Rashtreeya Sikshana Samithi Trust
R V COLLEGE OF ENGINEERING
(Autonomous Institution Affiliated to VTU, Belagavi)
Department of Computer Science & Engineering
Bengaluru – 560059

DATA STRUCTURES AND ITS APPLICATIONS

SUB CODE: 18IS33

III SEMESTER B.E

LABORATORY RECORD

2020-2021

Name of the student :	
USN :	
Section:	

R V COLLEGE OF ENGINEERING

(Autonomous Institution Affiliated to VTU, Belagavi)

Department Of Computer Science & Engineering

Bengaluru – 560059

Laboratory Certificate

This is to certify that Mr./Ms. _____ has satisfactorily completed the course of experiments in Practical _____ prescribed by the Department of Computer Science and Engineering during the year _____.

Name of the Candidate: _____

USN No: _____ Semester: _____

Marks	
Maximum	**Obtained**

Signature of the Staff in-charge **Head of the Department**

Date:

INDEX

Sl. no.	Program Name.	Date	Record Marks (max 6)	Viva Voice (max 4)	Total Marks
1.	Stack Operations				
2.	Infix to Postfix using Stack				
3.	Messaging system				
4.	Multiplication of polynomials				
5.	Addition of long positive integers				
6.	Sparse Matrix using doubly linked list				
7	Binary Tree operations				
8	Traversal of trees				
9	Hashing using Linear probing				
10	Priority queue operations				
	Total (100)				

Part B

Problem definition (1)	Application of relevant data structure with justification (1)	Incorporation of suggestions (1)	Design (1)	Source Code (2)	Demonstration (2)	Documentation (2)	Total (10)

Part – A

REDUCED RECORD MARKS = (TOTAL/100) *____ = _____

Part A + Part B = _____

LAB INTERNALS		
RECORD	**Max -**	
TEST	**Max -**	
TOTAL	**Max - 50**	
	Signature of the faculty	

PROGRAM 1

Use Stack operations to do the following:

i) Assign to a variable name Y the value of the third element from the top of the stack and keep the stack undisturbed.
ii) Given an arbitrary integer n pop out the top n elements. A message should be displayed if an unusual condition is encountered.
iii) Assign to a variable name Y the value of the third element from the bottom of the stack and keep the stack undisturbed.

(Hint: you may use a temporary stack)

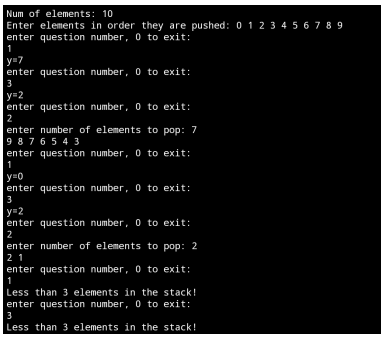
```
#include<stdio.h>
#include<stdlib.h>

#define MAX 20

int error = 0;

#define UNDERFLOW 1
#define OVERFLOW 2
#define BAD_INPUT 3

typedef struct stack {
        int array[MAX];
        int top; // position of element to insert next, is also size of array
} stack;

stack *new_stack() {
        stack* s = (stack *) malloc(sizeof (int) * (MAX+1));
        s->top = 0;
        return s;
}

int is_empty(stack *s) {
        return (s->top == 0);
}

int is_full(stack *s) {
        return (s->top == MAX);
}

int pop(stack *s) {
        if (is_empty(s)) {
                puts ("Stack is empty");
                error = UNDERFLOW;
        } else {
                int r = s->array[s->top - 1];
                s->top -= 1;
                return r;
```

```
        }
}

void push(stack *s, int n) {
        if (!is_full(s)) {
                s->array[s->top] = n;
                s->top += 1;
        } else {
                puts("Stack overflow");
                error = OVERFLOW;
        }
}

void display(stack *s) {
        int i = 0;
        for (i = s->top - 1; i >= 0; i--) {
                printf("%d ", (s->array)[i]);
        }
        printf((s->top == 0) ? "Empty stack\n" : "\n");
}

stack *read_stack() {
        stack *s = new_stack();
        int n, t, i;
        printf("Num of elements: ");
        scanf("%d", &n);
        printf("Enter elements in order they are pushed: ");
        for(i=0;i<n;i++) {
                scanf("%d", &t);
                push(s,t);
        }
        return s;
}

void assign_yt(stack *s) {
        int y;
        int i = 0;
        if (s->top < 3) {
                puts("Less than 3 elements in the stack!");
        } else {
                printf("y=%d\n", s->array[s->top - 3]);
        }
}

void pop_top_n(stack *s, int n) {
        int i = 0;

        if (s->top < n) {
                puts("Stack has less than requested elements");
                error = UNDERFLOW;
        } else {
                for (i = s->top - 1; i > s->top - 1 - n; i--) {
                        printf("%d ", (s->array)[i]);
                }
                s->top = i+1;
```

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

void assign_yb(stack *s) {
        if (s->top < 3) {
                puts("Less than 3 elements in the stack!");
                return;
        } else {
                printf("y=%d\n", s->array[2]);
        }
}

int main() {
        int ch;
        int n;
        stack *s = read_stack();

        while(1) {
                printf("enter question number, 0 to exit: \n");
                scanf("%d", &ch);
                switch (ch) {
                case 0: exit(EXIT_SUCCESS);
                case 1: assign_yt(s);
                        break;
                case 2: printf("enter number of elements to pop: ");
                        scanf("%d", &n);
                        pop_top_n(s, n);
                        break;
                case 3: assign_yb(s);
                        break;
                default: puts("Invalid question numbers");
                }
                error = 0;
        }
        return 0;
}
```

Output:

