M S Ramaiah Institute of Technology, Bangalore-54
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

CSL1533: Advanced Programming with C Laboratory

Third Semester UG Programme

Lab Manual

Term: Aug-Dec 2015

Credits: 0:0:1 **Total Credits: 1**

Course Title: **Advanced Programming with C**	***Course Code*: CSL1533**
Credits (L:T:P) : **0:0:1**	***Core/ Elective:*** **Core**
Type of course: **Practical**	***Total Contact Hours*: 28**

Prerequisites: Fundamentals of Computing Lab

Course Contents:

1. Illustrating Pointers for data operations
2. Examining Dynamic memory allocations
3. Composing Arrays in programs
4. Managing Structures in applications
5. Organizing Stacks in programs
6. Constructing Queues for applications
7. Setting up Linked lists for data set operations
8. Formulating Trees for data set maintenance
9. Developing applications to solve Graph based problems

Text Books:

1. Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2008.

Reference Books:

1. Yedidyah, Augenstein, Tannenbaum: Data Structures Using C and C++, 2nd Edition, Pearson Education, 2003.
2. Data Structures, Seynour Lipschutz and GAV Pai, Schaum's Outlines, McGraw Hill, 2008.
3. Richard F. Gilberg and Behrouz A. Forouzan: Data Structures A Pseudocode Approach with C, Cengage Learning, 2005

Course Outcomes:

At the end of the course, students will be able to

1. Illustrate dynamic memory usage in applications
2. Manage arrays and structures with programming solutions for real time problems
3. Prepare programming solutions using variations of stacks and queues
4. Develop data set operations using variations of linked list
5. Devise applications to solve tree based problems
6. Formulate solutions for problems based on graphs

Instructions for the SEE:

Students are required to pick randomly two questions . Students are informed to write the programs first, get it verified by the examiners and execute both the programs later. Viva will be followed after the execution. The break-up of mark is as follow:
Procedure/program writing: 8%, Execution: 35% , Viva-Voce: 7%

Problem Statement: 1

Write a C program to sort numbers using selection sort algorithm.

Illustration:

Selection sort carries out a sequence of passes over the table. At the first pass an entry is selected on some criteria and placed in the correct position in the table. The possible criteria for selecting an element are to pick the smallest or pick the largest. If the smallest is chosen then, for sorting in ascending order, the correct position to put it is at the beginning of the table. Now that the correct entry is in the first place in the table the process is repeated on the remaining entries. Once this has been repeated *n*-1 times the *n*-1 smallest entries are in the first *n*-1 places which leaves the largest element in the last place. Thus only *n*-1 passes are required.

Example:

9 2 5 7 4 8 on pass 1 look for smallest in 1st to 6th swap 2nd with first giving
2 9 5 7 4 8 on pass 2 look for smallest in 2nd to 6th swap 5th with second giving
2 4 5 7 9 8 on pass 3 look for smallest in 3rd to 6th swap 3rd with third giving
2 4 5 7 9 8 on pass 4 look for smallest in 4th to 6th swap 4th with fourth giving
2 4 5 7 9 8 on pass 5 look for smallest in 5th to 6th swap 5th with 6th giving
2 4 5 7 8 9 sorted.

C function for Selection sort:

```
#define swap(x,y,t)  ((t)= (x), (x)= (y), (y)= (t))

        void sort(int list[], int n)
        {
          int i, j, min, temp;
          for (i= 0; i< n-1; i++){
            min= i;
            for (j= i+1; j< n; j++){
              if (list[j]< list[min]) min= j;
            }
            swap(list[i], list[min], temp);
        }
```

Problem Statement: 2

Write a C program to implement recursive binary search.

Illustration:

The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

```
Sorted array: L = [1, 3, 4, 6, 8, 9, 11]
   Target value: X = 4
   Compare X to 6. X is smaller. Repeat with L = [1, 3, 4].
   Compare X to 3. X is larger. Repeat with L = [4].
   Compare X to 4. X equals 4, so the position is returned.
```

C function to implement recursivebinary search:

```
int binsearch(int list[], int searchnum, int left, int right)
{// search list[0]<= list[1]<=...<=list[n-1] for searchnum
int middle;
  if (left<= right){
  middle= (left+ right)/2;
  switch(compare(list[middle], searchnum)){
    case -1:return binsearch(list, searchnum, middle+1, right);
    case 0: return middle;
    case 1: return binsearch(list, searchnum, left, middle- 1);
  }
 }
 return -1;
}
```

Problem Statement: 3

Write a C program to perform following operation on polynomials using an array of structures.

