first = newnode;
         first->next = temp;
      }
      printf("\nInserted");
   }
   else if (pos>1 && pos<=cnt)
   {
      ptr = first;
      for (i = 1;i < pos;i++)
      {
         prev = ptr;
         ptr = ptr->next;
      }
      prev->next = newnode;
      newnode->next = ptr;
      printf("\n----INSERTED----");
   }
   else
   {
      printf("Position is out of range");
   }
}
```

//************ SORTED ASCEND************

```
void sorted_ascend()
{
    snode *nxt;
    int t;

    if (first == NULL)
    {
        ISEMPTY;
        printf(":No elements to sort\n");
    }
    else
    {
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {
            for (nxt = ptr->next;nxt != NULL;nxt = nxt->next)
            {
                if (ptr->value > nxt->value)
                {
                    t = ptr->value;
                    ptr->value = nxt->value;
                    nxt->value = t;
                }
            }
        }
        printf("\n---Sorted List---");
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {
            printf("%d\t", ptr->value);
        }
    }
}
//************ DELETE AT POSITION************

void delete_pos()
{
    int pos, cnt = 0, i;

    if (first == NULL)
    {
        ISEMPTY;
        printf(":No node to delete\n");
    }
    else
    {
        printf("\nEnter the position of value to be deleted:");
        scanf(" %d", &pos);
        ptr = first;
        if (pos == 1)
        {
            first = ptr->next;
            printf("\nElement deleted");
        }
        else
        {
            while (ptr != NULL)
            {
                ptr = ptr->next;
                cnt = cnt + 1;
            }
            if (pos > 0 && pos <= cnt)
            {
                ptr = first;
                for (i = 1;i < pos;i++)
                {
                    prev = ptr;
                    ptr = ptr->next;
                }
                prev->next = ptr->next;
            }
            else
            {
                printf("Position is out of range");
            }
            free(ptr);
            printf("\nElement deleted");
        }
    }
}
//************ UPDATE VALUE************
void update_val()
{
    int oldval, newval, flag = 0;

    if (first == NULL)
    {
        ISEMPTY;
        printf(":No nodes in the list to update\n");
    }
    else
    {
        printf("\nEnter the value to be updated:");
        scanf("%d", &oldval);
        printf("\nEnter the newvalue:");
        scanf("%d", &newval);
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {
            if (ptr->value == oldval)
            {
                ptr->value = newval;
                flag = 1;
                break;
            }
        }
        if (flag == 1)
        {
            printf("\nUpdated Successfully");
        }
        else
        {
            printf("\nValue not found in List");
```

```
      }
   }
}
//************ SEARCH KEY ************

void search()
{
   int flag = 0, key, pos = 0;

   if (first == NULL)
   {
      ISEMPTY;
      printf(":No nodes in the list\n");
   }
   else
   {
      printf("\nEnter the value to search");
      scanf("%d", &key);
      for (ptr = first;ptr != NULL;ptr = ptr->next)
      {
         pos = pos + 1;
         if (ptr->value == key)
         {
            flag = 1;
            break;
         }
      }
      if (flag == 1)
      {
         printf("\nElement %d found at %d position\n", key, pos);
      }
      else
      {
         printf("\nElement %d not found in list\n", key);
      }
   }
}
//************ DISPLAY ************

void display()
{
   if (first == NULL)
   {
      ISEMPTY;
      printf(":No nodes in the list to display\n");
   }
   else
   {
      for (ptr = first;ptr != NULL;ptr = ptr->next)
      {
         printf("%d\t", ptr->value);
      }
   }
}
```

```
void rev_display(snode *ptr)
{
   int val;

   if (ptr == NULL)
   {
      ISEMPTY;
      printf(":No nodes to display\n");
   }
   else
   {
      if (ptr != NULL)
      {
         val = ptr->value;
         rev_display(ptr->next);
         printf("%d\t", val);
      }

   }
}
```

CIRCULAR LINKED LIST :

```
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser or add your own getch, system("pause") or input loop
*/
typedef struct node
{
   int data;
   struct node *link;
}NODE;
typedef struct list
{
   NODE *head;
   NODE *rear;
   int count;
}LIST;
```

//********** GET NODE**************

```
NODE * getnode(int element)
{
          NODE * newnode;
          newnode=(NODE *)malloc(sizeof(NODE));
          newnode->data=element;
          newnode->link=NULL;
          return newnode;
}
```

