Name – Yashwant Chandrakant Bhosale
Div – SY comp Div 1
MIS – 612303039

## Lab Work 1 : Singly Linked List

### Question:

Define ADT SLL ( Singly Linked List of integers). Write the following functions with suitable prototypes for ADT SLL:

init_SLL()  //  to initialize the list

append()  // to add an element at the end of the list.

traverse() //   to display all the list elements

insert_at_beg() // to add an element at the beginning of the list

remove_at_pos() // to remove an element from the list from the given position

len() //returns the length of the list.

/*

You are free to include more functions.

The skeleton of function main() is given below. Use the same by replacing commented statements with actual function calls:

```
int main() {

  SLL L1;

  //call init()

  // call append() multiple times to insert elements in the respective list as in the

  // call traverse()

  // call

  // insert_at_beg()

  // remove_at_pos()

  // len()

  return 0;

}
```

Code:

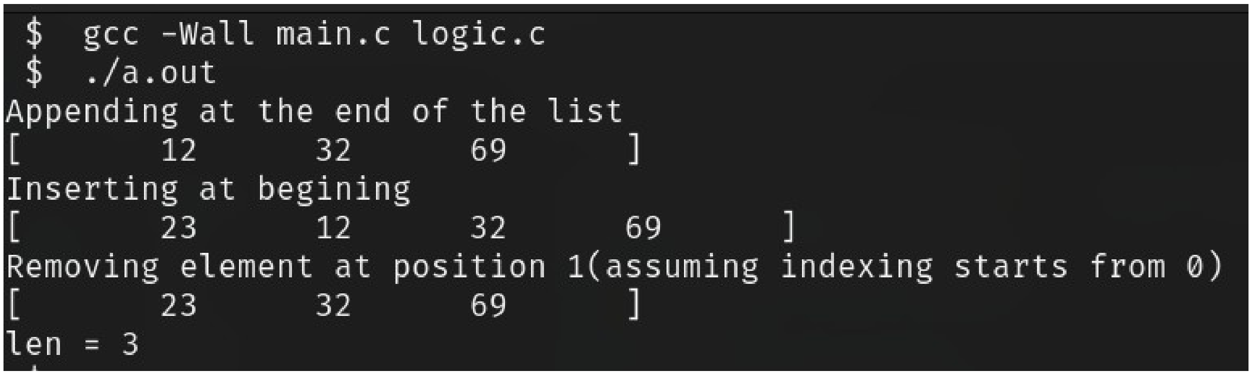
```
/* list.h: contains structure declarations and function prototypes */
typedef struct node {
    int data;
    struct node *next;
} node;

typedef node* list;

void init_sll(list *l);
void append(list *l, int data);
void traverse(list l);
void insert_at_beg(list *l, int data);
void remove_at_pos(list *l, int pos);
int len(list l);

/* logic.c: contains function definitions */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

void init_sll(list *l) {
    *l = NULL;
    return;
}

/* Function to append new node at the end of the list */
void append(list *l, int data) {
    node *nn = (node *) malloc(sizeof(node));
    nn -> data = data;
    nn -> next = NULL; /* set next of new node to NULL as it will be last
node */

    /* if list is empty, update head */
    if(*l == NULL) {
        *l = nn;
    }
    /* else traverse to the end of the list to append new node */
    else {
        node *p = *l;
        while(p -> next)
            p = p->next;
        p -> next = nn;
    }
    return;
}
```

```
/* Function to display elements in the list */
void traverse(list l) {
    node *p = l;
    printf("[\t");
    while(p) {
        printf("%d\t", p->data);
        p = p->next;
    }
    printf("]\n");
    return;
}

/* Function to insert node at the begining of the list */
void insert_at_beg(list *l, int data) {
    node *nn = (node *) malloc(sizeof(node));
    nn -> data = data;
    nn -> next = *l; /* next is current head of the list, *l can be NULL or
a node */
    *l = nn;
    return;
}

/* Function to remove element at specified position (Assuming indexing
start at position 0) */
void remove_at_pos(list *l, int index) {
    int i = 0;
    node *p = *l, *q = NULL;

    /* Conditions in the loop
     * 1. i < index-1: loop will terminate at one node before the required
node so we can modify next link of the previous node
     * 2. p->next: this checks whether we have reached end of the list
     */
    while(i < index-1 && p->next) {
        p = p->next;
        i++;
    }
    q = p -> next; /* store address of the node to be deleted */
    p -> next = q -> next; /* update next node link of previous node */
    free(q);
    return;
}
```

```
/* Function to find out length of the list */
int len(list l) {
    node *p = l;
    int len = 0;
    while(p) {
        p=p->next;
        len++;
    }
    return len;
}
```

```
/* main.c: Contains main flow of the program */
#include <stdio.h>
#include "list.h"

int main() {
    list l;
    init_sll(&l);
    printf("Appending at the end of the list\n");
    append(&l, 12);
    append(&l, 32);
    append(&l, 69);
    traverse(l);
    printf("Inserting at begining\n");
    insert_at_beg(&l, 23);
    traverse(l);
    printf("Removing element at position 1(assuming indexing starts from 
0)\n");
    remove_at_pos(&l, 1);
    traverse(l);
    printf("len = %d\n", len(l));
    return 0;
}
```

**Output:**



```
 $  gcc -Wall main.c logic.c
 $  ./a.out
Appending at the end of the list
[       12      32      69      ]
Inserting at begining
[       23      12      32      69      ]
Removing element at position 1(assuming indexing starts from 0)
[       23      32      69      ]
len = 3
```