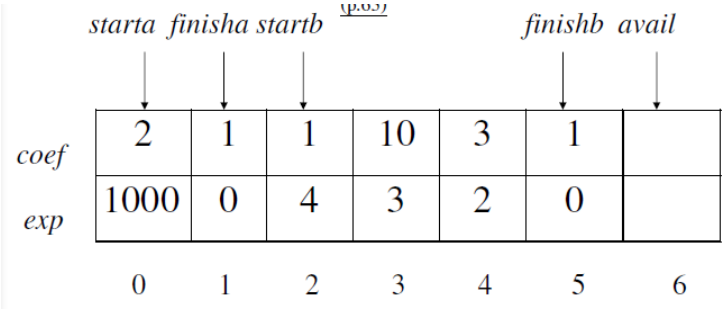
- Addition of two polynomials

Illustration:

Consider the following two polynomials.

$A(X)=2X^{1000}+1$ $B(X)=X^4+10X^3+3X^2+1$

These polynomials can be represented using an array of structures as below.



Structure Declaration:

```
#defineMAX_TERMS 100 /* size of terms array */
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

C function to add two polynomials:

```
void padd (int starta, int finisha, int startb, int finishb,
                                        int * startd, int *finishd)
{
/* add A(x) and B(x) to obtain D(x) */
   float coefficient;
  *startd = avail;
```

```
   while (starta <= finisha && startb <= finishb)
     switch (COMPARE(terms[starta].expon,
                                terms[startb].expon)) {
          case -1: /* a expon < b expon */
              attach(terms[startb].coef, terms[startb].expon);
              startb++;
              break;
          case  0: /* equal exponents */
                       coefficient = terms[starta].coef +
                                        terms[startb].coef;
                       if (coefficient)
                          attach (coefficient, terms[starta].expon);
                       starta++;                    startb++;
                       break;
          case 1: /* a expon > b expon */
                    attach(terms[starta].coef, terms[starta].expon);
                    starta++;
          }
/* add in remaining terms of  A(x) */
for( ; starta <= finisha; starta++)
     attach(terms[starta].coef, terms[starta].expon);
/* add in remaining terms of B(x) */
for( ; startb <= finishb; startb++)
    attach(terms[startb].coef, terms[startb].expon);
*finishd =avail -1;
}
void attach(float coefficient, int exponent)
{
/* add a new term to the polynomial */
    if (avail >= MAX_TERMS) {
       fprintf(stderr, "Too many terms in the polynomial\n");          exit(1);
      }
      terms[avail].coef  = coefficient;
      terms[avail++].expon = exponent;
}
```

Problem Statement: 4

Write a C program to perform following operations on Sparse Matrices

- Simple Transpose
- Fast Transpose

Illustration:

A sparse matrix is a matrix in which most of the elements are zero.

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

- Represented by a two-dimensional array. Sparse matrix wastes space.
- Each element is characterized by <row, col, value>.

Example of Sparse matrix and its transpose:

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Sturcture Declaration:

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
   typedef struct {
            int col;          int row;          int value;
            } term;
   term a[MAX_TERMS];
```

C function for Simple Transpose:

```
void transpose (term a[], term b[])
/* b is set to the transpose of a */
{
   int n, i, j, currentb;
   n = a[0].value;  /* total number of elements */
   b[0].row = a[0].col;  /* rows in b = columns in a */
   b[0].col = a[0].row;  /*columns in b = rows in a */
   b[0].value = n;
   if (n > 0) {                /*non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by columns in a */
           for( j = 1; j <=  n; j++)
           /*  find elements from the current column */
           if (a[j].col == i) {
          /* element is in current column, add it to b */
             b[currentb].row = a[j].col;
             b[currentb].col  = a[j].row;
             b[currentb].value = a[j].value;
             currentb++
           }
  }
}
```

C function for Fast Transpose:

```
void fast_transpose(term a[ ], term b[ ])
 {
 /* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i, j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols; b[0].col = a[0].row;
   b[0].value = num_terms;
```

```
    if (num_terms > 0){ /*nonzero matrix*/
      for (i = 0; i < num_cols; i++)
           row_terms[i] = 0;
      for (i = 1; i  <= num_terms; i++)
           row_term [a[i].col]++
      starting_pos[0] = 1;
      for (i =1; i < num_cols; i++)
           starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
      for (i=1; i <= num_terms, i++) {
                j = starting_pos[a[i].col]++;
                b[j].row = a[i].col;
                b[j].col = a[i].row;
                b[j].value = a[i].value;
        }
    }
}
```

Problem Statement: 5

Write a C program to perform pattern matching.