```
//************ INSERT NODE AT FRONT ************

void insertfront(LIST *lp,int ele)
{
        NODE * newnode;
        //create new node to be inserted
        newnode=getnode(ele);
        //check if list is empty
        if(lp->head==NULL)
        {
                lp->head=lp->rear=newnode;
                newnode->link=lp->head;
                lp->count++;
                return;
        }
        //to insert node in nonempty list insert@front
        newnode->link=lp->head;//point to previous head node
        lp->head=newnode;
        (lp->rear)->link=newnode;
        lp->count++;
}
//************ INSERT NODE AT REAR ************

void insertrear(LIST *lp,int ele)
{
        NODE *newnode;
  if(lp->head == NULL)//empty list
        {
    insertfront(lp,ele);
    return;
  }
  newnode = getnode(ele);
  newnode->link = lp->head;//(lp->rear)->link;
  lp->rear->link=newnode;
        lp->rear=newnode;
        lp->count++;
}
//************ INSERT NODE AT POS ************

void insertatpos(LIST *lp, int ele,int pos)
{
        NODE *newnode,*temp;
        int num;
        if(pos<1 || pos>lp->count)
        {
                printf("Invalid Position");
                return;
        }
        if(pos==1)
        {
                insertfront(lp,ele);
                return;
        }
        if(pos==lp->count)
```

```
	{
		insertrear(lp,ele);
		return;
	}
	//insert at specified position
	num=1;
	temp=lp->head;
	newnode = getnode(ele);
	while(num<pos-1)
	{
		temp= temp->link;
		num++;
	}
	newnode->link = temp->link;
	temp->link = newnode;
	lp->count++;
	return;
}
//************DELETE FIRST ************

void deletefirst(LIST *lp)
{
	NODE *prev = lp->head,*first = lp->head;
	//If list empty
	if(lp->head == NULL)
	{
		printf("\nList is emnpty");
		return;
	}

	//If list has onlu 1 node
	if(prev->link == prev)
	{
		lp->head = lp->rear = NULL;
		lp->count--;
		return;
	}

	//traverse the list from second node to first node
	while(prev->link != lp->head)
	{
		prev = prev->link;
	}
	prev->link = first->link;
	lp->head = prev->link;
	free(first);
	lp->count--;
	return;
}
//************DELETE LAST************

void deletelast(LIST *lp)
{
	NODE *curr = lp->head,*temp = lp->head,*prev;
```

```
        //If list empty
        if(lp->head == NULL)
        {
                printf("\nList is empty");
                return;
        }
        //If list has single node
        if(curr->link == curr)
        {
                free(curr);
                lp->head = lp->rear = NULL;
                lp->count--;
                return;
        }
        //traversing the list till second last
        while(curr->link != lp->head)
        {
                prev = curr;
                curr = curr->link;
        }
        prev->link = lp->head;
        lp->rear=prev;
        free(curr);
        lp->count--;
        return;

}
//************ DELETE PARTICULAR NODE************
void deleteitm(LIST *lp,int key,int ele)
{
        if(lp->head == NULL)
        {
                printf("\nList Is Empty!!!");
                return;
        }

        NODE *curr,*prev;
        curr = lp->head;
        //if key is present in first node
        if(lp->head->data == key)
        {
                deletefirst(lp);
                return;
        }
        //if key is present in last node
        if(lp->rear->data == key)
        {
                deletelast(lp);
                return;
        }
        //traverse still the req node
        while(curr->data != key)
        {
                if(curr->link == lp->head)
        {
            printf("\nGiven Node Element Is Not Present!!");
            break;
        }
        prev = curr;
        curr = curr->link;
    }

    //checking if it is the only node
    if(curr->link == lp->head)
    {
        lp->head = lp->rear = NULL;
        free(curr);
        return;
    }