```
    lab $ gcc -o stack dsa_1.c
lab $ ./stack
Num of elements: 6
Enter elements in order they are pushed: 1 4 8 0 32 76
enter question number, 0 to exit:
1
y=0
enter question number, 0 to exit:
2
enter number of elements to pop: 4
76 32 0 8
enter question number, 0 to exit:
3
Less than 3 elements in the stack!
enter question number, 0 to exit:
0
```



Implementation:

- This implementation uses statical array
    - Advantage: Easy to implement, no dynamic allocation required.
    - Disadvantage: max size needs to be known statically.
- Alternatively can use linked list
    - Advantage: Constant time push and pop, no size limit
    - Disadvantage: allocation of many small nodes →frgamentation and/or GC pressure. Not cache friendly.

- Can use dynamic array
    - Advantage: No size limit, cache friendly, no memory overhead of one ptr per element
    - Disadvantage: O(n) worst case insertion due to resizing.
- More advanced data structures eg: unrolled linked list can be used with varying tradeoffs.

Bugs encountered:

- Scanf in main function used to ask for a character. Due to problems with stdout buffering, didn't work as expected, changed it to int.

- Off by one error in pop_top_n function, writing s->top = i is wrong because i has been decremented once before exiting the loop

Calculations:

- Because top is used to indicate size here, 3rd element from top will be s->array[s->top - 3]
- Likewise, to pop n elements, we have to iterate through elements at top - 1 down to top - n - 1. After iteration, n is decremented once again and exits the loop. We get new value of top by adding 1 to i. (Last popped out element).

LLEGE OF ENGINEERING
(Autonomous Institution Affiliated to VTU, Belagavi)
Department of Computer Science & Engineering
Bengaluru – 560 059

Name: Mahesh Bhaskar Hegde

USN: 1RV19CS082

Date: 29/10/2020

PROGRAM 2

Write a C program that parses Infix arithmetic expressions to Postfix arithmetic expressions using a Stack.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 20

typedef struct op_stack {
    int top;
    char ops[MAX];
} op_stack;

void push(op_stack *s, char op) {
    if(s->top == MAX) return;
    s->top++;
    s->ops[s->top - 1] = op;
}

char pop(op_stack *s) {
    if (s->top == 0) return '?';
    s->top--;
    return s->ops[s->top];
}

char peek(op_stack *s) {
    if (s->top == 0) return '?';
    return s->ops[s->top - 1];
}

int is_op (char op) {
    return (op == '+' || op == '-' || op == '*' ||
        op == '/' || op == '^');
}

int preced (char ch) {
    switch(ch) {
    case '+': return 1;
    case '-': return 1;
    case '*': return 2;
```

```
    case '/': return 2;
    case '^': return 3;
    case '$': return 3;
    default: return 0;
    }
}

// for error handling
// check whether the current position in string comes after an operator

int after_op(char *point, char *begin) {
        while(point != begin) {
            if (is_op(*(point-1))) return 1;
            if (*(point - 1) != ' ') return 0;
            point --;
        }
        return is_op(*point);
}

int main() {
    char input[32];
    op_stack stack;

    char *string;
    stack.top = 0;
    printf("enter infix expression: ");
    scanf("%30[^\n]", input);
    input[30] = (char)0;
    string = input;

    while (*string) {
        char ch = *string;

        if (ch == ' ') {
            string ++;
            continue;
        } else if (ch == '(') {
            push(&stack, ch);
        } else if (ch == ')') {
            while(peek(&stack) != '(') {
                printf(" %c", pop(&stack));
            }
            pop(&stack);
        } else if (is_op(ch)) {
            if(string == input || after_op(string,input) || !string[1]) {
                printf("\nParse error: Invalid Infix expression\n");
                exit(EXIT_FAILURE);
            }

            // if stack is empty or has left bracket, we get lower precedence anyway
            // see implementation of preced()
            while(preced(peek(&stack)) >= preced(ch)) {
                printf(" %c", pop(&stack));
            }
```

```
            // ensure a space occurs between any two operands in output
            putchar(' ');
            push(&stack, ch);
        } else {
            // For simplicity, We ignore spaces while parsing
            // and manually print them between operands
            putchar(ch);
        }

        string ++;
    }

    while(stack.top != 0) {
        printf(" %c", pop(&stack));
    }

    printf("\n");
    return 0;
}
```

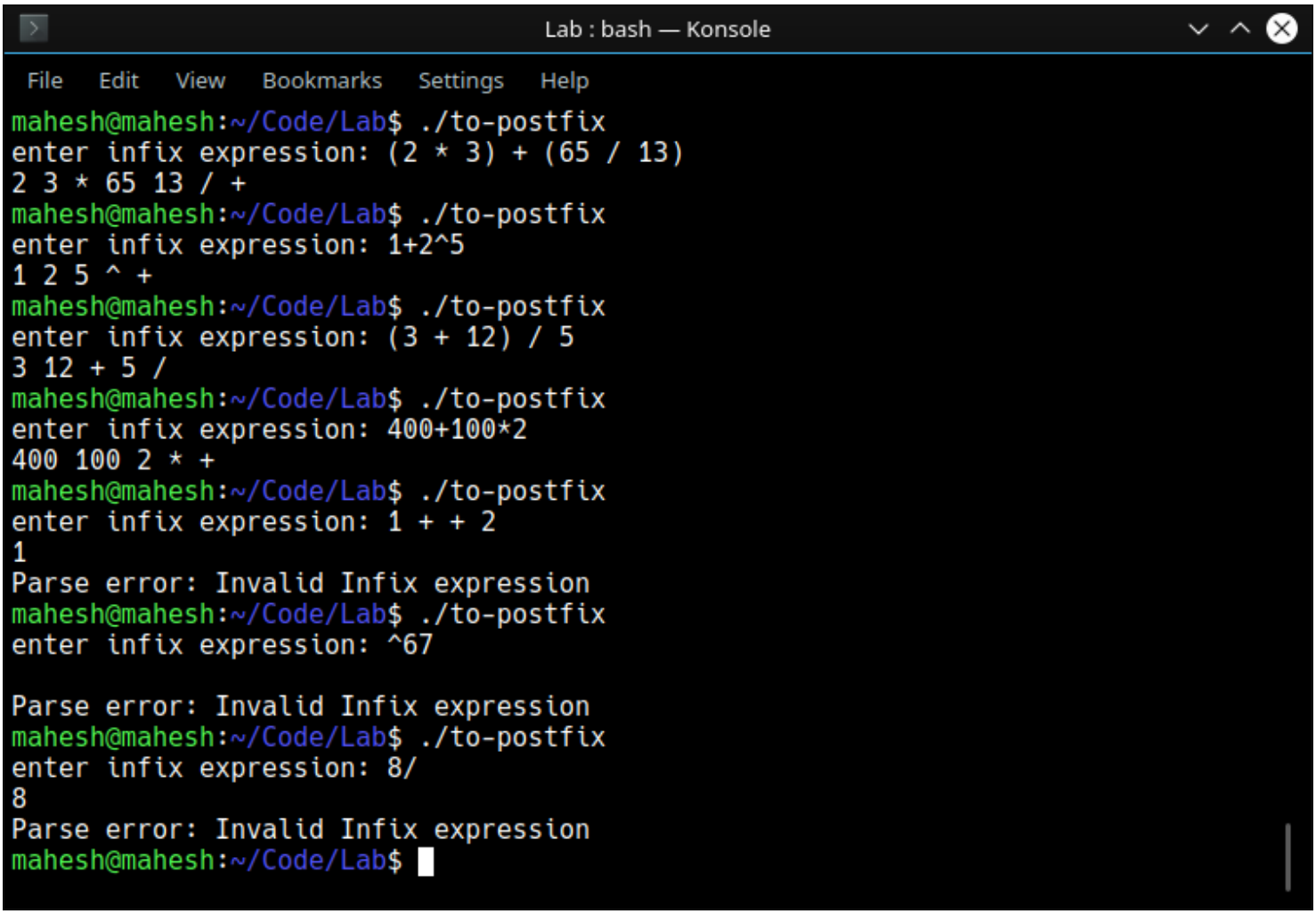
Algorithm:

- While there's character's left in input, read them.
- If character is not an operator, push it to output queue.
- If an operator, as long as stack has higher precedence operators on top, pop those operators from stack and push them to output queue. It will stop when a lower precedence operator / '(' is found or stack is empty. Then push operator to output queue.
- If '(' then push to stack.
- If ')' then pop stack items until we get matching left parenthesis. Discard left parenthesis.
- Once input is read, pop all items remaining in stack and push to output queue.

Implementation Details:

- the program can handle parenthesis
- Spaces in input are also handled
- The program will give error if infix expression is not well formed, i.e an operator is not surrounded by two operands. The error is printed mid-parsing because we are pushing directly to standard output. In C++ we can replace it by std::stringstream and print it at the end.
- The program can be extended to handle unary operators.
- Switch is not used to select the branch, because we need to use is_op function which doesn't correspond to a constant integer expression. The improvement in readability is subjective.