- Matching End Indices First(nfind)
- KMP Algorithm

Illustration:

Matching End indices first:

KMP Algorithm:

Example> pat = 'abcabcac'

s = '…ab????????' => s = '….ab????????'

pat= 'abcadcac' => pat = 'abcabcac'
s = '…abca??????' => s = '…abca??????'

pat = 'abcabcac' => pat = 'abcabcacab'

Failure function example:

j	0	1	2	3	4	5	6	7
pattern	a	b	c	a	b	c	a	c
	-1	−1	−1					
f(j)	0				1	2	3	-1

C function for pattern matching by end induces first:

```
int nfind (char *string, char *pat)
{
    int i, j, start = 0
    int lasts = strlen(string)-1, lastp = strlen(pat)-1; int endmatch
    = lastp;
    for (i = 0; endmatch<=lasts; endmatch++, start++) {
    if (string[endmatch] == pat[lastp]){
                for(;j<lastp && string[i]==pat[j]; i++, j++;)
                 if (j == lastp)
                            return start;
```

```
        }
}
return -1;
}
```

C function for KMP algorithm:

```
int pmatch (char *string, char *pat)
{
   int i-0,j=0;

     lens=strlen(string), lenp=strlen(pat);

                    while (i < lens && j < lenp){
           if (string[i] == pat[j]) { i++; j++; } else if (j == 0) i++;
           else j = failure[j-1]+1;
     }
     return (j == lenp ? i-lenp : -1);
}
void fail(char *pat)

{

int i,j;

failure[0]=-1;
int n=strlen(pat);

for(j=1;j<n;j++)

{

i=failure[j-1];

while(pat[j]!=pat[i+1]&&i>=0)

i=failure[i];

if(pat[j]==pat[i+1])

failure[j]=i+1;

else

failure[j]=-1;

}

}
```

Problem Statement : 6

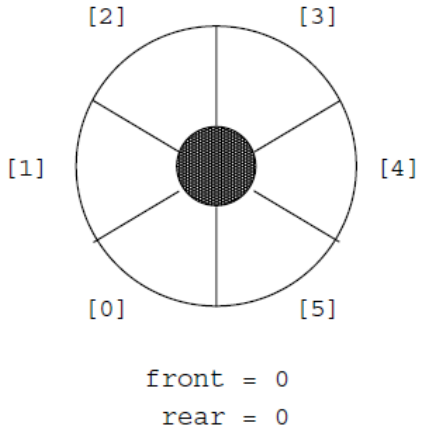
Write a C program to perform the following operations for a circular queue implemented using dynamically allocated arrays.

1. Insert an item
2. Delete an item
3. Display a circular queue

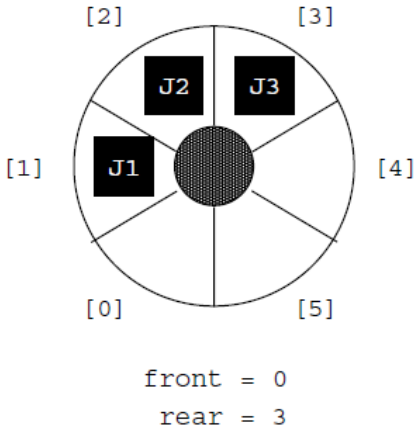
Illustration:

For circular queue representation:

Initially *front* and *rear* to 0. The *front* index always points one position counterclockwise from the first element in the queue and the *rear* index points to the current end of the queue.