    //If more than one node in list
    curr = lp->head;
    while(curr != NULL)             //Checking other nodes
    {
        if(curr->data == key)       //If key matches
        {
            if(prev != NULL) //change the links
                prev->link = curr->link;

            free(curr);             //Delete the node
            lp->count--;
            return;
        }
        prev = curr;
        curr= curr->link;
    }
}
//************ DISPLAY************

void display(LIST lp)
{
    NODE *p=lp.head;
    if(p==NULL)
    {
        printf("list is empty");
        return;
    }
    do{
        printf("%d->",p->data);
        p=p->link;
    } while(p!=lp.head);
    printf("\nCount=%d\n",lp.count);
}
//************ DRIVER CODE************

int main(int argc, char *argv[]) {
    LIST lp;
    lp.count=0;
```

```
	lp.head=lp.rear=NULL;
	insertfront(&lp,30);
	insertfront(&lp,20);
	//display(lp);
	insertrear(&lp,40);
	insertatpos(&lp,50,2);
	display(lp);
	deletefirst(&lp);
	display(lp);
	deletelast(&lp);
	deleteitm(&lp,30);
	display(lp);
	return 0;
}
```

DOUBLY LINKED LIST :

```
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser or add your own getch, system("pause") or input loop
*/
typedef struct node
{
   int data;
   struct node *next;
   struct node *prev;
}NODE;

typedef struct
{
   int count;
   NODE *head,*rear;
}LIST;
NODE * getnode(int a)
{
	NODE *p;
	p=(NODE *)malloc(sizeof(NODE));
	p->data=a;
	p->next=p->prev=NULL;
	return p;
}
//************ INSERT NODE FRONT ************

void insertfront(LIST *lp,int a)
{
	NODE *newnode;
	newnode=getnode(a);
	//check if list is empty
	if(lp->head==NULL)
	{
		lp->head=lp->rear=newnode;
		lp->count++;
		return;
```

```
            }
            //insert in non empty list
            newnode->next = lp->head;
            (lp->head)->prev=newnode;
   lp->head = newnode;
   lp->count++;
   return;
}
```

//************ INSERT NODE REAR ************

```
void insertrear(LIST *lp,int ele)
{
   NODE *newnode;
   newnode=getnode(ele);
   //check if list is empty
   if(lp->head==NULL)
   {
      lp->head=newnode;//insertfront(lp,ele);
      lp->rear=newnode;
      lp->count++;
      return;
   }
   //for non empty list
   (lp->rear)->next=newnode;
   newnode->prev=lp->rear;
   newnode->next=NULL;
   lp->rear=newnode;
   lp->count++;
}
```

//************ DISPLAY************

```
void display(LIST lp)
{
            NODE *temp;
            //check for empty list
            if(lp.head==NULL)
            {
                        printf("\n List is empty");
                        return;
            }
            temp=lp.head;
            printf("NULL");
            while(temp!=NULL)
               {
      printf("<- %d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL");
   printf("\nNode Count: %d\n", lp.count);
}
```

//************ INSERT BEFORE KEY************

```
void insertbeforekey(LIST *lp,int ele,int key)
{
        NODE *t,*newnode;
        newnode=getnode(ele);
        if(lp->head==NULL)
  {
    lp->head=newnode;//insertfront(lp,ele);
    lp->rear=newnode;
    lp->count++;
    return;
  }
  //non empty list
  t=lp->head;
  while(t!=NULL&&t->data!=key)
  {
        t=t->next;
        }
        if(t==NULL)
        {
                printf("Key Element not found!!!");
                return;
        }
        newnode->next=t;
        newnode->prev=t->prev;
        t->prev->next=newnode;
        t->prev=newnode;
        lp->count++;
        return;

}
//************ DELETE FIRST************

void deletefirst(LIST *lp)
{
        NODE *temp;
        temp = lp->head;
        //If list is empty
        if(temp == NULL)
        {
                printf("List is Empty !!\n");
                return;
        }
        //If list contains only  1 node
        if(lp->head == lp->rear)
        {
                lp->head = lp->rear =NULL;
                free(temp);
                return;
        }
        else
        {
                temp = lp->head;
                lp->head = temp->next;
                temp->next->prev =NULL;
                lp->count--;
                free(temp);
        return;
    }
}
//************ DELETE REAR************

void deleterear(LIST *lp)
{
    //If list empty
    if(lp->head == NULL)
    {
        printf("List is Empty !!\n");
        return;
    }
    NODE *temp;
    temp = lp->rear;
    //If list contains only  1 node
    if(lp->head == lp->rear)
    {
        lp->head = lp->rear =NULL;
        free(temp);
        return;
    }
    else
    {
        temp = lp->rear;
        lp->rear = temp->prev;
        temp->prev->next = NULL;
        lp->count--;
        free(temp);
        return;
    }
}
//************ DELETE ITEM************