Output Screenshot:


Lab : bash — Konsole
File Edit View Bookmarks Settings Help
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: (2 * 3) + (65 / 13)
2 3 * 65 13 / +
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: 1+2^5
1 2 5 ^ +
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: (3 + 12) / 5
3 12 + 5 /
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: 400+100*2
400 100 2 * +
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: 1 + + 2
1
Parse error: Invalid Infix expression
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: ^67
Parse error: Invalid Infix expression
mahesh@mahesh:~/Code/Lab$ ./to-postfix
enter infix expression: 8/
8
Parse error: Invalid Infix expression
mahesh@mahesh:~/Code/Lab$

PROGRAM 3
Write a C program to simulate the working of Messaging System in which a message is placed in a circular Queue by a Message Sender, a message is removed from the circular queue by a Message Receiver, which can also display the contents of the Queue.

Code:

```
// Implement a circular queue

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

#define MAX 4

enum queue_error {NONE, UNDERFLOW, OVERFLOW};

enum queue_error error = NONE;

typedef struct queue {
        int size, begin;
        char *array[MAX];
} Queue;

int wrap_index(int num, int max) {
        return (num >= max) ? (num - max) : num;
}

void enqueue(Queue *q, char *s) {
        if (q->size == MAX) {
                error = OVERFLOW;
                return;
        }
        q->array[wrap_index(q->begin+q->size, MAX)] = s;
        q->size++;
}

// Returns null if no messages are left

char *dequeue(Queue *q) {
        char *to_ret = NULL;
        if (q->size == 0) {
                error = UNDERFLOW;
                return NULL;
        }

        to_ret = q->array[q->begin];

        q->begin++;
        q->size--;
        if (q->begin == MAX) { q->begin = 0; }
```

```
        return to_ret;
}

void print_queue(Queue *q) {
        int i;
        printf("\e[33;1m[Queue State] ");
        printf("q->size=%d,  ", q->size);
        printf("q->begin=%d, elements: ", q->begin);
        for (i=0;i<q->size;i++) {
                printf("\"%s\" ", q->array[wrap_index(q->begin + i, MAX)]);
        }
        if(q->size == 0) printf("<None>");
        printf("\e[0m\n");
}

void push_msg(Queue *q) {
        error = NONE;
        char *msg = malloc(32*sizeof(char));
        size_t sz = 0;
        printf("Enter message: ");
        scanf(" %30[^\n]", msg);
        enqueue(q, msg);

        if (error == OVERFLOW) {
                printf("\e[31;1mQueue Overflow!!\e[0m\n\n");
        }
}

void print_msg(Queue *q) {
        char *msg = dequeue(q);
        // printf("DEBUG: DEQUEUE done\n");
        // printf("DEBUG: MSG = %p\n", msg);
        if (msg) {
                printf("\e[32;1mMessage: %s\e[0m\n\n", msg);
                free(msg);
        } else {
                printf("\e[32;1mNo new messages\e[0m\n\n");
        }
        fflush(stdout);
}

int main() {
        int choice;
        Queue q;
        q.size = 0; q.begin = 0;
        while(1) {
                print_queue(&q);
                printf("0: exit    ");
                printf("1: Insert a Message    ");
                printf("2: Read a message    ");
                printf("3: Print the queue\n");
                scanf("%u", &choice);
                switch(choice) {
                case 0: return 0;
                case 1: push_msg(&q);
```

```
                break;
        case 2: print_msg(&q);
                break;
        case 3: print_queue(&q);
                break;
        default: printf("Invalid Choice\n");
                return 10;
        }
    }
    return 0;
}
```

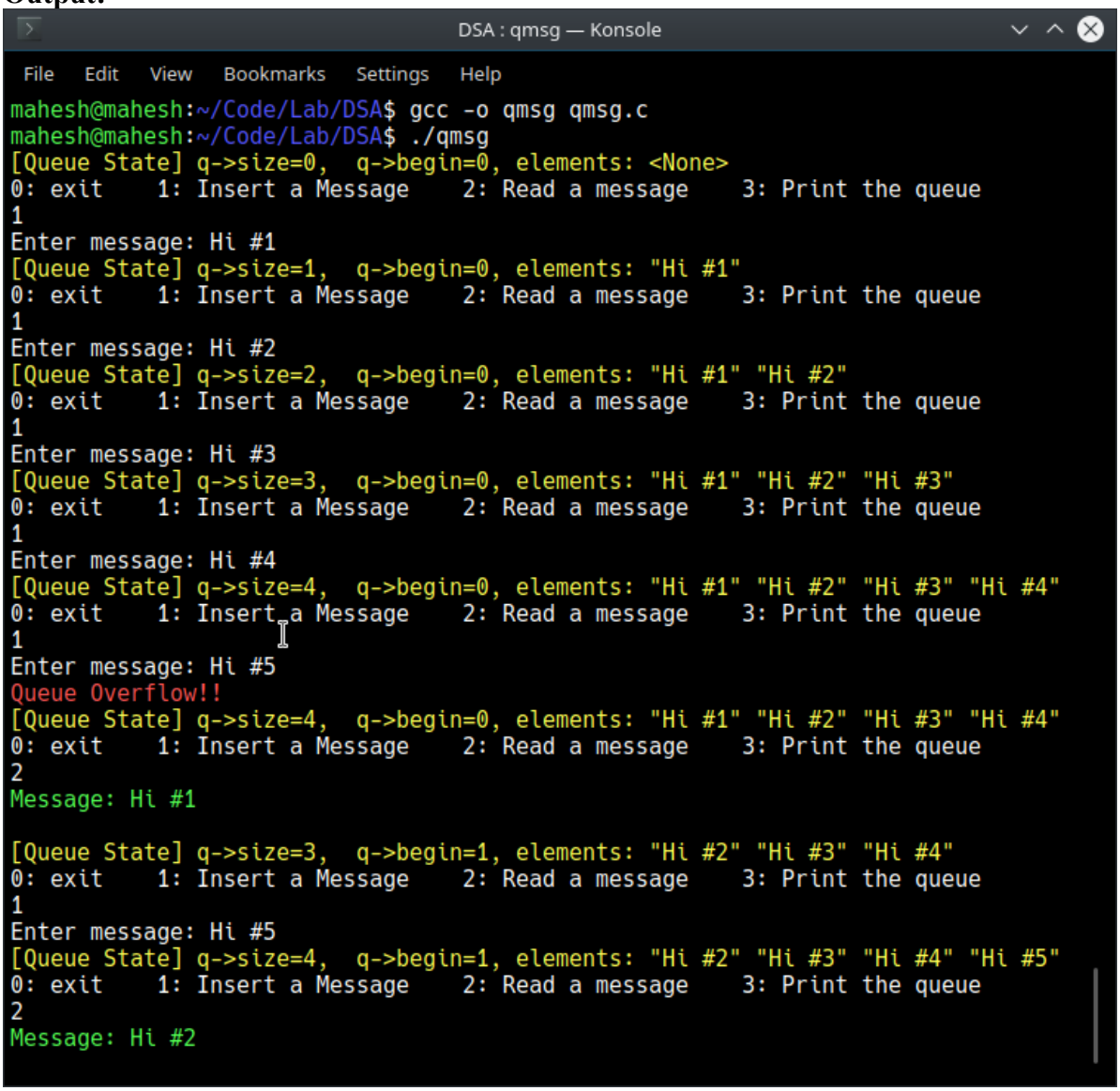
Implementation Details:

- The queue uses modular arithmetic to wrap the calculation for index.
- The wrap_index function conditionally subtracts size from queue length, saving a remainder operation. An invariant is maintained in dequeue operation that begin pointer is always in bounds.
- Coloured output is used to print the queue and visualize its state, using terminal escape codes.
- Dequeue operation can return null on underflow, because user input will always be non-null string.