Empty Circular Queue



Non empty circular queue

Structure Declaration:

```
#define MAX_QUEUE_SIZE 100
typedef struct {
int key;
/* other fields */
} element;
element queue[MAX_QUEUE_SIZE];
```

C function to add an item in circular queue:

```
void addq(element item)
{
        rear = (rear + 1) % MAX_QUEUE_SIZE;
        if (front == rear) {
        queue_full(rear);
        /* reset rear and print error */
        return;
        }
        queue[rear] = item;
}
```

C function to delete an item from circular queue:

```
element deleteq()
{
        element item;
        if (front == rear)
        return queue_empty();
        /* queue_empty returns an
        error key */
        front = (front + 1) % MAX_QUEUE_SIZE;
        return queue[front];
}
```

```
void queueFull()
{
element* newQueue;
MALLOC(newQueue, capacity*2, element);
int start=(front+1)%capacity;
if(start < 2) //either 1 or 0, 1 when front at 0, 0 when front at capacity - 1
copy(queue+start, queue+start+capacity-1, newQueue);
```

```
else
{
copy(queue+start, queue+capacity , newQueue);
copy(queue, queue+rear+1, newQueue+capacity-start);
}
front=2*capacity-1;
rear=capacity-2;
capacity*=2;
free(queue);
queue=newQueue;
}

void copy(element* start, element* end, element* newQueue)
{
element* j;
element* i;
i=newQueue;
j=start;
for(; j<end; j++, i++)
{
*i=*j;
}
}
```

Problem Statement: 7

Write a C program to implement following stack application:

- Evaluation of Postfix Expression

Illustration:

Steps for evaluation of postfix expression :

- Scan left-to-right

- Place the operands on a stack until an operator is found

- Perform operations and push back result

Given postfix expression: 62/3-42*+

Token	Stack[0]	Stack[1]	Stack[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Structure Declaration:

```
#define MAX_STACK_SIZE 100 /* max stack size */
#define MAX_EXPR_SIZE 100
/* max expression size */
typedef enum {lparen, rparen, plus, minus,
times, divide, mode, eos, operand
} precedence;
int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string */
```

C function to evaluate postfix expression:

```
int eval()
{
        precedence token;
        char symbol;
```

```
        int op1, op2;
        int n = 0;   int top = -1;
        token = get_token(&symbol, &n);
        while (token != eos)
        if (token == operand)
        push(&top, symbol-'0');
        else {
        op2 = pop(&top);     op1 = pop(&top);
        switch (token) {
        case plus: push(&top, op1+op2); break;
        case minus: push(&top, op1-op2); break;
        case times: push(&top, op1*op2); break;
        case divide: push(&top, op1/op2); break;
        case mod: push(&top, op1%op2);
        }
        }
        token = get_token(&symbol, &n);
        }
        return pop(&top);
}
precedence get_token(char *psymbol, int *pn)
{
        *psymbol = expr[(*pn)++];
        switch (*psymbol)
        case `(` : return lparen;
        case `)` : return rparen;
        case `+` : return plus;
        case `-` : return minus;
        case `*` : return times;
        case `/` : return divide;
        case `%` : return mod;
        case ` ` : return eos;
        default : return operand; /* no error checking */
        }
        }
```

Problem Statement: 8

Write a C program to implement following stack application:

- Infix to Postfix Conversion

Illustration:
An infix expression contains an operator between its operands.

Examples of infix to postfix:

Infix	Postfix
2+3*4	2 3 4*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
((a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

Translation of a+b*c to postfix:

token	stack [0]	[1]	[2]	top	output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

C function for to convert an infix to postfix expression:

```
void postfix(void)
{
        char symbol;
        precedence token;
        int n = 0;
        int top = 0;
        stack[0] = eos;
        for (token = get_token(&symbol, &n); token != eos; token =
        get_token(&symbol, &n)) {
        if (token == operand)
```

```
        printf("%c", symbol);
        else if (token == rparen) {
                while (stack[top] != lparen)
        print_token(pop(&top));
        pop(&top);
        }
        else {
        while (isp[stack[top]] >= icp[token])
        print_token(pop(&top));
        push(&top, token);
        }
        }
        while ((token = pop(&top)) != eos)
        print_token(token);
        printf("\n");
}
```

Problem Statement: 9

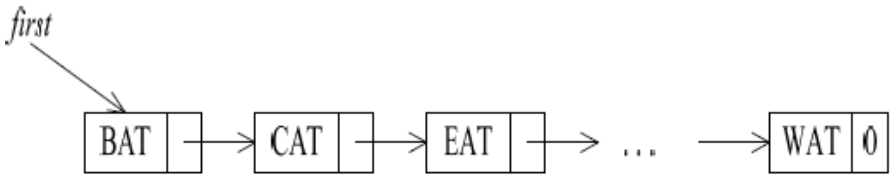
Write a c program to implement following operation on singly linked lists.