void deleteitm(LIST *lp,int key)
{
    NODE *curr = lp->head,*prev;
    //If list empty
    if(lp->head == NULL)
    {
        printf("\nList Empty!!!\n");
        return;
    }
    //If key present in first node
    if(lp->head->data == key)
    {
        deletefirst(lp);
        return;
    }
    //If key present in lst node
    if(lp->rear->data == key)
    {
        deleterear(lp);
        return;
        }
        curr = lp->head;
        //traverse still we find key
        while(curr->data != key)
        {
                if(curr->next == lp->rear)
                {
                        printf("\nKey element not found!!\n");
                        break;
                }
                prev = curr;
                curr = curr->next;
        }
        prev->next = curr->next;
        curr->next->prev =prev;
        free(curr);
        lp->count--;
        return;

}
//************DISPLAY REVERSE************

displayreverse(LIST l)//display from right to left
{
         struct node *temp;
  if (l.head == NULL)
  {
    printf("The list is empty!");
    return;
  }
  temp = l.rear;
  printf("\nNULL");
  while (temp != NULL)
  {
    printf("<- %d -> ", temp->data);
    temp = temp->prev;
  }
  printf("NULL");
  printf("\nNode Count: %d\n", l.count);
}
int main(int argc, char *argv[]) {
        LIST lp;
        lp.count=0;
        lp.head=lp.rear=NULL;
        int ele;
        insertfront(&lp,15);
        insertfront(&lp,5);
        insertfront(&lp,25);
        insertfront(&lp,4);
        insertfront(&lp,55);
        display(lp);
        insertrear(&lp,111);
        insertrear(&lp,121);
        insertbeforekey(&lp,110,121);
```

```
            display(lp);
            deletefirst(&lp);
    display(lp);
    deleterear(&lp);
    display(lp);
    printf("\nEnter the element to be deleted :: ");
    scanf("%d",&ele);
    deleteitm(&lp,ele);
    display(lp);
            displayreverse(lp);
}
```

LIST ADT :

//********** DECLARATION**************

```
LIST* createList (int (*compare)(void* argu1, void* argu2));
LIST* destroyList (LIST* list);
int addNode (LIST* pList, void* dataInPtr);
bool removeNode (LIST* pList,
void* keyPtr,
void** dataOutPtr);
bool searchList (LIST* pList,
void* pArgu,
void** pDataOut);


bool retrieveNode (LIST* pList,
void* pArgu,
void** dataOutPtr);
bool traverse (LIST* pList,
int fromWhere,
void** dataOutPtr);
int listCount (LIST* pList);
bool emptyList (LIST* pList);
bool fullList (LIST* pList);
static int _insert (LIST* pList,
NODE* pPre,
void* dataInPtr);
static void _delete (LIST* pList,
NODE* pPre,
NODE* pLoc,
void** dataOutPtr);
static bool _search (LIST* pList,
NODE** pPre,
NODE** pLoc,
void* pArgu);
//End of List ADT Definitions
```

/*=============== createList ============== */

```
LIST* createList
(int (*compare) (void* argu1, void* argu2))
{
//Local Definitions
LIST* list;
//Statements
list = (LIST*) malloc (sizeof (LIST));
if (list)
{
list->head = NULL;
list->pos = NULL;
list->rear = NULL;
list->count = 0;
list->compare = compare;
} // if
return list;
} // createList
```

/*================== addNode =================*/

```
int addNode (LIST* pList, void* dataInPtr)
{
//Local Definitions
bool found;
bool success;
NODE* pPre;
NODE* pLoc;
//Statements
found = _search (pList, &pPre, &pLoc, dataInPtr);
if (found)
// Duplicate keys not allowed
return (+1);
success = _insert (pList, pPre, dataInPtr);
if (!success)
// Overflow
return (-1);
return (0);
} // addNode
```

/*=================== _insert ================== */

```
static bool _insert (LIST* pList, NODE* pPre,
void* dataInPtr)
{
//Local Definitions
NODE* pNew;
//Statements
if (!(pNew = (NODE*) malloc(sizeof(NODE))))
return false;
pNew->dataPtr = dataInPtr;
pNew->link = NULL;
if (pPre == NULL)
{
// Adding before first node or to empty list.
pNew->link = pList->head;
pList->head = pNew;
if (pList->count == 0)
// Adding to empty list. Set rear
pList->rear = pNew;
} // if pPre
else
{
// Adding in middle or at end
pNew->link = pPre->link;
pPre->link = pNew;
// Now check for add at end of list
ƒif (pNew->link == NULL)
ƒ pList->rear = pNew;
} // if else
(pList->count)++;
return true;
} // _insert
```

/*================= removeNode ================*/