# Lab Work 2: Doubly Linked List

## Question:

Write following functions with suitable prototypes for ADT Doubly Linked List :

init_DLL()  // initiliazes doubly linked list
insert_beg( )  //    to add an element in the end of the  DLL.
insert_end( )  //    to add an element in the beginning of the  DLL.
insert_pos( )  //    to add an element at the position specified by user of the  DLL.
remove_beg() // deletes the first node of the DLL
remove_end() // deletes the last node of the DLL
remove_pos( )  //    to delete an element at the position specified by user of the  DLL.
sort()   // sort the DLL
displayRL()  //   to display all the elements of the list starting from right element to left.
displayLR()  //   to display all the elements of the list starting from left element to right.
is_palindrome()  /*The functions determines the given DLL  is a palindrome or not. For example, if the list is:  {1, 2, 3, 2, 1} is a palindrome. For example, if the DLL is:  {1, 2, 3, 1, 2, 1} is a not a palindrome.*/
remove_duplicates() /* The functions removes a duplicate node keeping only one node so that elements of node are unique.
For example, if the DLL is:  {1, 2, 3, 2, 1} after the function is invoked the DLL changes to:  {1, 2, 3 } */
You are free to include more functions.
Skeleton of function main() is given below, use same by replacing commented statements by actual function calls:

```
int main() {
DLL L1;
//call init() // call init for  list

// call insert_beg( )  // call multiple times
// insert_end( )  //   // call multiple times
// call displayLR()

// call insert_pos( )
// call displayRL()

// call is_palindrome()
// call remove_beg()
// call displayLR()

// call remove_end()
// call displayLR()

// call remove_pos()
// call displayLR()

return 0;
}
```

Code:

```
/* list.h: Contains function prototypes and struct declarations */
typedef struct node {
    struct node *prev;
    int data;
    struct node *next;
} node;

typedef struct {
    struct node *head, *tail;
} list;
void init(list *l);
void insert_end(list *l, int data);
void insert_from_begining(list *l, int data);
void insert_at_index(list *l, int data, int index);
void remove_at_index(list *l, int index);
void sort(list l);
void remove_end(list *l);
void remove_beg(list *l);
void remove_duplicates(list *l);
void printLR(list l);
void printRL(list l);

/* list.c: contains function definitions for above functions */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "list.h"
#include "hash.h"

void init (list *l) {
    l -> head = NULL;
    l -> tail = NULL;
    return;
}

int isEmpty(list l) {
    if(!l.head)
        return 1;
    return 0;
}
```

```
void insert_end (list *l, int data) {
    node *nn = (node *) malloc(sizeof(node));
    nn -> next = NULL;
    if (l -> head == NULL) {
        nn -> prev = NULL;
        nn -> data = data;
        l -> tail = nn;
        l -> head = nn;
    }else {
        nn -> data = data;
        l -> tail -> next = nn;
        nn -> prev = l -> tail;
        l -> tail = nn;
    }
    return;
}

void insert_from_begining(list *l, int data) {
    node *nn;
    nn = (node *) malloc(sizeof(node));
    nn->data = data;
    if(l -> head == NULL) {
        nn -> next = NULL;
        nn -> prev = NULL;
        l ->head = nn;
    }else {
        nn -> next = l -> head;
        nn -> prev = NULL;
        l -> head -> prev = nn;
        l -> head = nn;
    }
    return;
}


void insert_at_index(list *l, int data, int index) {
    int i = 0;
    node *nn = (node *) malloc(sizeof(node)), *p = l->head;
    nn -> data = data;
    while(i < index-1) {  // index-1 because we want to reach just one node
before the required node
        if(p->next == NULL)
            return; // invalid index
        p = p->next;
        i++;
    }
    nn -> next = p -> next;
    nn -> prev = p;
    nn -> next -> prev = nn;
    p -> next = nn;
    return;
}
```

```
void remove_at_index(list *l, int index) {
    int i = 0;
    node *p = l -> head, *q=NULL;
    while(i < index-1) { // index-1 because we want to reach just one node
before the required node
        if(p->next == NULL)
            return; // invalid index
        p = p -> next;
        i++;
    }
    q = p->next;
    p -> next = q -> next;
    q -> next -> prev = p;
    free(q);
    return;
}

void remove_end(list *l) {
    node *p = l -> tail;
    if (!p) {
        return;
    }
    if(p -> prev){
        l -> tail = p -> prev;
        p -> prev -> next = NULL;
    }
    else
        l -> tail = NULL;
    free(p);
    return;
}
void remove_beg(list *l) {
    node *p = l -> head;
    if(!p) {
        return;
    }
    if(p -> next){
        l -> head = p -> next;
        p -> next -> prev = NULL;
    }
    else
        l -> head = NULL;
    free(p);
    return;
}
void swap(node *n1, node *n2) {
    int temp = n1 -> data;
    n1 -> data = n2 -> data;
    n2 -> data = temp;
    return;
}
```

```c
int list_len(list l) {
    int len=0;
    node *p = l.head;
    while(p) {
        len++;
        p = p->next;
    }
    return len;
}

void sort(list l) {
    node *p = l.head, *q = NULL;
    int len = list_len(l), sorted = 0; // Keep track of no. of sorted
elements to avoid repeated comparisons
    while(p->next) {
        q = l.head;
        int j = 0;
        while(j < (len-sorted) && q->next) {
            if(q->data > q->next->data)
                swap(q, q->next);
            q = q->next;
            j++;
        }
        p = p->next;
        sorted++;
    }
    return;
}


/* Definitions and declarations for hash table functions on further pages*/
void remove_duplicates(list *l) {
    node *p = l -> head;
    // Simple hash table to keep track of elements
    int len = list_len(*l);
    int hash_table[len];
    init_ht(hash_table, len);
    for (int i = 0; i < len; i++) {
        int index = hash(p -> data, len);
        node *next = p->next;
        if(!insert(hash_table, p->data, index, len)) {
            node *q = p -> prev;
            q -> next = p -> next;
            p -> next -> prev = q;
            free(p);
        }
        p = next;
    }
}
```