- Insert a node
- Delete a node
- Print the linked list

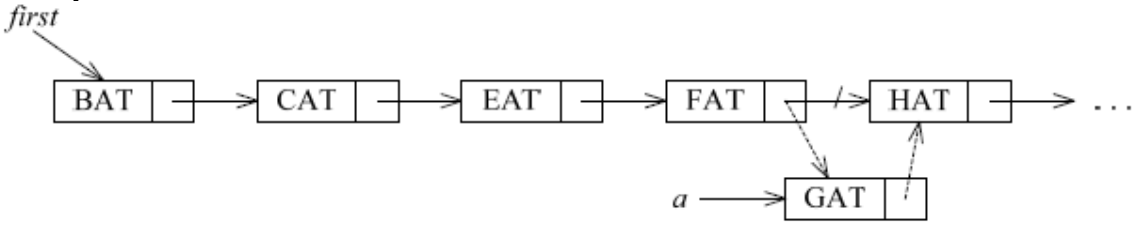
Illustration:

A singly linked list consists of number of nodes where each node has da field and link field. Pointer first points to the first node while last nodes link field contains a NULL.

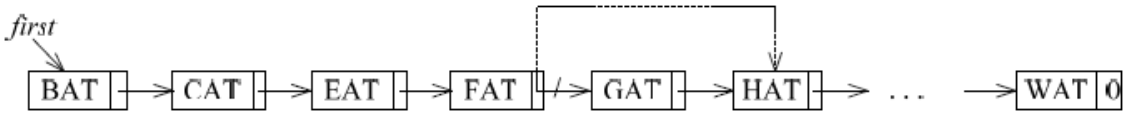
Examaple of a linked list:



Example: To insert a node



Example: To delete a node



Structure Declaration:

```
typedef struct list_node *listPointer;
typedef struct listNode {
 int data;
listPointer link;
};
```

C function to insert a node:

```
void insert(listPointer *first, listPointer x)
{ /* insert a new node with data = 50 into the chain first after node x */
        listPointer temp;
        temp = (listPointer) malloc(sizeof(listNode));
        temp->data = 50;
        if (*first) {//noempty list
        temp->link = x ->link;
        x->link = temp; }
        else
        { //empty list
        temp->link = NULL;
        *first =temp; }
 }
```

C function to delete a node:

```
void delete(listPointer *first, listPointer trail, listPointer x)
 {/* delete x from the list, trail is the preceding node ptr is the head of the list */
        if (trail)
         trail->link = x->link;
        else
         *first = (*first) ->link;
        free(x);
 }
```

C function to print the linked list:

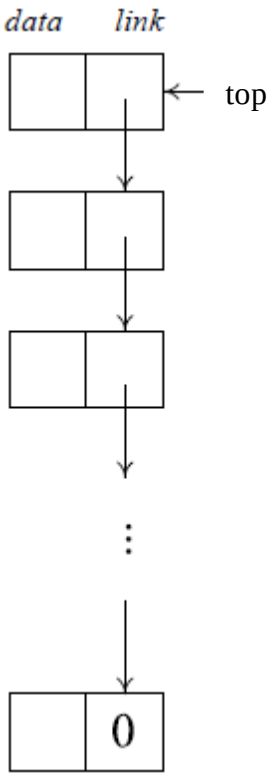
```
void print_list(listPointer first)
{
         printf("The list contains: ");
        for ( ; first; first = first->link)
        printf("%4d", first->data);
        printf("\n");
}
```

Problem Statement: 10

Write a C program to implement linked stack.

Illustration:
In a linked stack we can easily add or delete a node from the top of the stack.