```
bool removeNode ƒ(LIST* pList, void* keyPtr,
ƒvoid** dataOutPtr)
{
//Local Definitions
bool found;
NODE* pPre;
NODE* pLoc;
//Statements
found = _search (pList, &pPre, &pLoc, keyPtr);
if (found)
_delete (pList, pPre, pLoc, dataOutPtr);
return found;
} // removeNode
```

/*================= _delete ================ */

```
void _delete (LIST* pList, NODE* pPre,
NODE* pLoc, void** dataOutPtr)
{
//Statements
*dataOutPtr = pLoc->dataPtr;
if (pPre == NULL)
// Deleting first node
pList->head = pLoc->link;
else
// Deleting any other node
pPre->link = pLoc->link;
// Test for deleting last node
if (pLoc->link == NULL)
pList->rear = pPre;
(pList->count)--;
free (pLoc);
return;
} // _delete
```

/*================== searchList =================*/

```
bool searchList (LIST* pList, void* pArgu,
void** pDataOut)
{
//Local Definitions
bool found;
NODE* pPre;
NODE* pLoc;
//Statements
found = _search (pList, &pPre, &pLoc, pArgu);
if (found)
*pDataOut = pLoc->dataPtr;
else
*pDataOut = NULL;
return found;
} // searchList
```

/*================== _search =================*/

```
bool _search (LIST* pList, NODE** pPre,
NODE** pLoc, void* pArgu)
{
//Macro Definition
#define COMPARE \
( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )

#define COMPARE_LAST \
((* pList->compare) (pArgu, pList->rear->dataPtr))
//Local Definitions
int result;
//Statements
*pPre = NULL;
*pLoc = pList->head;
if (pList->count == 0)
return false;
// Test for argument > last node in list
if ( COMPARE_LAST > 0)
{
*pPre = pList->rear;
*pLoc = NULL;
return false;
} // if
while ( (result = COMPARE) > 0 )
{
// Have not found search argument location
*pPre = *pLoc;
*pLoc = (*pLoc)->link;
} // while
if (result == 0)
// argument found--success
return true;
else
return false;
} // _search


/*================= retrieveNode ================*/
static bool retrieveNode (LIST* pList,
ffvoid* pArgu,
f void** dataOutPtr)
{
//Local Definitions
bool found;
NODE* pPre;
NODE* pLoc;
//Statements
found = _search (pList, &pPre, &pLoc, pArgu);
if (found)
{
*dataOutPtr = pLoc->dataPtr;
return true;
} // if
*dataOutPtr = NULL;
return false;
} // retrieveNode


/*================= emptyList ================*/
bool emptyList (LIST* pList)
{
//Statements
return (pList->count == 0);
} // emptyList


/*================= fullList =================*/
bool fullList (LIST* pList)
{
//Local Definitions
NODE* temp;
//Statements
if ((temp = (NODE*)malloc(sizeof(*(pList->head)))))
```

```
{
free (temp);
return false;
} // if

// Dynamic memory full
return true;
} // fullList


/*================== listCount ==================*/
int listCount(LIST* pList)
{
//Statements
return pList->count;
} // listCount

/*================== traverse =================*/

bool traverse (LIST* pList,
int fromWhere,
void** dataPtrOut)
{
//Statements
if (pList->count == 0)
return false;
if (fromWhere == 0)
{
// Start from first node
pList->pos = pList->head;
*dataPtrOut = pList->pos->dataPtr;
return true;
} // if fromwhere
else
{
// Start from current position
if (pList->pos->link == NULL)
return false;
else
{
pList->pos = pList->pos->link;
*dataPtrOut = pList->pos->dataPtr;
return true;
} // if else
} // if fromwhere else
} // traverse


/*================== destroyList =================*/
LIST* destroyList (LIST* pList)
{
//Local Definitions
NODE* deletePtr;
//Statements
```

```
if (pList)
{
while (pList->count > 0)
{
ƒ// First delete data
free (pList->head->dataPtr);
// Now delete node
deletePtr = pList->head;
pList->head = pList->head->link;
pList->count--;
free (deletePtr);
} // while
free (pList);
} // if
return NULL;
} // destroyList
```