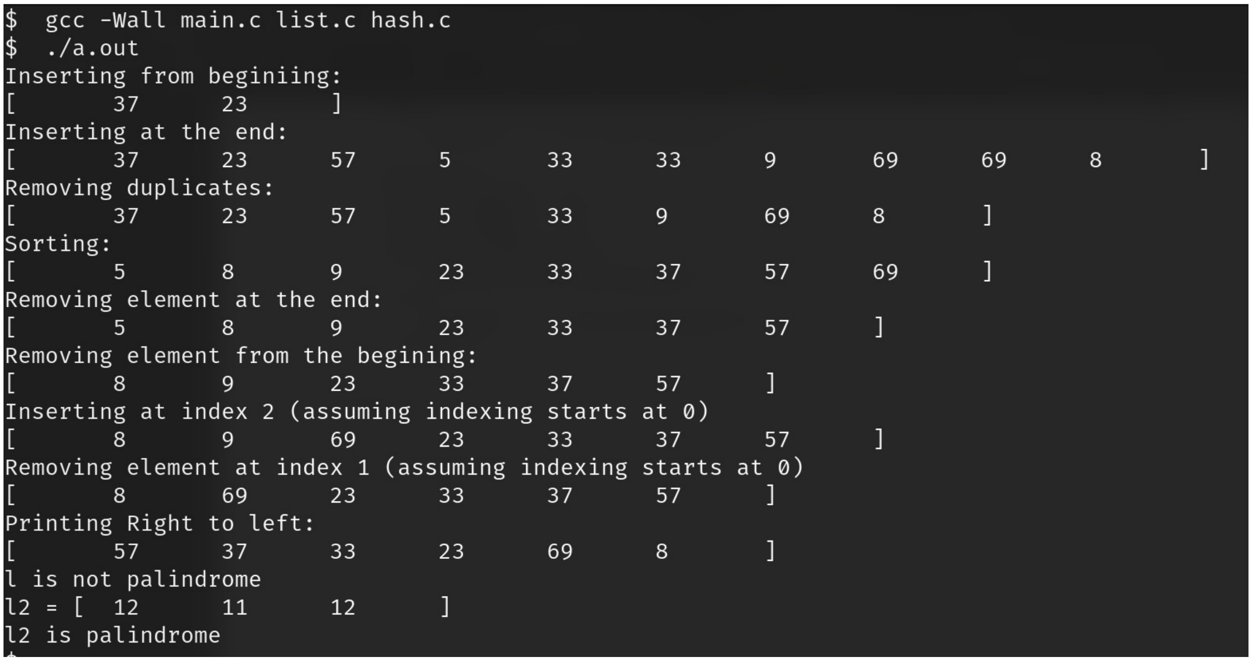
```
void printLR(list l) {
    node *p;
    p = l.head;
    printf("[\t");
    if(!p){
        printf("]");
        return;
    }
    while(p) {
        printf("%d\t", p->data);
        p = p->next;
    }
    printf("]\n");
    return;
}


void printRL(list l) {
    node *p;
    p = l.tail;
    printf("[\t");
    if(!p) {
        printf("]\n");
        return;
    }
    while(p) {
        printf("%d\t", p -> data);
        p = p->prev;
    }
    printf("]\n");
    return;
}
```

Use of hash table:
Program involves a function for removing duplicates. Rather than traversing through whole array repeatedly to check for duplicated hash table is used to maintain track of elements. Linear Probing is used to prevent collisions.

```
/ * hash.h: Contains function prototypes for hash table */
void init_ht(int *ht, int len);
int hash(int key, int len);
short int insert(int *arr, int element, int index, int len);

/* hash.c: function definitions for hash table functions */
#include <limits.h>

void init_ht(int *ht, int len) {
    for (int i = 0; i < len; i++) {
        ht[i] = INT_MIN;
    }
    return;
}

int hash(int key, int len) {
    int hash = key * 31; // Multiply by a prime number
    return hash % len;
}

short int insert(int *arr, int element, int index, int len) {
    if(arr[index] == INT_MIN) {
        arr[index] = element;
    }
    else {
        if(arr[index] == element)
            return 0;
        while(arr[index] != INT_MIN) {
            if(arr[index] == element)
                return 0;
            index = (index+1) % len;
        }
        arr[index] = element;
    }
    return 1;
}
```

```
/* main.c: contains main flow of the program */
#include <stdio.h>
#include "list.h"

int main() {
    list l, l2;
    init(&l);
    printf("Inserting from beginiing: \n");
    insert_from_begining(&l, 23);
    insert_from_begining(&l, 37);
    printLR(l);

    printf("Inserting at the end: \n");
    insert_end(&l, 57);
    insert_end(&l, 5);
    insert_end(&l, 33);
    insert_end(&l, 33);
    insert_end(&l, 9);
    insert_end(&l, 69);
    insert_end(&l, 69);
    insert_end(&l, 8);
    printLR(l);

    printf("Removing duplicates: \n");
    remove_duplicates(&l);
    printLR(l);

    printf("Sorting: \n");
    sort(l);
    printLR(l);

    printf("Removing element at the end: \n");
    remove_end(&l);
    printLR(l);

    printf("Removing element from the begining: \n");
    remove_beg(&l);
    printLR(l);

    printf("Inserting at index 2 (assuming indexing starts at 0) \n");
    insert_at_index(&l, 69, 2);
    printLR(l);

    printf("Removing element at index 1(assuming indexing starts at 0)\n");
    remove_at_index(&l, 1);
    printLR(l);

    printf("Printing Right to left: \n");
    printRL(l);
    if(is_palindrome(l))
        printf("l is palindrome\n");
    else
        printf("l is not palindrome\n");

    init(&l2);
    insert_end(&l2, 12);
    insert_end(&l2, 11);
    insert_end(&l2, 12);
    printf("l2 = ");
    printLR(l2);
    if(is_palindrome(l2))
        printf("l2 is palindrome\n");
    else
        printf("l2 is not palindrome\n");
    return 0;
}
```

Output:



```
$  gcc -Wall main.c list.c hash.c
$  ./a.out
Inserting from beginiing:
[       37      23      ]
Inserting at the end:
[       37      23      57      5       33      33      9       69      69      8       ]
Removing duplicates:
[       37      23      57      5       33      9       69      8       ]
Sorting:
[       5       8       9       23      33      37      57      69      ]
Removing element at the end:
[       5       8       9       23      33      37      57      ]
Removing element from the begining:
[       8       9       23      33      37      57      ]
Inserting at index 2 (assuming indexing starts at 0)
[       8       9       69      23      33      37      57      ]
Removing element at index 1 (assuming indexing starts at 0)
[       8       69      23      33      37      57      ]
Printing Right to left:
[       57      37      33      23      69      8       ]
l is not palindrome
l2 = [  12      11      12      ]
l2 is palindrome
```