Representaion of Linked Stack:



Structure Defintion:

```
#define MAX_STACKS 10 /* maximum number of stacks */ typedef
struct {
 int key; /* other fields */
} element;
typedef struct stack *stackPointer;
typedef struct stack
 { element data;
stackPointer link; };
stackPointer top[MAX_STACKS];
```

C function to push an item in linked stack:

```
void add(stackPointer *top, element item)
{ /* add an element to the top of the stack */
        stackPointer temp = (stackPointer) malloc (sizeof (stack));
        if (IS_FULL(temp))
        { fprintf(stderr, " The memory is full\n"); exit(1); }
        temp->data = item;
        temp->link = *top;
        *top= temp;
}
```

C function to pop an item from linked stack:

```
element delete(stackPointer *top)
{ /* delete an element from the stack */
        stackPointer temp = *top;
        element item;
        if (IS_EMPTY(temp)) { fprintf(stderr, "The stack is empty\n"); exit(1); }
        item = temp->data;
        *top = temp->link;
        free(temp);
        return data;
}
```

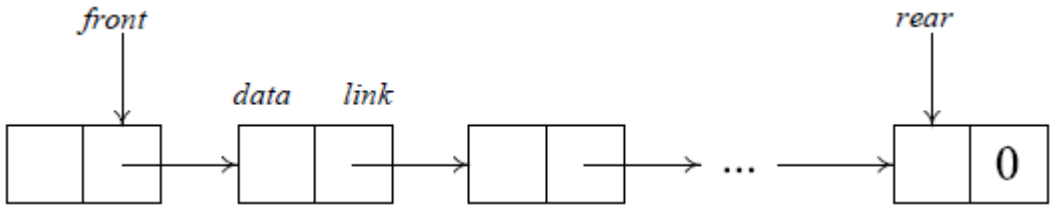
Problem Statement: 11

Write a C program to implement linked queue.

Illustration:

In a linked queue we can easily add a node to the rear of a queue and delete a node from the front of the queue.

Representaion of linked queue:



Structure Declaration:

```
#define MAX_QUEUES 10 /* maximum number of queues */
 typedef struct queue *queuePointer;
typedef struct queue
 { element data;
 queuePointer link;
};
 queuePointer front[MAX_QUEUE], rear[MAX_QUEUES];
```

C function to add an item in linked queue:

```
void addq(element item)
{ /* add an element to the rear of the queue */
queuePointer temp = (queuePointer)
malloc(sizeof(queue));
        if (IS_FULL(temp)) {
        fprintf(stderr, " The memory is full\n"); exit(1);
        }
        temp->data = item;
        temp->link = NULL;
        if(*front)
        (*rear)->link=temp;
        else
```

```
        *front=temp;
        *rear = temp;
}
```

C function to delete an item from linked queue:

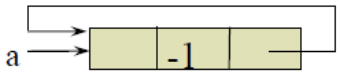
```
element deleteq()
 { /* delete an element from the queue */
        queuePointer temp = *front; element item;
        if (IS_EMPTY(*front)) { fprintf(stderr, "The queue is empty\n"); exit(1); }
        item = temp->data;
          *front = temp->link;
        free(temp);
        return item;   }
```

Problem Statement: 12

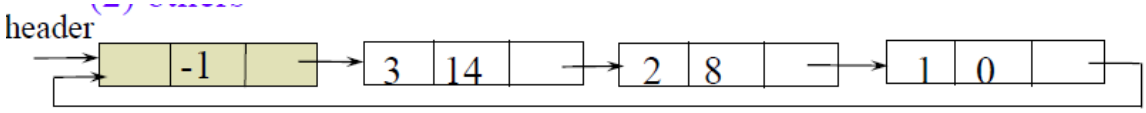
Write a C program to add two polynomials represented as circular linked lists.

Illustration:

Polynomials represented using linked lists with header nodes can be effieciently manipulated.



Zero polynomial



$$a = 3x^{14} + 2x^{8} + 1$$

Structure Declaration:

```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
 int coef;
int expon;
polyPointer link; };
polyPointer a, b, c;
```

C function to add two polynomials:

```
poly_pointer cpadd(polyPointer a, polyPointer b) polyPointer int

{
        starta, d, lastd;
        sum, done = FALSE;
        starta = a;
        a = a->link;
        b = b->link;
        d = get_node();
```

```
d-> expon = -1; lastd = d;
do {
    switch (COMPARE(a->expon, b->expon))
    {
    case -1: attach(b->coef, b->expon, &lastd);
               b = b-> link; break;
    case 0: if (starta == a) done = TRUE;
            else
              { sum = a-> coef + b-> coef;
                 if (sum) attach(sum,a->expon,&lastd);
                    a = a->link; b = b->link;
    case 1: attach(a->coef,a->expon,&lastd); } break;
             a = a-> link; }
} while (!done);
lastd->link = d;
return d;
}
```