Inferences:

- If begin < maximum size, then begin + size can be wrapped without % operator, by subtracting maximum size if final index calculation is out of bounds. Size won't exceed maximum size (defined as MAX in this program).

Output:



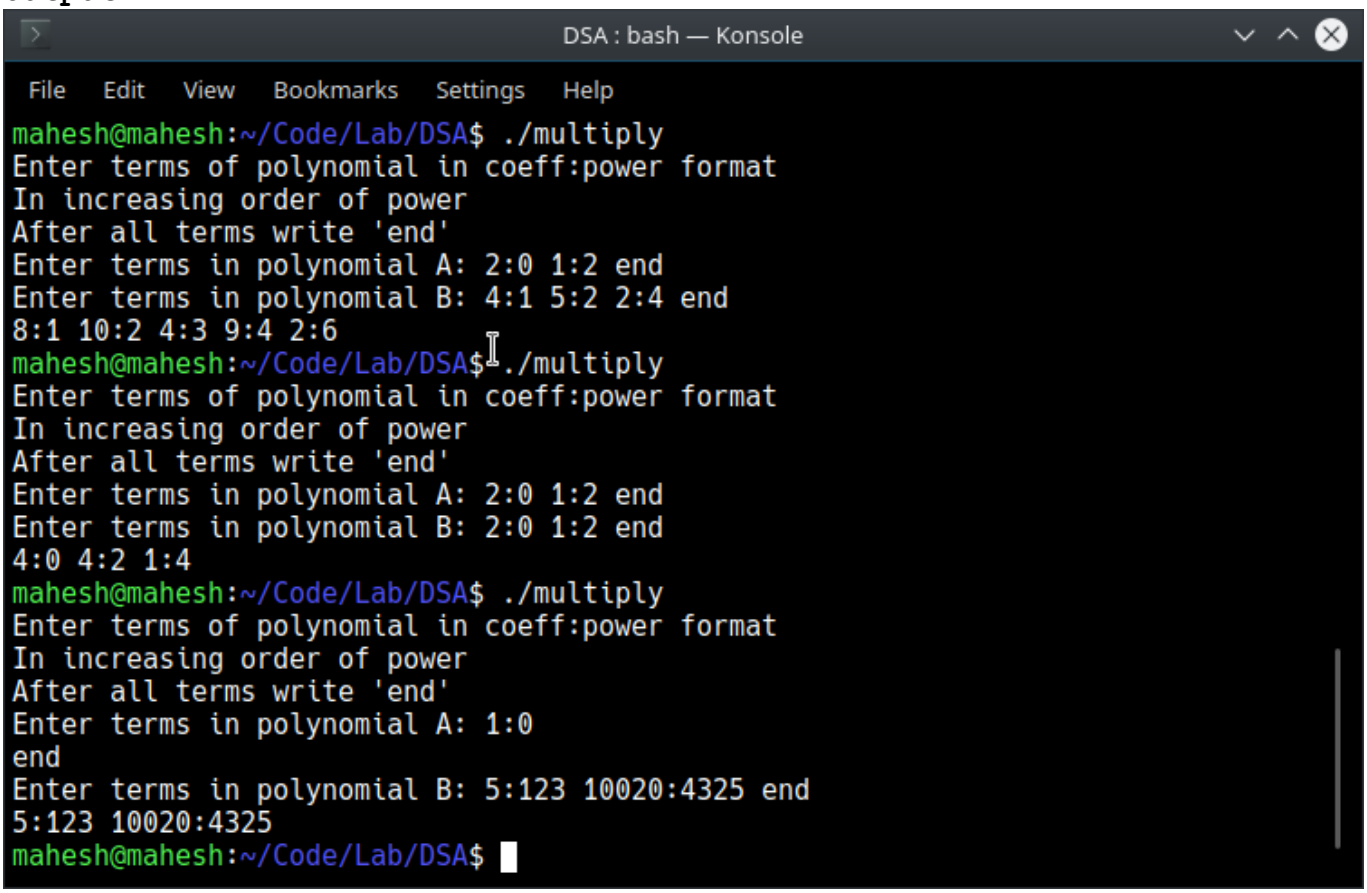
PROGRAM 4
Implement a program to multiply two polynomials using single linked list.

Code:

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

typedef struct node {
     int coeff, power;
     struct node *next;
} Node;

Node *node(int coeff, int power, Node *next) {
     Node *nd = (Node *)malloc(sizeof(node));
     nd->coeff = coeff; nd->power = power; nd->next = next;
     return nd;
}

void print_pol(Node *pol) {
     if (!pol) printf("(null)");
     while(pol) {
          if (pol->coeff != 0) printf("%d:%d ", pol->coeff, pol->power);
          pol = pol->next;
     }
     printf("\n");
}

Node *le_find(Node *list, int power) {
     while(list->next) {
          if (list->next->power >= power) return list;
          list = list->next;
     }
     return list;
}

Node *add_term(Node *to, int coeff, int power) {
     Node *reach = le_find(to, power);
     if (reach->next != NULL && reach->next->power == power) {
          reach->next->coeff += coeff;
     } else {
          reach->next = node(coeff,power,reach->next);
     }

     return reach;
}

Node *multiply(Node *a, Node *b) {
     Node *b_cpy = b;
     Node *res = node(-1,-1,NULL); // dummy node
```

```
     while(a) {
          b = b_cpy;
          Node *r = res;
          while(b) {
               r = add_term(r,
                    b->coeff*a->coeff,
                    b->power+a->power
               );
               b = b->next;
          }
          a = a->next;
     }
     return res->next;
}


int main() {
     Node *a = node(-1,-1,NULL), *b = node(-1,-1,NULL), *result;
     // make a copy of dummy nodes
     Node *a_begin = a, *b_begin = b;
     int coeff, power, last_power;
     printf("Enter terms of polynomial in coeff:power format\n");
     printf("In increasing order of power\n");
     printf("After all terms write \'end\'\n");
     printf("Enter terms in polynomial A: ");
     while(scanf(" %d:%d", &coeff, &power) == 2) {
          if (power <= last_power && a != a_begin) {
               printf("Error: needs increasing order of power\n");
               return 1;
          }
          last_power = power;
          a->next = node(coeff,power,NULL);
          a = a->next;
     }

     scanf("end");

     printf("Enter terms in polynomial B: ");
     while(scanf(" %d:%d", &coeff, &power) == 2) {
          if (power <= last_power && b != b_begin) {
               printf("Error: needs increasing order of power\n");
               return 1;
          }
          last_power = power;
          b->next = node(coeff,power,NULL);
          b = b->next;
     }