# Lab Work 3: Circular Linked List

## Question:

Write following functions with suitable prototypes for ADT Circular Linked List :

init_CLL()  // initiliazes doubly linked list

insert_beg( )  //    to add an element in the end of the  CLL.

insert_end( )  //    to add an element in the beginning of the  CLL.

insert_pos( )  //    to add an element at the position specified by user of the  CLL.

remove_beg() // deletes the first node of the CLL

remove_end() // deletes the last node of the CLL

remove_pos( )  //    to delete an element at the position specified by user of the  CLL.

sort()   // sort the CLL

display()  //   to display all the elements of the list

You are free to include more functions.

Skeleton of function main() is given below, use same by replacing commented statements by actual function calls:

```
int main() {

CLL L1;

//call init() // call init for  list

// call insert_beg( )  // call multiple times

// insert_end( )  //   // call multiple times

// call display()

// call insert_pos( )

// call remove_beg()

// call remove_end()

// call remove_pos()

return 0;
}
```

Code:

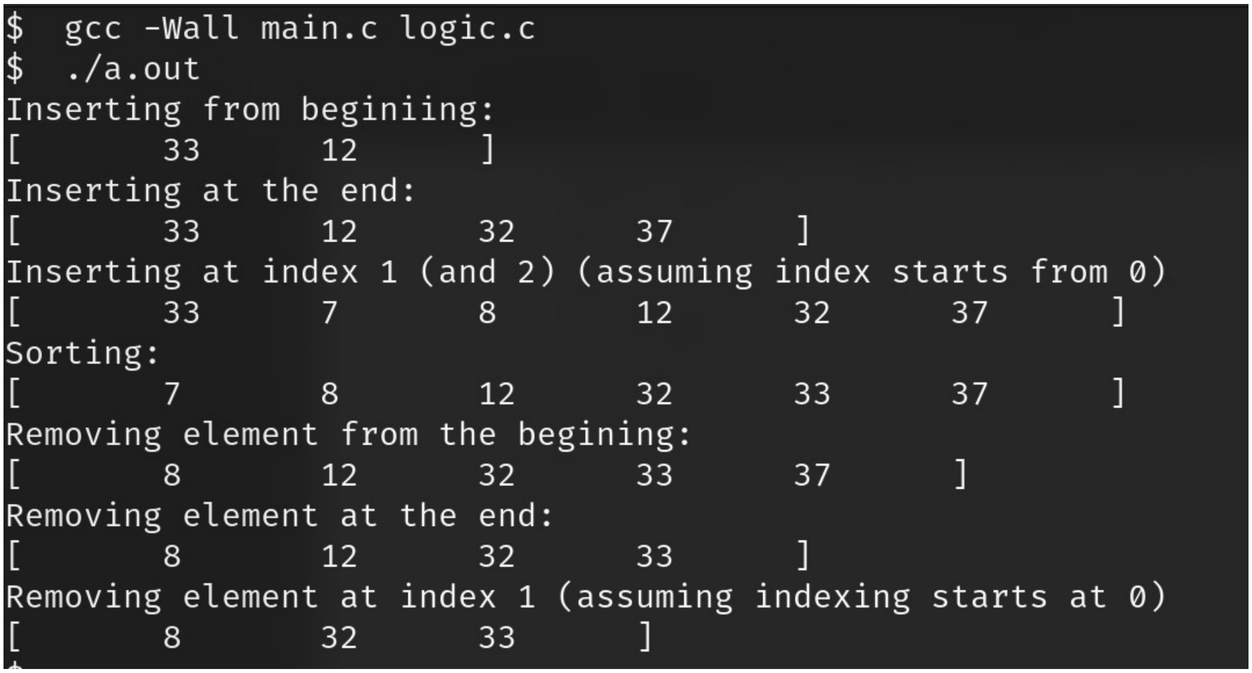
```c
/* cll.h: contains struct declarations and function prototypes for cll */
typedef struct node {
    int data;
    struct node *next;
} node;

typedef node *list;

void init_cll(list *l);
void insert_beg(list *l, int data);
void insert_end(list *l, int data);
void insert_pos(list *l, int pos, int data);
void remove_beg(list *l);
void remove_end(list *l);
void remove_pos(list *l, int pos);
void sort(list *l);
void display(list l);

/* logic.c: contains function definitions for cll */
#include <stdio.h>
#include <stdlib.h>
#include "cll.h"

void init_cll(list *l) {
    *l = NULL;
    return;
}

void insert_beg(list *l, int data) {
    node *nn = (node *) malloc(sizeof(node)), *p = NULL;
    nn -> data = data;
    if(*l == NULL) {
        *l = nn;
        nn -> next = *l;
    } else {
        p = *l;
        nn -> next = *l;
        while(p -> next != *l) {
            p = p -> next;
        }
        p -> next = nn;
        *l = nn;
    }
    return;
}
```

```c
void insert_end(list *l, int data) {
    node *nn = (node *) malloc(sizeof(node)), *p = NULL;
    nn->data = data;
    if(*l == NULL) {
        *l = nn;
        nn -> next = *l;
    } else {
        p = *l;
        while(p -> next != *l) {
            p = p->next;
        }
        p->next = nn;
        nn->next = *l;
    }
    return;
}

void insert_pos(list *l, int pos, int data) {
    int i = 0;
    node *nn = (node *) malloc(sizeof(node)), *p = NULL;
    nn -> data = data;
    p = *l;
    if(!p) {
        *l = nn;
        nn -> next = *l;
    }else {
        while(i < pos-1) {
            if(p -> next == *l)
                return; // Invalid position
            p = p->next;
            i++;
        }
        nn -> next = p->next;
        p -> next = nn;
    }
    return;
}

void remove_beg(list *l) {
    if(*l == NULL)
        return;
    node *p = *l, *q = *l;
    while(q -> next != *l) {
        q = q->next;
    }
    *l = (*l) -> next;
    q -> next = *l;
    free(p);
    return;
}
```

```c
void remove_end(list *l) {
    if(*l == NULL)
        return;
    node *p = *l, *q = NULL;
    while(p -> next != *l) {
        p = p->next;
        if(p -> next -> next == *l) {
            q = p;
        }
    }
    q -> next = *l;
    free(p);
    return;
}

void remove_pos(list *l, int pos) {
    if(*l == NULL)
        return;
    int i = 0;
    node *p = *l, *q = NULL;
    while(i < pos-1) {
        if(p -> next == *l)
            return; // Invalid positionI
        p = p->next;
        i++;
    }
    q = p -> next;
    p -> next = q -> next;
    free(q);
    return;
}

void swap(node *n1, node *n2) {
    int temp = n1 -> data;
    n1 -> data = n2 -> data;
    n2 -> data = temp;
    return;
}
```

```
/* Bubble sort on linked list O(n^2) */
void sort(list *l) {
    node *p, *q;
    p = *l;
    while(p -> next != *l) {
        q = *l;
        while(q -> next != *l) {
            if(q -> data > q -> next -> data)
                swap(q, q->next);
            q = q -> next;
        }
        p = p->next;
    }
    return;
}




void display(list l ) {
    node *p = l;
    printf("[\t");
    while(p && p -> next != l) {
        printf("%d\t", p->data);
        p = p -> next;
    }
    printf("%d\t", p->data);
    printf("]\n");
    return;
}

/* main.c: Contains main flow of the program */
#include <stdio.h>
#include "cll.h"

int main() {
    list l;
    init_cll(&l);
    printf("Inserting from beginiing: \n");
    insert_beg(&l, 12);
    insert_beg(&l, 33);
    display(l);

    printf("Inserting at the end: \n");
    insert_end(&l, 32);
    insert_end(&l, 37);
    display(l);

    printf("Inserting at index 1 (and 2)(assuming index starts from 0)\n");
    insert_pos(&l, 1, 7);
    insert_pos(&l, 2, 8);
    display(l);
    printf("Sorting: \n");
    sort(&l);
    display(l);

    printf("Removing element from the begining: \n");
    remove_beg(&l);
    display(l);

    printf("Removing element at the end: \n");
    remove_end(&l);
    display(l);

    printf("Removing element at index 1(assuming indexing starts at 0)\n");
    remove_pos(&l, 1);
    display(l);
    return 0;
}
```

**Output:**



```
$  gcc -Wall main.c logic.c
$  ./a.out
Inserting from beginiing:
[       33      12      ]
Inserting at the end:
[       33      12      32      37      ]
Inserting at index 1 (and 2) (assuming index starts from 0)
[       33      7       8       12      32      37      ]
Sorting:
[       7       8       12      32      33      37      ]
Removing element from the begining:
[       8       12      32      33      37      ]
Removing element at the end:
[       8       12      32      33      ]
Removing element at index 1 (assuming indexing starts at 0)
[       8       32      33      ]
```

# Lab Work 4: Stack

## Question:

Implement a stack of integers using array. Invoke all stack functions using a menu driven program.

**Code:**

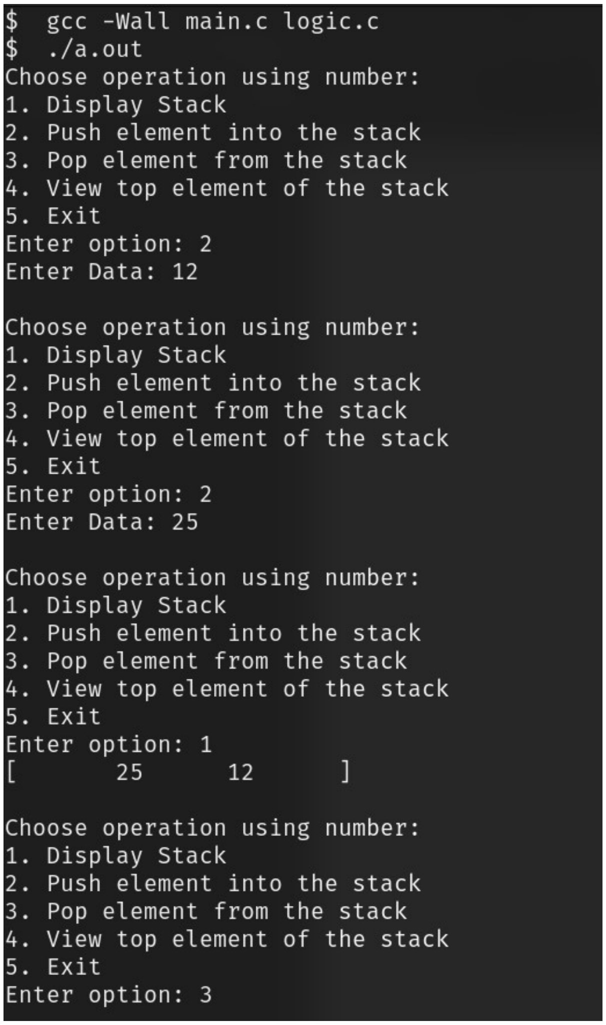
```
/* stack.h: Contains struct declarations and function prototypes */
typedef struct {
    int *arr;
    int size;
    int top;
} stack;

void init(stack *s, int size);
void push(stack *s, int data);
int pop(stack *s);
int peek(stack s);
void display(stack s);

/* logic.c: contains function definitions for functions */
#include "stack.h"

/* Function to initialize stack */
void init(stack *s, int size) {
    s -> arr = (int *) malloc(size * sizeof(int));
    s -> top = -1;
    s -> size = size;
    return;
}

/* Function to push an element in the stack */
void push(stack *s, int data) {
    if(s->top >= s->size-1){
        s -> arr = (int *) realloc(s->arr, (s->size+1) * sizeof(int));
    }
    s->top++;
    s->arr[s->top] = data;
    return;
}

/* Function to pop an element from the stack */
int pop(stack *s) {
    int element = s->arr[s->top];
    s->top--;
    return element;
}

/* Function to peek into the stack */
int peek(stack s) {
    return s.arr[s.top];
}
```

```
/* Function to display the stack */
void display(stack s) {
    printf("[\t");
    for(int i = s.top; i >= 0; i--) {
        printf("%d\t", s.arr[i]);
    }
    printf("]\n");
    return;
}

/* main.c: menu driven flow for the program */
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

void display_menu() {
    printf("Choose operation using number:\n");
    printf("1. Display Stack\n");
    printf("2. Push element into the stack\n");
    printf("3. Pop element from the stack\n");
    printf("4. View top element of the stack\n");
    printf("5. Exit\n");
    return;
}

void read_option(int option, stack *s) {
    switch (option){
    case 1:{
        display(*s);
        break;
    }
    case 2: {
        int data;
        printf("Enter Data: ");
        scanf("%d", &data);
        push(s, data);
        break;
    }
    case 3: {
        pop(s);
        break;
    }
    case 4: {
        printf("top = %d\n", peek(*s));
        break;
    }
    default:
        printf("Invalid Option\n");
        break;
    }
    return;
}
```

```
int main() {
    int option = 0;
    stack s;
    init(&s, 3);
    while(1) {
        display_menu();
        printf("Enter option: ");
        scanf("%d", &option);
        if(option == 5)
            break;
        read_option(option, &s);
        printf("\n");
    }

    return 0;
}
```

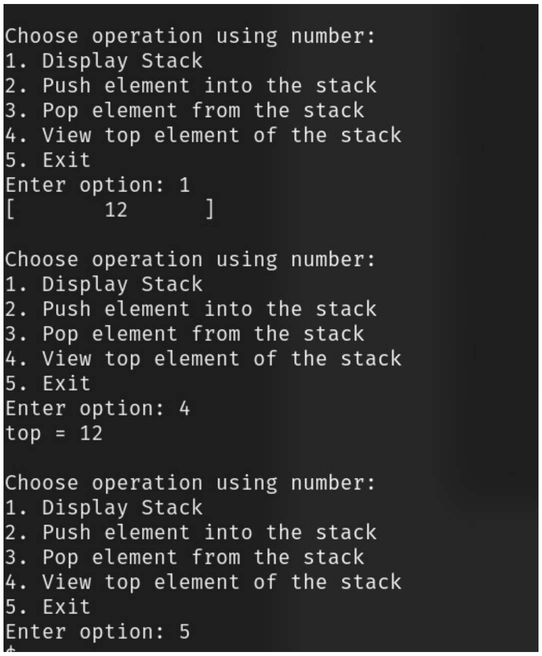
**Output:**

**(1)**



```
$  gcc -Wall main.c logic.c
$  ./a.out
Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 2
Enter Data: 12

Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 2
Enter Data: 25

Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 1
[       25      12      ]

Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 3
```

**(2)**



```
Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 1
[       12      ]

Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 4
top = 12

Choose operation using number:
1. Display Stack
2. Push element into the stack
3. Pop element from the stack
4. View top element of the stack
5. Exit
Enter option: 5
```

# Lab work 5: Stack Application

## Question:

Write a C program to convert an Infix expression to Postfix form using ADT stack. Further, evaluate the obtained postfix expression. The program should handle multiple digits and only valid infix expressions.

char_stack.h and int_stack.h → used seperate declarations for character stack (for conversion to postfix) and integer stack (for evaluation), although functionality stays same.

**Code:**

```c
/* char_stack.h: This is header file containig function prototypes for
character stack */
typedef struct Char_stack {
    int *arr;
    int size;
    int top;
} char_stack;

void init_char_stack(char_stack *s, int size);
void push_cs(char_stack *s, int data);
int pop_cs(char_stack *s);
int peek_cs(char_stack s);
short int is_cs_empty(char_stack s);
void view_char_stack(char_stack s);

/* char_stack.c: function definitions for character stack */
#include <stdio.h>
#include <stdlib.h>
#include "char_stack.h"

void init_char_stack(char_stack *s, int size) {
    s->arr = (int *)malloc(sizeof(int) * size);
    s->size = size;
    s->top = -1;
    return;
}

void push_cs(char_stack *s, int data) {
    s->arr[++s->top] = data;
    return;
}

int pop_cs(char_stack *s) {
    int element = s->arr[s->top];
    s->top--;
    return element;
}

int peek_cs(char_stack s) { return s.arr[s.top]; }

short int is_cs_empty(char_stack s) { return s.top == -1; }
```

```
void view_char_stack(char_stack s) {
    printf("[\t");
    for (int i = 0; i <= s.top; i++) {
        printf("%c\t", s.arr[i]);
    }
    printf("]\n");
}

/* int_stack.h: function prototypes and struct declaration for integer
stack */
typedef struct int_stack {
    int *arr;
    int size;
    int top;
} int_stack;

void init_int_stack(int_stack *s, int size);
void push_int(int_stack *s, int data);
int pop_int(int_stack *s);
int peek_int(int_stack s);
short int is_int_empty(int_stack s);
void view_int_stack(int_stack s);

/* int_stack.c: function definitions for integer stack */
#include <stdio.h>
#include <stdlib.h>
#include "int_stack.h"

void init_int_stack(int_stack *s, int size) {
    s->arr = (int *)malloc(sizeof(int) * size);
    s->size = size;
    s->top = -1;
    return;
}

void push_int(int_stack *s, int data) {
    s->arr[++s->top] = data;
    return;
}

int pop_int(int_stack *s) {
    int element = s->arr[s->top];
    s->top--;
    return element;
}

int peek_int(int_stack s) { return s.arr[s.top]; }

short int is_int_empty(int_stack s) { return s.top == -1; }
```

```
void view_int_stack(int_stack s) {
    printf("[\t");
    for (int i = 0; i <= s.top; i++) {
        printf("%d\t", s.arr[i]);
    }
    printf("]\n");
}

/*
main.c: Contains main flow of the program
main.c has three main parts:
1) Utility Function definitions: These functions are used by main functions
to perform some small task
2) Main Functions: There are 2 main functions I) infix to postfix II)
function to evaluate postfix
3) Actual main function: It contains the flow of the program
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "int_stack.h"
#include "char_stack.h"

                          /* Utility functions */
/* Function to check if give character is whitespace */
short int is_whitespace(char c) {
    char whitespace_characters[] = {' ', '\t', '\n'};
    int len = sizeof(whitespace_characters) / sizeof(char);
    for (int i = 0; i < len; i++) {
        if (c == whitespace_characters[i]) return 1;
    }
    return 0;
}

/* Function to check if given character is operator */
short int is_operator(char c) {
    char operators[] = {'+', '-', '*', '/', '%'};
    int len = sizeof(operators) / sizeof(char);
    for (int i = 0; i < len; i++) {
        if (c == operators[i]) {
            return 1;
        }
    }
    return 0;
}
```

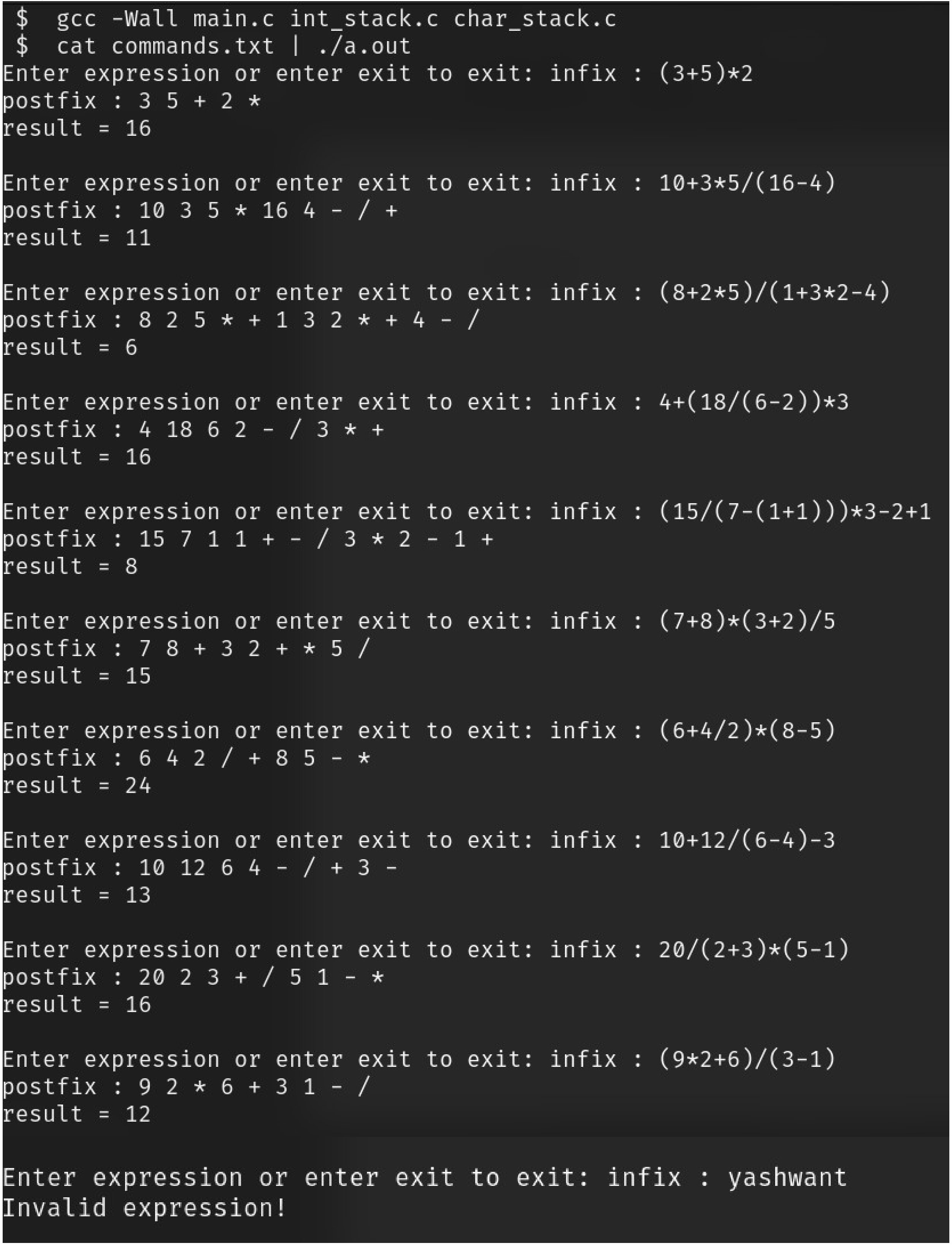
```
/* Function to read expression ignoring white spaces */
void read_expression(char *str) {
    int i = 0;
    while ((str[i] = getchar()) != '\n') {
        if (is_whitespace(str[i])) i--;
        i++;
    }
    str[i] = '\0';
    return;
}

/* Function to check precedance of operator */
short int precedance(char c) {
    if (c == '+' || c == '-')
        return 1;
    else if (c == '*' || c == '/')
        return 2;
    return -1;
}

/* Function to swap two characters in string */
void swap(char *str, int i, int j) {
    char temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    return;
}

/* Function to reverse a string */
void reverse_str(char *str) {
    for (int i = 0; i < strlen(str) / 2; i++) {
        swap(str, i, strlen(str) - i - 1);
    }
    return;
}
```

```c
/* Function to check if the given string has valid circular parentheses */
short int valid_paranthesis(char *str) {
    char_stack s;
    init_char_stack(&s, 64);

    for (int i = 0; i < strlen(str); i++) {
        if (str[i] == '(') {
            push_cs(&s, str[i]);
        } else if (str[i] == ')') {
            if (is_cs_empty(s) || peek_cs(s) != '(') {
                return 0; // Mismatched or missing opening parenthesis
            }
            pop_cs(&s);
        }
    }

    return is_cs_empty(s);
}

/* check if character is legal for the program */
short int check_legal_chars(char c) {
    if(is_operator(c))
        return 1;
    if(c == '(' || c ==')')
        return 1;
    if(c >= '0' && c <= '9')
        return 1;
    return 0;
}

/* Function to validate infix */
short int validate_infix(char *infix) {
    if(!valid_paranthesis(infix)) {
        return 0;
    }
    for(int i = 0;i < strlen(infix);i++) {
        if(!check_legal_chars(infix[i]))
            return 0;

        if(is_operator(infix[i]) && ( is_operator(infix[i-1]) ||
is_operator(infix[i+1]) || infix[i-1] == '(' || infix[i+1] == ')')) {
            return 0;
        }
    }
    return 1;
}
/* Main function 1: Infix to Postfix */
void infix_to_postfix(char *infix_string, char_stack *cs) {
    int i = 0, ptr = 0;
    char postfix[128];

    while (infix_string[i] != '\0') {
        switch (infix_string[i]) {
            case '(': {
                push_cs(cs, infix_string[i++]);
                break;
            }
            case ')': {
                while(peek_cs(*cs) != '('){
                    postfix[ptr++] = pop_cs(cs);
                    postfix[ptr++] = ' ';
                }
                pop_cs(cs);
                i++;
                break;
            }
            default: {
                if (is_operator(infix_string[i])) {
                    while (!is_cs_empty(*cs) && precedance(infix_string[i])
<= precedance(peek_cs(*cs))) {
                        postfix[ptr++] = pop_cs(cs);
                        postfix[ptr++] = ' ';
                    }
                    if(infix_string[i] != ')')
                        push_cs(cs, infix_string[i++]);
                } else{
                    while(infix_string[i] && !is_operator(infix_string[i]))
{

                        postfix[ptr++] = infix_string[i++];
                        if(infix_string[i] == ')')
                            break;
                    }
                    postfix[ptr++] = ' ';
                }
                break;
            }
        }
    }
    while(!is_cs_empty(*cs)){
        postfix[ptr++] = pop_cs(cs);
        postfix[ptr++] = ' ';
    }
    postfix[ptr] = '\0';
    strcpy(infix_string, postfix);
    return;
}
```

```
/* Function to compute result from operator and operands */
int operator_result(char operator, int a, int b) {
    switch (operator){
    case '+':{
        return a+b;
        break;
    }
    case '-': {
        return a-b;
        break;
    }
    case '*': {
        return a*b;
        break;
    }
    case '/': {
        return a/b;
        break;
    }
    case '%': {
        return a%b;
        break;
    }
    default:
        break;
    }
    return INT_MIN;
}

/* Main function 2: Function to evaluate postfix expression */
int evaluate(char *postfix) {
    char *token = NULL;
    int_stack s;
    init_int_stack(&s, 256);
    token = strtok(postfix, " ");
    while(token) {
        if(is_operator(token[0])) {
            int b = pop_int(&s);
            int a = pop_int(&s);
            int result = operator_result(*token, a, b);
            push_int(&s, result);
        } else {
            int num = atoi(token);
            push_int(&s, num);
        }
        token = strtok(NULL, " ");
    }
    return peek_int(s);
}
```

```
int main() {
    char str[128];
    char_stack cs;
    init_char_stack(&cs, 128);
    while(1) {
        printf("Enter expression or enter exit to exit: ");
        read_expression(str);
        if(strcmp(str, "exit") == 0)
            break;
        printf("infix : %s\n", str);
        if(validate_infix(str)) {
            infix_to_postfix(str, &cs);
            printf("postfix : %s\n", str);
            printf("result = %d\n", evaluate(str));
        }else {
            printf("Invalid expression!\n");
        }
        printf("\n");
    }
    return 0;
}

/*
commands.txt: contains list of expressions of various types, purpose is to
make testing easier by piping these directly to the program
 */

(3 + 5) * 2
10 + 3 * 5 / (16 - 4)
(8 + 2 * 5) / (1 + 3 * 2 - 4)
4 + (18 / (6 - 2)) * 3
(15 / (7 - (1 + 1))) * 3 - 2 + 1
(7 + 8) * (3 + 2) / 5
(6 + 4 / 2) * (8 - 5)
10 + 12 / (6 - 4) - 3
20 / (2 + 3) * (5 - 1)
(9 * 2 + 6) / (3 - 1)
yashwant
exit
```

**Output:**



```
 $  gcc -Wall main.c int_stack.c char_stack.c
 $  cat commands.txt | ./a.out
Enter expression or enter exit to exit: infix : (3+5)*2
postfix : 3 5 + 2 *
result = 16

Enter expression or enter exit to exit: infix : 10+3*5/(16-4)
postfix : 10 3 5 * 16 4 - / +
result = 11

Enter expression or enter exit to exit: infix : (8+2*5)/(1+3*2-4)
postfix : 8 2 5 * + 1 3 2 * + 4 - /
result = 6

Enter expression or enter exit to exit: infix : 4+(18/(6-2))*3
postfix : 4 18 6 2 - / 3 * +
result = 16

Enter expression or enter exit to exit: infix : (15/(7-(1+1)))*3-2+1
postfix : 15 7 1 1 + - / 3 * 2 - 1 +
result = 8

Enter expression or enter exit to exit: infix : (7+8)*(3+2)/5
postfix : 7 8 + 3 2 + * 5 /
result = 15

Enter expression or enter exit to exit: infix : (6+4/2)*(8-5)
postfix : 6 4 2 / + 8 5 - *
result = 24

Enter expression or enter exit to exit: infix : 10+12/(6-4)-3
postfix : 10 12 6 4 - / + 3 -
result = 13

Enter expression or enter exit to exit: infix : 20/(2+3)*(5-1)
postfix : 20 2 3 + / 5 1 - *
result = 16

Enter expression or enter exit to exit: infix : (9*2+6)/(3-1)
postfix : 9 2 * 6 + 3 1 - /
result = 12

Enter expression or enter exit to exit: infix : yashwant
Invalid expression!
```