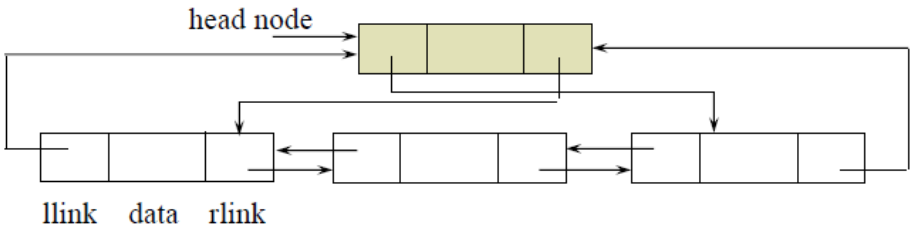
Problem Statement: 13

Write a C program to perform following operations on doubly linked list.

- Insert a node
- Delete a node

Illustration:

In a doubly linked list, we can traverse in both the directions from any node.



Structure Declaration:

typedef struct node *nodePointer;

typedef struct node {

nodePointer llink;

element data;

nodePointer rlink;

}

C function io insert a node in doubly linked list:

```
void dinsert(nodePointer node, nodePointer newnode)
{
   newnode-> llink = node;
   newnode-> rlink = node-> rlink;
   node-> rlink-> llink = newnode;
   node-> rlink = newnode;
}
```

C function io delete a node from doubly linked list:

```
void ddelete(nodePointer node, nodePointer deleted)
{
        if (node==deleted)
                printf("Deletion of head nodenot permitted.\n");
        else {

                deleted->llink->rlink= deleted->rlink;
                deleted->rlink->llink= deleted->llink;
                free(deleted);
             }
}
```

Problem Statement: 14

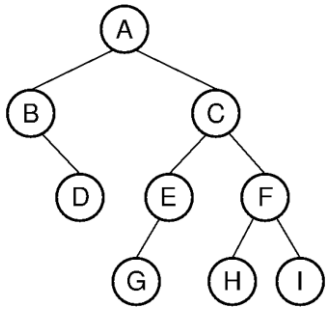
Write a C program to perform following traversals on binary tree:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Illustration:

Traversing a tree involves iterating over all nodes in some manner. Because from a given node there is more than one possible next node.

For the given bin ary tree



Inorder: B D A G E C H F I
Preorder : A B D C E G F H I
Postorder: D B G E H I F C A

Structure Declaration:

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```

C function for inorder traversal:

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d", ptr->data);
      inorder(ptr->right_child);
   }
}
```

C function for preorder traversal:

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
   if (ptr) {
      printf("%d", ptr->data);
      preorder(ptr->left_child);
      preorder(ptr->right_child);
   }
}
```

C function for postorder traversal:

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
   if (ptr) {
      postorder(ptr->left_child);
      postorder(ptr->right_child);
      printf("%d", ptr->data);
   }
}
```

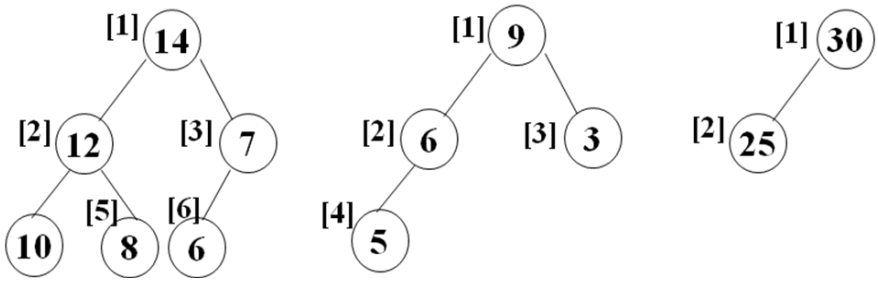
Problem Statement: 15

Write a C program to implement heap operations.

Illustration:

A max tree is a tree in which the key value in each node is no smaller than the key values in its children. A max heap is a complete binary tree that is also a max tree.

Examples of Max heap:



C function insert an element in max heap:

```
void insert_max_heap(element item, int *n)
{
  int i;
  if (HEAP_FULL(*n)) {
    fprintf(stderr, "the heap is full.\n");
    exit(1);
  }
  i = ++(*n);
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