     result = multiply(a_begin->next,b_begin->next);
     print_pol(result);
     return 0;
}
```

Output:



Implementation details:

- Enter terms of the polynomial in coefficient:power format, separated by spaces and terminated by "end"
- The linked list implementation only holds the entered powers of the polynomial, thus it can be more efficient than arrays for sparse polynomials.
- We use dummy nodes extensively in this program to simplify list handling, by eliminating special cases with linked lists of one or zero elements.

PROGRAM 5
Implement a program to add 2 long numbers using circular linked list.

Code:

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

typedef struct node {
    int val;
    struct node *next;
} Node;

Node *node(int val, Node *next) {
    Node *nd = (Node *)malloc(sizeof(Node));
    nd->val = val;
    nd->next = next;
    return nd;
}

Node *read_list() {
    int n;
    Node *header = node(-1,NULL);
    Node *cur = header;
    while(isdigit(n = getchar())) {
        cur = node(n - '0', cur);
    }

    header->next = cur;
    return header;
}

int val(Node *nd, Node *h) {
    return (nd == h) ? 0 : nd->val;
}

Node *add(Node *ah, Node *bh) {
     int carry = 0;
     Node *a = ah->next, *b = bh->next;
     Node *sumh = node(-1,NULL), *sum;
     sumh->next = sumh; // self loop
     sum = sumh;
     while( a != ah || b != bh) {

          int dsum = val(a, ah) + val(b, bh) + carry;
          sum = node(dsum % 10, sum);
          carry = dsum / 10;
          if (a != ah) a = a->next;
```

```
            if (b != bh) b = b->next;
        }
        if (carry != 0) sum = node(carry, sum);
        sumh->next = sum;
        return sumh;
}

void print_result(Node *numh) {
    Node *num = numh->next;
    while(num != numh) {
        printf("%d", num->val);
        num = num->next;
    }
    printf("\n");
}

int main () {
    Node *a, *b, *ans;
    printf("Enter an expression: ");
    a = read_list();
    if(scanf("+ ") != 0) {
        fprintf(stderr, "Invalid Input");
        return 1;
    }
    b = read_list();
    ans = add(a,b);
    print_result(ans);
    return 0;
}
```

Implementation:

- We read as long integer in form of an addition expression
- Node type has a constructor that takes all fields, this allows for concise code.
- Header node is used. Using header node simplifies certain operations on circular singly linked list.

Output:

PROGRAM 6

Design a doubly linked list to represent sparse matrix. Each node in the list can have the row and column index of the matrix element and the value of the element. Print the complete matrix as the output.

Code:

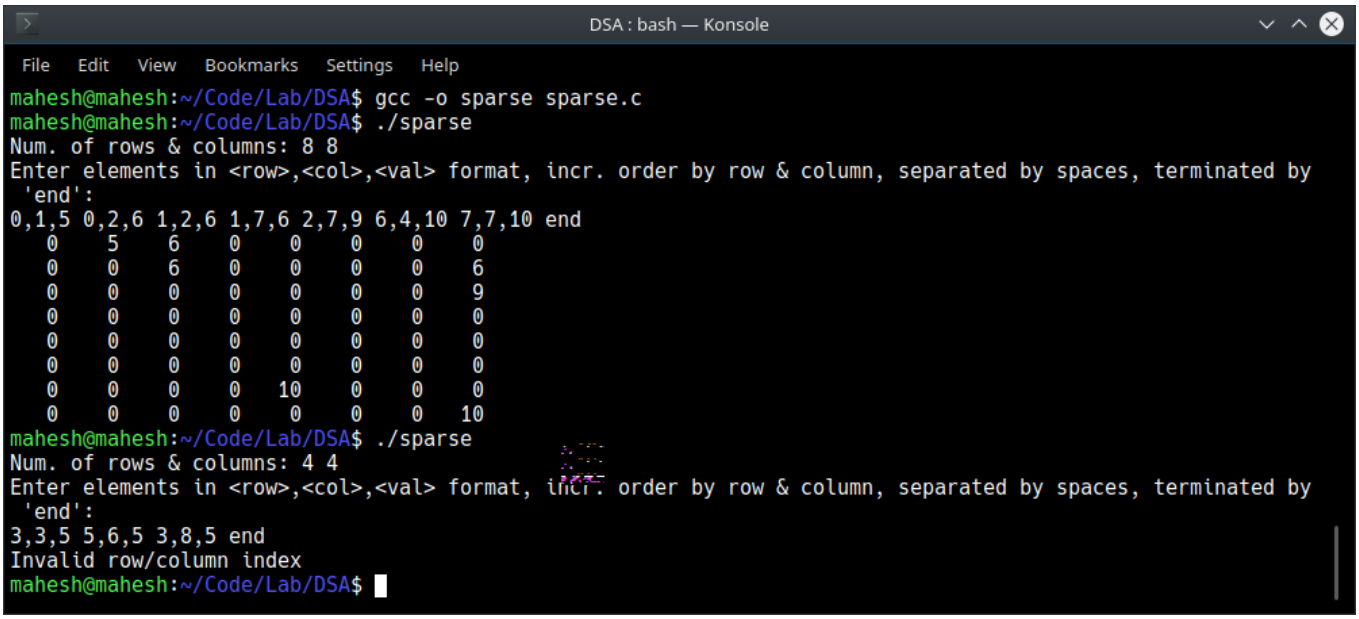
```
#include<stdio.h>
#include<stdlib.h>

typedef struct node {
      int row, col, val;
      struct node *prev, *next;
} Node;

Node *node(int row, int col, int val, Node *prev, Node *next) {
      Node *nd = (Node *)malloc(sizeof(Node));
      nd->row = row;
      nd->col = col;
      nd->val = val;
      nd->prev = prev;
      nd->next = next;
      return nd;
}

void print_sparse(Node *matrix, int m, int n) {
      int i, j;
      for (i=0; i < m; i++) {
            for(j = 0; j < n; j++) {
                  if(matrix != NULL &&
                              matrix->row == i &&
                              matrix->col == j) {
                        printf("%4d ", matrix->val);
                        matrix = matrix->next;
```

```
                } else {
                        printf("%4d ", 0);
                }
            }
            printf("\n");
        }
}

int main() {
        int row, col, val, m, n;
        Node *list = node(-1,-1,-1,NULL,NULL); // DUMMY NODE
        Node *lbegin = list;
        printf("Num. of rows & columns: ");

        if(scanf("%d %d", &m, &n) != 2) {
            printf("Enter valid dimensions\n");
            return 1;
        }

        printf("Enter elements in <row>,<col>,<val> format, ");
        printf("incr. order by row & column, ");
        printf("separated by spaces, terminated by \'end\':\n");
        while(scanf("%d,%d,%d", &row, &col, &val) == 3) {
            if (row >= m || col >= n) {
                printf("Invalid row/column index\n");
                return 1;
            }
            list->next = node(row,col,val,list,list->next);
            list = list->next;
        }
        print_sparse(lbegin->next, m, n);
        return 0;
}
```

Implementation:

- Non zero entries of the sparse matrix are entered in increasing order of rows. For elements in same row, they are inserted in increasing order of column.
- Row and column are zero indexed.