C function to delete an element from max heap:

```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
   highest key */
  item = heap[1];
  /* use last element in heap to adjust heap */
  temp = heap[(*n)--];
  parent = 1;
  child = 2;
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n)&&
        (heap[child].key<heap[child+1].key))
      child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

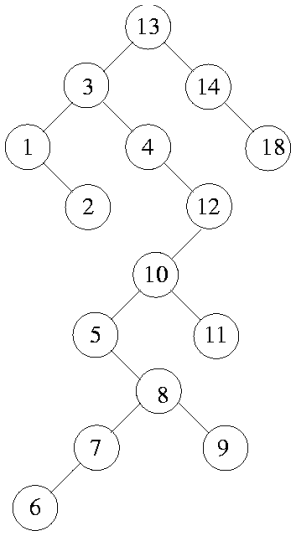
Problem Statement: 16

Write a C program to perform following operation on binary serach tree.

- Insert a node
- Search a node

Illustration:

In Binary search tree, every element has a unique key. The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree. The left and right subtrees are also binary search trees.



Structure Declaration:

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```

C function to insert a node in binary search tree:

```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
      temp = modified_search(*node, num);
  if (temp || !(*node)) {
   ptr = (tree_pointer) malloc(sizeof(node));
   if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   if (*node)
     if (num<temp->data) temp->left_child=ptr;
        else temp->right_child = ptr;
   else *node = ptr;
  }
}
```

C function to search a node in binary search tree:

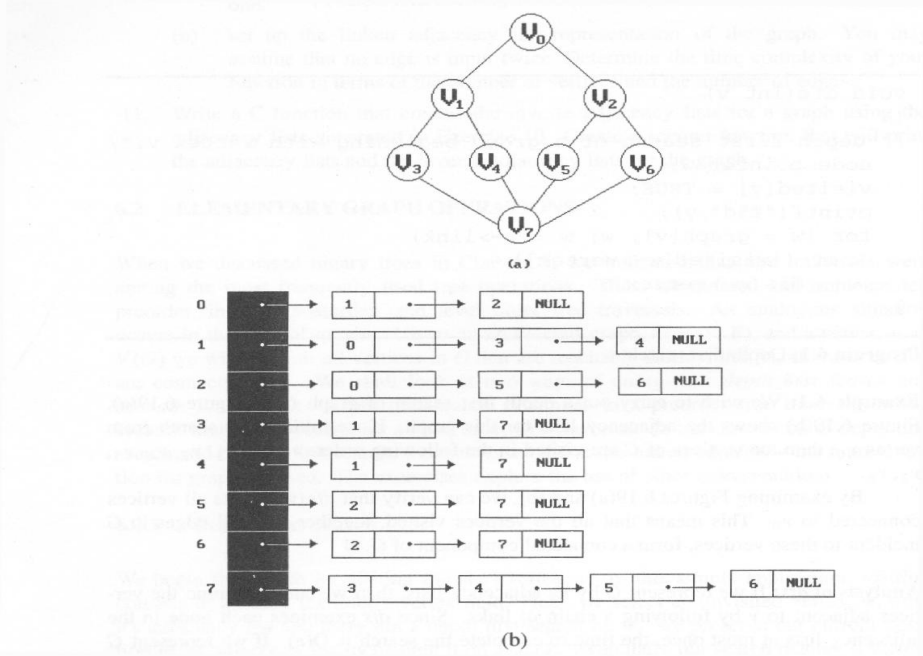
```
tree_pointer search(tree_pointer root,
                                    int key)
{
/* return a pointer to the node that contains key. If there is no such
node, return NULL */
  if (!root) return NULL;
  if (key == root->data) return root;
  if (key < root->data)
      return search(root->left_child,
                  key);
  return search(root->right_child,key);
}
```

Problem Statement: 17

Write a C program to implement Breadth first search and depth first search.

Illustration:

The graph and its equivalent adjacency list representation:



Depth-first search (DFS**)** is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors.

For th above graph:

Depth first search: V0, V1, V3, V7, V4, V5, V2, V6

Breadth first search: V0, V1, V2, V3, V4, V5, V6, V7

Structure Declaration:

For DFS:

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
   int vertex;
   struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

For BFS:

```
typedef struct queue *queue_pointer;
typedef struct queue {
   int vertex;
   queue_pointer link;
};
```

C function for depth first search(DFS):

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%5d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

C function for breadth first search(BFS):

```
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%5d", v);
  visited[v] = TRUE;
  addq(&front, &rear, v);
         while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
        printf("%5d", w->vertex);
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```