Output:


DSA : bash — Konsole
File Edit View Bookmarks Settings Help
mahesh@mahesh:~/Code/Lab/DSA$ gcc -o sparse sparse.c
mahesh@mahesh:~/Code/Lab/DSA$ ./sparse
Num. of rows & columns: 8 8
Enter elements in <row>,<col>,<val> format, incr. order by row & column, separated by spaces, terminated by
'end':
0,1,5 0,2,6 1,2,6 1,7,6 2,7,9 6,4,10 7,7,10 end
0 5 6 0 0 0 0 0
0 0 6 0 0 0 0 6
0 0 0 0 0 0 0 9
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 10 0 0 0
0 0 0 0 0 0 0 10
mahesh@mahesh:~/Code/Lab/DSA$ ./sparse
Num. of rows & columns: 4 4
Enter elements in <row>,<col>,<val> format, incr. order by row & column, separated by spaces, terminated by
'end':
3,3,5 5,6,5 3,8,5 end
Invalid row/column index
mahesh@mahesh:~/Code/Lab/DSA$

PROGRAM 7

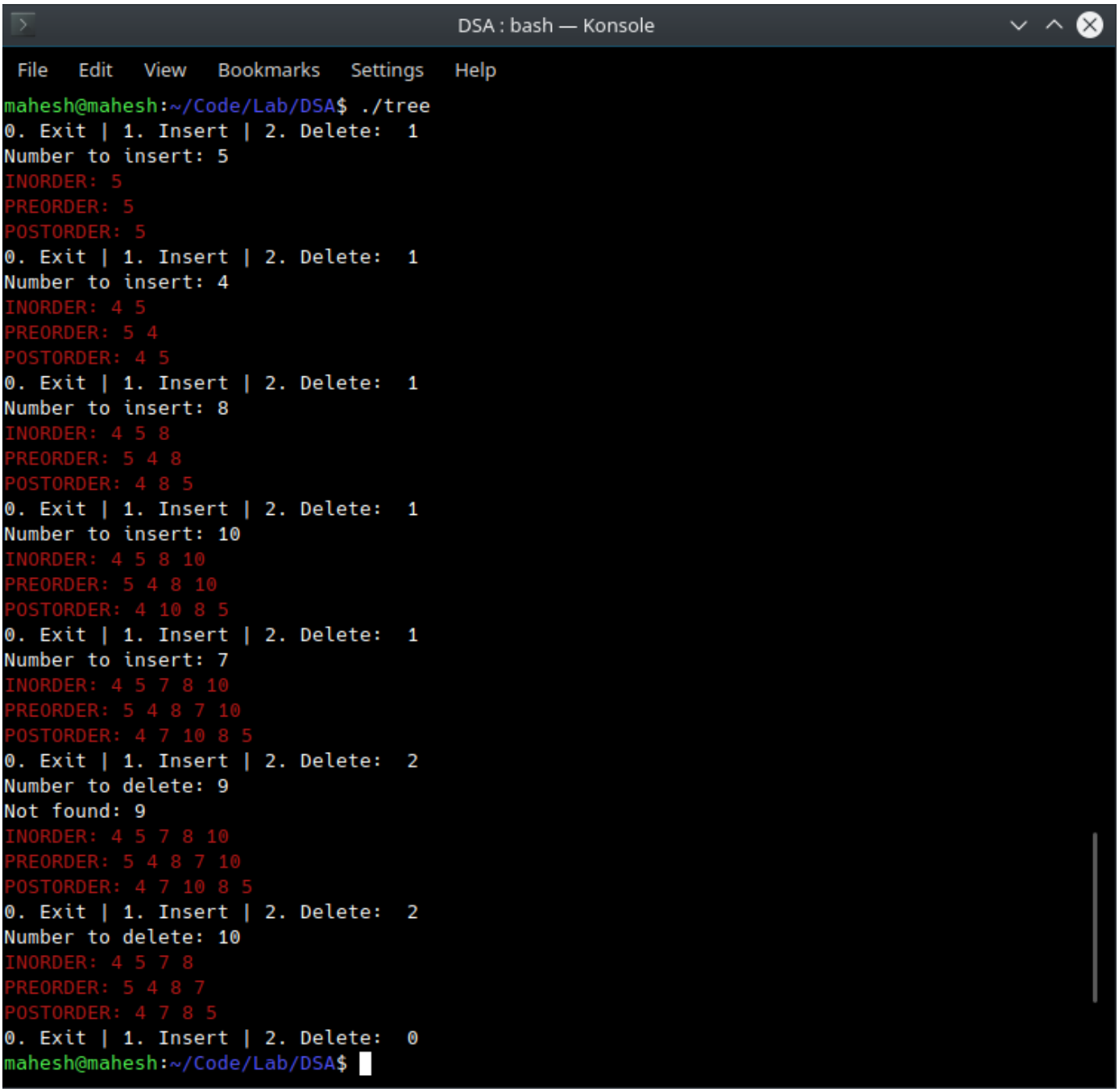
Write a C program to create Binary Tree and provide insertion and deletion operations and to traverse the tree using In-order, Preorder and Post order (recursively)

Code:

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

typedef struct node {
	int val;
	struct node *left, *right;
} Node;

Node *node(int data) {
	Node *nd = malloc(sizeof(Node));
	nd->val = data;
	nd->left = NULL;
	nd->right = NULL;
	return nd;
}

Node *insert(Node *tree, Node *data) {
	if (tree == NULL) return data;
	if (tree->val == data->val) return tree;
	if (tree->val > data->val) {
		tree->left = insert(tree->left, data);
	} else {
		tree->right = insert(tree->right, data);
	}
	return tree;
}

Node *next_largest(Node *tree) {
	Node *r;
	if (!tree) return NULL;
	if ((r = next_largest(tree->left)) == NULL) {
		return tree;
	} else {
		return r;
	}
}

Node *delete(Node *tree, int val) {
	// If a node is deleted
	// the smallest left node is swapped with it
	if (!tree) {
		printf("Not found: %d\n", val);
		return NULL;
	} else if (tree->val == val) {
		if (tree->left == NULL) {
			Node *r = tree->right;
			free(tree);
			return r;
		} else if (tree->right == NULL) {
			Node *r = tree->left;
			free(tree);
			return r;
		} else {
```

```
                Node *nl = next_largest(tree->right);
                tree->val = nl->val;
                free(nl);
                return tree;
            }
        } else if (tree->val > val) {
            tree->left = delete(tree->left, val);
            return tree;
        } else {
            tree->right = delete(tree->right, val);
            return tree;
        }
}

void inorder(Node *tree) {
        if (!tree) return;
        inorder(tree->left);
        printf("%d ", tree->val);
        inorder(tree->right);
}

void preorder(Node *tree) {
        if (!tree) return;
        printf("%d ", tree->val);
        preorder(tree->left);
        preorder(tree->right);
}

void postorder(Node *tree) {
        if (!tree) return;
        postorder(tree->left);
        postorder(tree->right);
        printf("%d ", tree->val);
}

int main() {
        int n, choice;
        Node *tree = NULL;
        while(1) {
            printf("0. Exit | 1. Insert | 2. Delete:  ");
            scanf("%d", &choice);
            switch(choice) {
                case 0: return 0;
                case 1: printf("Number to insert: ");
                        scanf("%d", &n);
                        tree = insert(tree, node(n));
                        break;
                case 2: printf("Number to delete: ");
                        scanf("%d", &n);
                        tree = delete(tree, n);
                        break;
                default: printf("Invalid choice!!\n");
            }
            printf("\e[31mINORDER: ");
            inorder(tree);
            printf("\nPREORDER: ");
            preorder(tree);
            printf("\nPOSTORDER: ");
            postorder(tree);
            printf("\e[0m\n");
        }
        return 0;
}
```

Output:



Observation & implementation details:

- The insertion interface is simplified by choice of a binary search tree insertion scheme. Otherwise we will have to ask user input the exact position which will detract from the main point.
- The functions return the tree itself. They have to be reassigned but this works in general case eg NULL can be used to represent an empty tree with all operations working on it.
- If we have a parent pointer in structure, a non recursive but significantly complex versions of above functions can be written. They still conceptually require O(n) extra space in form of n parent pointers, but will be a good compromise where parent pointers are needed anyway.

PROGRAM 8

Given a String representing a parentheses-free infix arithmetic expression, implement a program to place it in a tree in the infix form. Assume that a variable name is a single letter. Traverse the tree to produce an equivalent postfix and prefix expression string.

Code:

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

#define MAX 20

typedef struct node {
     char sym;
     struct node *left, *right;
} Node;

Node *node(char sym) {
     Node *nd = malloc(sizeof(Node));
     nd->sym = sym;
     nd->left = NULL;
     nd->right = NULL;
     return nd;
}

typedef struct stack {
     Node *data[MAX];
     int top;
} Stack;

void preorder(Node *tree) {
     if (!tree) return;
     printf("%c ", tree->sym);
     preorder(tree->left);
     preorder(tree->right);
}

void postorder(Node *tree) {
     if (!tree) return;
     postorder(tree->left);
     postorder(tree->right);
     printf("%c ", tree->sym);
}

void inorder(Node *tree) {
     if (!tree) return;
     inorder(tree->left);
     printf("%c ", tree->sym);
     inorder(tree->right);
```

```
}

Node *pop(Stack *s) {
     return (s->top == 0) ? NULL : s->data[--s->top];
}

void push(Stack *s, Node *e) {
     if (s->top == MAX) {
          printf("Overflow\n");
     } else {
          s->data[s->top] = e;
          s->top++;
     }
}

int preced(char sym) {
     switch(sym) {
     case '+':
     case '-': return 1;
     case '*':
     case '/': return 2;
     case '^': return 3;
     default: return 0;
     }
}

Node *build_expr(char *s) {
     Node *temp, *t = NULL, *l = NULL, *r = NULL;
     char *sp;
     Stack tree, op;
     tree.top = 0; op.top = 0;
     for(sp = s; *sp; sp++) {
          char ch = *sp;
          temp = node(ch);
          if (isalpha(ch)) {
               push(&tree, temp);
          } else {
               if (op.top == 0) {
                    push(&op,temp);
               } else {
                    while(op.top != 0
                         &&   preced(ch)   <=   preced(op.data[op.top-1]-
>sym)) {
                         t = pop(&op);
                         r = pop(&tree);
                         l = pop(&tree);
                         if (t) t->left = l;
                         if (t) t->right = r;
                         push(&tree, t);
                    }
                    push(&op, temp);
               }
          }
     }
     while(op.top != 0) {
          t = pop(&op);
```

```
            r = pop(&tree);
            l = pop(&tree);
            t->left = l;
            t->right = r;
            push(&tree, t);
      }
      return pop(&tree);
}

int main() {
      Node *root = NULL;
      int ch, ele;
      char str[MAX];
      int i = 0;

      printf("Enter expression: ");
      scanf("%[^\n]", str);
      root = build_expr(str);
      printf("Inorder: ");
      inorder(root);
      printf("\nPostorder: ");
      postorder(root);
      printf("\nPreorder: ");
      preorder(root);
      printf("\n");
      return 0;
}
```

Output:

PROGRAM 9

Write a C program to implement Hashing using Linear probing. Implement insertion, deletion, search and display.

Code:

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

#define MAX 32

const int DELETED = -1;
const int UNUSED = -2;

// A Hash Set of positive integers
typedef struct hash_set {
      int array[MAX];
      int count;
} HashSet;


int hash(int num) {
      return num % MAX;
}

void init_hashset(HashSet *h) {
      int i;
      h->count = 0;
      for (i=0;i<MAX;i++) h->array[i] = UNUSED;
}

int find(HashSet *h, int val) {
      int hs = hash(val), i;
      for(i = 0; i < MAX; i++) {
            int hindex = (hs+i)%MAX;
            if (h->array[hindex] == UNUSED) {
                  return -1;
            } else if (h->array[hindex] == val) {
                  return hindex;
            }
      }
      return -1; // Searched entire array, not found
}


void insert(HashSet *h, int val) {
      int hs = hash(val); // hash is already modulo array size
      int i;
      if (h->count == MAX) {
            printf("HashSet is full!!\n");
            return;
      }

      if (find(h, val) != -1) return;
```

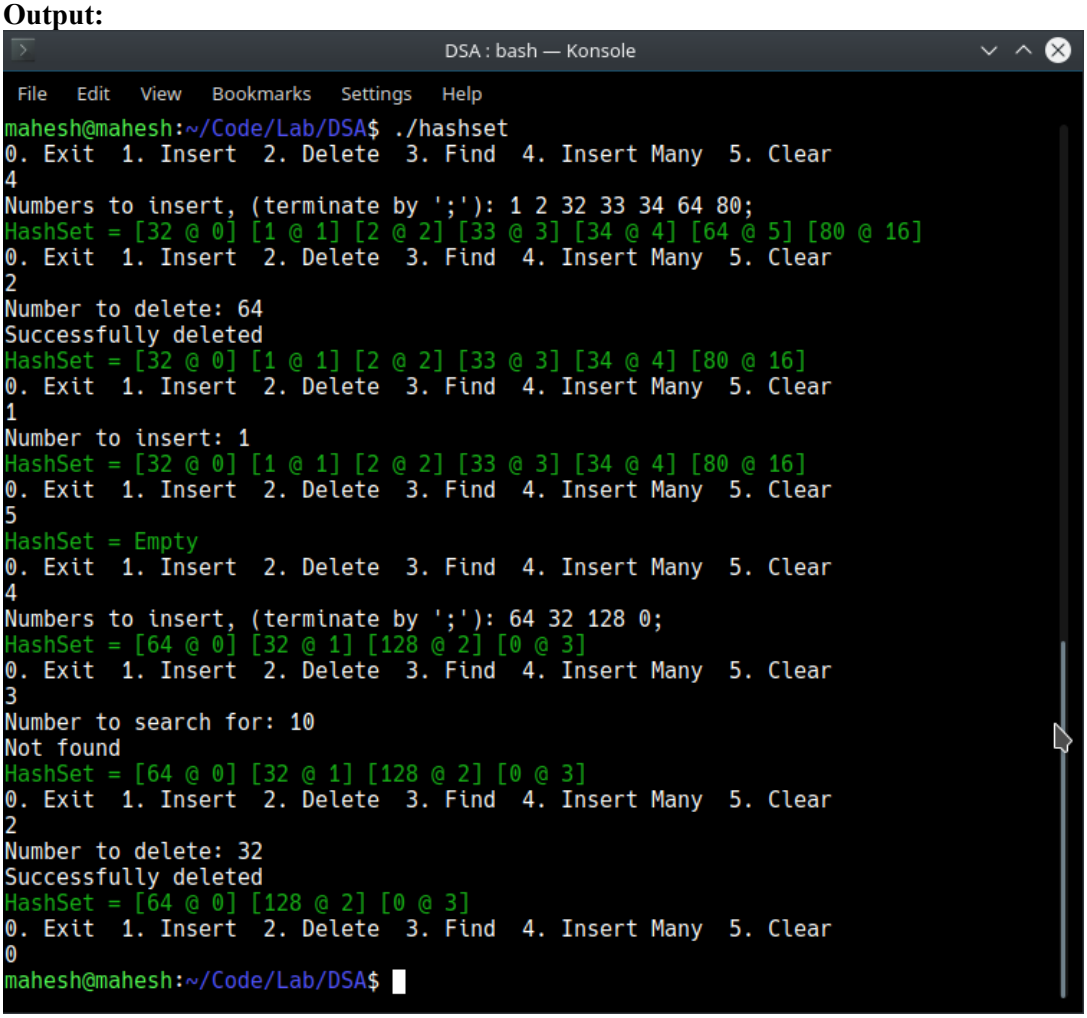
```
	for (i=0;i<MAX;i++) {
		int hindex = (hs+i)%MAX;
		if(h->array[hindex] == UNUSED
				|| h->array[hindex] == DELETED) {
			h->array[hindex] = val;
			h->count++;
			return;
		}
	}
	return;
}

int delete(HashSet *h, int val) {
	int pos = find(h, val);
	if (pos == -1) return 0;
	h->array[pos] = DELETED;
	h->count--;
	return 1;
}

void print_hashset(HashSet *h) {
	int i;
	if (h->count == 0) printf("Empty");
	for(i = 0; i < MAX; i++) {
		if (h->array[i] != UNUSED && h->array[i] != DELETED) {
			printf("[%d @ %d] ", h->array[i], i);
		}
	}
	printf("\n");
}

int main() {
	HashSet h;
	init_hashset(&h);
	while(1) {
		int n, ch, pos;
		printf("0. Exit  1. Insert  2. Delete  3. Find");
		printf("  4. Insert Many  5. Clear\n");
		scanf("%d", &ch);
		switch(ch) {
		case 0: return 0;
		case 1: printf("Number to insert: ");
			scanf("%d", &n);
			insert(&h, n);
			break;
		case 2: printf("Number to delete: ");
			scanf("%d", &n);
			printf(delete(&h,  n)  ?  "Successfully  deleted\n"  :  "Not
Found\n");
			break;
		case 3: printf("Number to search for: ");
			scanf("%d", &n);
			pos = find(&h, n);
			if (pos != -1) {
				printf("Found at bucket %d\n", pos);
			} else {
				printf("Not found\n");
			}
			break;
		case 4: printf("Numbers to insert, (terminate by \';\'): ");
			while(scanf("%d", &n) == 1) {
```

```
                        insert(&h, n);
                    }
                    scanf(";");
                    break;
                case 5: init_hashset(&h);
                    break;
                default: printf("Invalid Choice\n");
                }
                printf("\e[32mHashSet = ");
                print_hashset(&h);
                printf("\e[0m");
        }
}
```

Implementation Notes & Observations:

- To make operations amortized constant time, we need two invalid states, one for unused, another for used and deleted.
- The program prints all valid elements and their positions after each operation. This is useful for visualizing working of Hash based set and also debugging.
- Program also provides interface to insert a lot of numbers, and clear the entire hash set from command line. This is useful for quick testing.
- This particular implementation is a hash based `set`. Most popular application of hashing is dictionary or associative array data structures found in most programming languages, which provide a Map[K,V] interface where K is key type and V is value type. A Map can be theoretically defined as a set of key value pairs in which equality of two elements depends only on key. Thus a dictionary interface can be implemented using similar techniques.

Output:



PROGRAM 10

Write a C program to implement priority queue to insert, delete and display the elements.

Code:

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

#define MAX 32

typedef struct binary_heap {
      int array[MAX];
      int len;
} Heap;

int left(int i) {
      return 2*i + 1;
}

int right(int i) {
      return 2*i + 2;
}

int parent(int i) {
      return i == 0 ? 0 : (i-1)/2;
}

void init_heap(Heap *h) {
      h->len = 0;
}

void bubble_up(Heap *h, int i) {
      int *a = h->array;
      assert(i < MAX);
      int p = parent(i);
      while(i > 0 && a[i] < a[p]) {
            int temp = a[i];
            a[i] = a[p];
            a[p] = temp;
            i = p;
            p = parent(i);
      }
}

void insert(Heap *h, int n) {
      if (h->len == MAX) {
            printf("Overflow!!\n");
      } else {
            h->array[h->len++] = n;
            bubble_up(h, h->len - 1);
      }
}
```

```
void bubble_down(Heap *h, int i) {
      int n = h->len;
      int *a = h->array;
      do {
            int j = -1;
            int r = right(i);
            if (r < n && a[r] < a[i]) {
                  int l = left(i);
                  j = a[l] < a[r] ? l : r;
            } else {
                  int l = left(i);
                  if (l < n && a[l] < a[i]) j = l;
            }
            if (j >= 0) {
                  int temp = a[i];
                  a[i] = a[j];
                  a[j] = temp;
            }
            i = j;
      } while (i >= 0);
}

int delete_min(Heap *h) {
      if (h->len == 0) {
            printf("Underflow!!\n");
            return -1;
      } else {
            int *a = h->array;
            int r = a[0];
            a[0] = a[--h->len];
            bubble_down(h, 0);
            return r;
      }
}

void print_heap(Heap *h) {
      int i;
      printf("\e[35m[");
      if (h->len == 0) printf("Empty Heap");
      for(i = 0; i < h->len; i++) {
            printf("%d ", h->array[i]);
      }
      printf("]\e[0m\n");
}

int main() {
      int n, choice;
      Heap h;
      init_heap(&h);
      while(1) {
            printf("1. Insert   2. Delete Min  3. Clear  ");
            printf("4. Insert Many  0. Exit\n");
            scanf("%d", &choice);
            switch(choice) {
            case 0: return 0;
            case 1: printf("Enter number: ");
                  scanf("%d", &n);
                  insert(&h, n);
                  break;
            case 2: n = delete_min(&h);
                  if (n >= 0) printf("Deleted %d\n", n);
```

```
                break;
            case 3: init_heap(&h);
                break;
            case 4: printf("Enter semicolon terminat. list of nums: ");
                while(scanf("%d", &n) == 1) {
                    insert(&h, n);
                }
                scanf(";");
                break;
            default: printf("Invalid choice!!\n");
            }
            print_heap(&h);
        }
        return 0;
}
```

Implementation Details:

- We print the heap as an array after every operation which helps to visualize the data structure.
- We utilize every element in the array, by slightly adjusting the formula to calculate left/right/parent nodes.
- This program uses separate bubble_up and bubble_down functions to maintain the heap property after an operation, which makes code clearer.

Output: