# LAB MANUAL
## On
# Data Structures using C Lab
## For MCA 2nd Semester of VTU

**Instructions to the Readers:**

- This document is a useful material for the 2nd Semester MCA students (2013 Scheme – 13MCA26) of Visvesvaraya Technological University, Belagavi, Karnatka.
- Though these programs are compiled and verified on Turbo C compiler, they can be executed on different C compilers with little/no modifications.
- Additional explanation for the program is provided, wherever necessary. However, it is advised to know the theory concepts by referring to the respective subject notes (available in the website).
- Clarifications/suggestions are welcome!!



**1. Convert a valid prefix expression into postfix expression.**

**Logic of the program:**

- Read a valid prefix expression from the keyboard as a string variable.
- Reverse this string (use strrev() function)
- Read every character from the reversed prefix expression (say, *sym*)
- Store the single character *sym* into a string variable *temp* by attaching null character (\0) to it.
- If the *sym* is an operator (like +, - etc), pop two entries from the stack. Concatenate these two popped strings and also the *sym.* Push the concatenated string (*postfix)* (*op1+op2+sym*) into the stack.
- If the *sym* was not an operator (means, it will be an alphabet/digit) then push the *temp* into the stack.
- Repeat the above steps till the end of prefix expression (*strlen(prefix)*).
- You will get the required postfix expression in the variable *postfix*.

**Note: Students are advised to give only valid prefix expression as an input to get the correct result. One can write an infix expression on a paper and convert it into prefix format manually and then it can be given as input to the program.**

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

void push(char item[], char s[][20], int *top)
{
        strcpy(s[++(*top)], item);
}

char * pop(char s[][20], int *top)
{
        return s[(*top)--];
}

void pre_post(char prefix[], char postfix[])
{
        char s[20][20], sym, *op1, *op2, temp[2];
        int top=-1, i;

        strrev(prefix);

        for(i=0;i<strlen(prefix);i++)
        {
                sym=prefix[i];
                temp[0]=sym;
                temp[1]='\0';

                switch(sym)
                {
                        case '+':
                        case '-':
                        case '*':
                        case '/':
                        case '^':    op1=pop(s, &top);
                                     op2=pop(s,&top);
                                     strcpy(postfix, op1);
                                     strcat(postfix, op2);
                                     strcat(postfix, temp);

                                     push(postfix, s, &top);
                                     break;
                        default:
                                push(temp, s, &top);
                                break;
```

```
                }
        }
}

void main()
{
        char prefix[20],postfix[20];

        printf("Enter valid prefix expression:");
        scanf("%s", prefix);
        pre_post(prefix, postfix);
        printf("\nThe postfix expression is %s", postfix);
}
```

**2. Write a C program to evaluate a given postfix expression and its values for the variables.**

**Logic of the program:**

For this program, the user can give any valid postfix expression in the form of digits or alphabets. For example, if the expression is *ab+*, you have to read the values for operands *a* and *b* from the keyboard. If the expression is *93+*, then it is assumed that 9 is one operand and 3 is another operand. Remember that, for postfix expression evaluation, single digit operands only are considered, when your expression is of digits. That is, if you want to add two numbers 12 and 3, you cannot give 123+. Because, the program reads three digits 1, 2 and 3 separately, but not as 12 and 3. In such situation, it is better to give expression in the alphabet format like *ab+*, where the operands *a* and *b* can be any value read from the keyboard.

Steps:

(i) Scan a symbol in postfix expression from left to right.

(ii) If the symbol is an operator, pop two elements from the stack and perform the operation indicated.

(iii) Push the result of step (ii) in to the stack.

(iv) If the symbol is operand, there are two possibilities:

a. Operand is a digit: It is in the form of a character. So, it has to be converted into pure number by subtracting character zero from it. For example,
Character '9' – character '0'
= 57 – 48 (ASCII values of '9' and '0')
= 9 (pure number)

b. Operand is an alphabet: Then you have to read the data from the keyboard. For example, if you need to evaluate the postfix expression ab+, you should read a and b from keyboard.

(v) Repeat the above steps till all the symbols get exhausted in the given postfix expression.

(vi) Now, pop the element from the stack, which will be the result of entire postfix expression.

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

float oper(char sym, float op1, float op2)
{
        switch(sym)
        {
                case '+': return op1 + op2;
                case '-': return op1 - op2;
                case '*': return op1 * op2;
                case '/': if(op2== 0)
                        {
                                printf("Can't evaluate");
                                exit(0);
                        }
                        return op1 / op2;
                case '^':
                case '$': return pow(op1,op2);
        }
}

void push(float item, int *top, float s[ ])
{
        s[++(*top)] = item;
}

float pop(int *top, float s[ ])
{
        return s[(*top)--];
}

void main()
{
        float s[20], result, op1, op2, x;
        int top = -1, i;
        char postfix[20], sym;
```

```
        printf("Enter valid postfix expression\n");
        scanf("%s",postfix);

        for(i=0;i<strlen(postfix);i++)
        {
                sym = postfix[i];

                if(isdigit(sym))
                        push(sym-'0', &top, s);
                else if (isalpha(sym))
                {
                        printf("Enter the value of %c: ", sym);
                        scanf("%f",&x);
                        push(x,&top,s);
                }

                else
                {
                        op2 = pop(&top,s);
                        op1 = pop(&top,s);
                        result = oper(sym,op1,op2);
                        push(result,&top,s);
                }
        }
        result = pop(&top,s);
        printf("Result =%f",result);
}
```

**3. Write a C program simulate the working of circular queue providing the following operations: insert, delete and display.**

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 3

void insert(int q[], int *f, int *r, int item)
{
        if(*f==(*r+1)%MAX)
        {
                printf("\nQueue overflow");
                return;
        }
```

```
        *r=(*r+1)%MAX;
        q[*r]=item;


        if(*f==-1)
                (*f)++;
}

void del(int q[], int *f, int *r)
{
        if(*f==-1 )
        {
                printf("\nQueue underflow");
                return;
        }

        printf("\nDeleted element is %d", q[*f]);

        if(*f==*r)
                *f=*r=-1;
        else
                *f=(*f+1)%MAX;
}

void disp(int q[], int *f, int *r)
{
        int i;

        if(*f==-1)
        {
                printf("\nNo elements to display!!");
                return;
        }
        printf("\n Contents of queue:\n");

        if(*f>*r)
        {
                for(i=*f;i<MAX;i++)
                        printf("%d\t",q[i]);
                for(i=0;i<=*r;i++)
                        printf("%d\t", q[i]);
        }
```

```
        else
        {
                for(i=*f;i<=*r;i++)
                        printf("%d\t",q[i]);
        }
}

void main()
{       int q[10], f=-1, r=-1,item, opt;

        for(;;)
        {       printf("\n*****Circular Queue operations*****");
                printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
                printf("\nEnter your option: ");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item to be inserted:");
                                scanf("%d",&item);
                                insert(q,&f,&r,item);
                                break;
                        case 2: del(q, &f, &r);
                                break;
                        case 3: disp(q, &f,&r);
                                break;
                        case 4:
                        default:exit(0);
                }
        }
}
```

**4. Demonstrate recursion –**

**a. Calculate GCD and LCM of 3 integer numbers**

**b. Solve Towers of Hanoi problem**

**c. Calculate the sum for a given number 'n' from 1 to n**

**Program to calculate GCD and LCM of 3 integer numbers.**

```
int GCD(int m, int n)
{
    if (n==0)
            return m;
```

```
    return GCD(n,m%n);
}


int LCM(int m, int n)
{
    return m*n/GCD(m,n);
}

void main()
{
    int m, n,p, gcd, lcm;

    printf("Enter m, n and p:");
    scanf("%d%d%d", &m, &n, &p);

    gcd=GCD(m,GCD(n,p));
    lcm=LCM(m, LCM(n,p));

    printf("\nGCD=%d \t LCM=%d",gcd,lcm);
}
```

**Program to solve Towers of Hanoi problem**

```
#include<stdio.h>

int count=0;

void tower(int n, char s, char t, char d)
{
    if(n==1)
    {
        printf("Move disc 1 from %c to %c ", s, d);
        count ++;
        return;
    }
    tower(n-1, s, d, t);
    printf("Move disc %d from %c to %c", n, s,d);
    count ++;
    tower(n-1, t, s, d);
}
```

```
void main()
{
        int n;

        printf("Enter the number of discs");
        scanf("%d", &n);
        tower(n, 'A','B','C');
        printf("Total number of moves =%d", count);
}
```

**Program to calculate the sum for a given number 'n' from 1 to n**

```
#include<stdio.h>

int sum(int n)
{
        if (n==1)
                return n;
        return n+sum(n-1);
}

void main()
{
        int n, s;

        printf("Enter n:");
        scanf("%d",&n);
        s=sum(n);
        printf("\nSum=%d",s);
}
```

5. **Simulate the working of a linked list providing the following operations:**
   a. **Insert at the beginning**
   b. **Insert at the end**
   c. **Insert before a given element**
   d. **Insert at the position**
   e. **Display**

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

```
struct node
{
        int data;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));
        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}

NODE insert_front(int item, NODE start)
{
        NODE temp;
        temp = getnode();
        temp->data=item;
        temp->link=start;
        return temp;
}

void display(NODE start)
{
        NODE temp;
        if(start==NULL)
        {
            printf("No element to display\n");
            return ;
        }
        printf("The contents of list:\n");

        temp=start;

        while(temp!=NULL)
        {
                printf("%d\n", temp->data);
                temp=temp->link;
```

```
        }
}

NODE insert_rear(int item, NODE start)
{
        NODE temp, cur;
        temp=getnode();
        temp->data=item;
        temp->link=NULL;

        if (start==NULL)
                return temp;

        cur=start;
        while(cur->link!=NULL)
                cur=cur->link;

        cur->link=temp;
        return start;
}

NODE insert_pos(int item, int pos, NODE start)
{
        NODE temp, cur, prev;
        int p;
        temp=getnode();
        temp->data = item;
        temp->link=NULL;

        if(pos==1)
        {       temp->link=start;
                return temp;
        }
        cur=start;
        prev=NULL;
        p=1;

        while(cur!=NULL && p!=pos)
        {
                prev=cur;
                cur=cur->link;
                p++;
        }
```

```
        if(p==pos)
        {
                prev->link=temp;
                temp->link=cur;
                return start;
        }
        else
        {
                printf("\nInvalid Position!!!");
                return start;
        }
}

NODE insert_before(int key, int item, NODE start)
{
        NODE temp,prev,cur;
        temp = getnode();
        temp->data = item;
        temp->link = NULL;

        if (start == NULL)
        {
                printf("\nInvalid request !! \n");
                return start;
        }
        if(key==start->data)
        {
                temp->link=start;
                return temp;
        }
        prev=NULL;
        cur = start;

        while(key!=cur->data && cur!=NULL)
        {
                prev=cur;
                cur=cur->link;
        }
        if(cur==NULL)
        {
                printf(" key not found !!");
                return start;
        }
```

```
        else
        {
                prev->link=temp;
                temp->link=cur;
                return start;
        }
}

void main()
{
        int opt, item, pos, key;
        NODE start=NULL;

        for(;;)
        {       printf("\n1.Insert Front\n 2.Insert Rear\n 3. Display\n");
                printf(" 4.Insert at Position\n  5.Insert Before\n");

                printf("Enter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item");
                                scanf("%d",&item);
                                start=insert_front(item,start);
                                break;
                        case 2: printf("\nenter item");
                                scanf("%d",&item);
                                start=insert_rear(item,start);
                                break;
                        case 3: display(start);
                                break;
                        case 4: printf("\nenter item");
                                scanf("%d",&item);
                                printf("\nenter the position:");
                                scanf("%d",&pos);

                                if(pos<=0)
                                        printf("\n Invalid position!!");
                                else
                                        start=insert_pos(item, pos, start);
                                break;
                        case 5: printf("\nenter item");
                                scanf("%d",&item);
```

```
                        printf("\nEnter key, before which new item has to be
                                                                inserted:");
                        scanf("%d",&key);
                        start=insert_before(key,item, start);
                        break;
                default: exit(0);
            }
        }
}
```

**6. Simulate the working of a circular linked list providing the following operations:**

a. **Delete from the beginning**
b. **Delete from the end**
c. **Delete a given element**
d. **Delete every alternate element**
e. **Display and insert is mandatory**

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

struct node
{
        int data;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));
        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}
```

```
NODE insert_front(int item, NODE last)
{
        NODE temp;
        temp = getnode();

        temp->data=item;
        temp->link=temp;

        if(last==NULL)
                return temp;

        temp ->link = last->link;
        last ->link=temp;
        return last;
}

void display(NODE last)
{
        NODE temp;

        if(last==NULL)
        {
                printf("No element to display\n");
                return ;
        }

        temp=last->link;
        printf("The contents of list:\n");

        while(temp!=last)
        {
                printf("%d\n" temp->data);
                temp=temp->link;
        }

        printf("%d", temp->data);
}

NODE delete_front(NODE last)
{
        NODE temp;
        if(last==NULL)
        {
                printf("no element to delete\n");
```

```
                return NULL;
        }
        if(last->link==last)
        {
                printf("The item deleted is %d", last->data);
                free(last);
                return NULL;
        }
        temp=last->link;
        last->link=temp->link;
        printf("Item deleted is %d", temp->data);
        free(temp);
        return last;
}

NODE delete_rear(NODE last)
{
        NODE prev;

        if(last==NULL)
        {
                printf("no element to delete\n");
                return NULL;
        }

        if(last->link==last)
        {
                printf("The item deleted is %d", last->data);
                free(last);
                return NULL;
        }
        prev=last->link;

        while(prev->link!=last)
                prev=prev->link;

        prev->link=last->link;
        printf("\nDeleted element is %d", last->data);
        free(last);
        return prev;
}

NODE del_item(int key, NODE last)
{
```

```
        NODE prev, cur;

        if(last==NULL)
        {
                printf("\nList is empty!!\n");
                return last;
        }

        if(last==last->link &&key==last->data)
        {
                printf("\nDeleted item is %d", last->data);
                free(last);
                return NULL;
        }
        prev=last;
        cur=last->link;

        while(cur!=last && key!=cur->data)
        {
                prev=cur;
                cur=cur->link;
        }

        if(key==cur->data)
        {
                prev->link=cur->link;
                printf("\nDeleted item is %d", cur->data);

                if(cur==last)
                        last=prev;

                free(cur);
        }
        else
                printf("\nKey not found\n");

        return last;
}

NODE del_alternate(NODE last)
{
        NODE cur, prev;
        int count=1, pos=1;
```

```
        if(last==NULL)
        {
                printf("\nList is empty!!");
                return last;
        }

        if(last==last->link)
        {
                printf("\nDeleted item %d", last->data);
                free(last);
                return NULL;
        }
        prev=last;
        cur=last->link;

        while(cur!=last)
        {
                if(pos%2!=0)
                {
                        NODE temp=cur;
                        printf("\nDeleted item %d", temp->data);
                        prev->link=cur->link;
                        prev=cur=cur->link;
                        free(temp);
                }
                else
                        cur=cur->link;

                pos++;
        }

        if(cur==last && pos%2!=0)
        {
                printf("\nDeleted item is %d", cur->data);
                prev->link=cur->link;
                free(cur);
                return prev;
        }
        else
                return last;
}

void main()
{
```

```
        int opt, item, key;
        NODE last=NULL;

        for(;;)
        {
                printf("\n1.Insert Front\n 2.Delete Front \n 3.Delete Rear");
                printf("\n  4.Display\n 5.Delete a given element\n");
                printf(" 6.Delete alternative element\n");
                printf("Enter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nenter item");
                                 scanf("%d",&item);
                                last=insert_front(item, last);
                                break;
                        case 2:  last=delete_front(last);
                                break;
                        case 3:  last=delete_rear(last);
                                break;
                        case 4: display(last);
                                break;
                        case 5: last=delete_front(last);
                                break;
                        case 5: printf("\nEnter key element to be deleted:");
                                scanf("%d",&key);
                                last=del_item(key, last);
                                break;
                        case 6: last=del_alternate(last);
                                break;
                        default: exit(0);
                }
        }
}
```

**7. Simulate the working of a dequeue.**

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

```
struct node
{
        int data;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));
        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}
NODE insert_front(int item, NODE start)
{
        NODE temp;
        temp = getnode();
        temp->data=item;
        temp->link=start;
        return temp;
}

NODE delete_front(NODE start)
{
    NODE temp;
    if(start==NULL)
    {
        printf("no element to delete\n");
        return start;
    }
    temp=start;
    printf("Deleted item=%d", temp->data);
    start=start->link;
    free(temp);
    return start;
}

NODE insert_rear(int item, NODE start)
{
```

```
        NODE temp, cur;
        temp=getnode();
        temp->data=item;
        temp->link=NULL;

        if (start==NULL)
                return temp;

        cur=start;
        while(cur->link!=NULL)
                cur=cur->link;

        cur->link=temp;
        return start;
}

void display(NODE start)
{
        NODE temp;
        if(start==NULL)
        {
            printf("No element to display\n");
            return ;
        }
        printf("The contents of list:\n");

        temp=start;

        while(temp!=NULL)
        {
                printf("%d\n", temp->data);
                temp=temp->link;
        }
}

NODE delete_rear(NODE start)
{
        NODE prev, cur;

        if(start==NULL)
        {       printf("no element to delete\n");
                return start;
        }
```

```
        if(start->link==NULL)
        {       printf("\nDeleted element is%d", start->data);
                free(start);
                return NULL;
        }

        prev=NULL;
        cur=start;

        while(cur->link!=NULL)
        {       prev=cur;
                cur=cur->link;
        }
        printf("\nDeleted element is %d", cur->data);     .
        free(cur);
        prev->link=NULL;
        return start;
}

void main()
{       int opt, item;
        NODE start=NULL;

        for(;;)
        {
                printf("1.Insert Front\n 2.Insert Rear\n 3. Display\n");
                printf(" 4.Delete Front\n  5.Delete Rear\n");
                printf("enter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nenter item");
                                scanf("%d",&item);
                                 start=insert_front(item,start);
                                 break;
                        case 2: printf("\nenter item");
                                 scanf("%d",&item);
                                 start=insert_rear(item,start);
                                 break;
                        case 3: display(start);
                                 break;
                        case 4: start=delete_front(start);
                                 break;
```

```
                        case 5: start=delete_rear(start);
                                break;
                        default: exit(0);
                    }
                }
            }
```

**8. Simulate the working of a double linked list to implement stack and queue.**
**Answer:**
We know that to implement stack, we need following operations:

- Insert at front
- Delete from front
- Display

To implement queue, we need:

- Insert at rear
- Delete from front
- Display

Hence, we have to perform insert_front(), insert_rear(), delete_front() and display() operations on doubly linked list.

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

struct node
{
        struct node *llink;
        int data;
        struct node *rlink;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));

        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}
```

```
NODE insert_front(int item, NODE start)
{
        NODE temp;
        temp = getnode();
        temp->llink=NULL;
        temp->data=item;
        temp->rlink=start;
        start->llink=temp;
        return temp;
}

void display(NODE start)
{
        NODE temp;

        if(start==NULL)
        {
                printf("List is empty\n");
                return ;
        }

        temp=start;
        printf("\nThe contents of list:\n");

        while(temp!=NULL)
        {
                printf("%d\n", temp->data);
                temp=temp->rlink;
        }
}

NODE delete_front(NODE start)
{
        NODE temp;
        if(start==NULL)
        {
                printf("\nList is empty!!");
                return start;
        }
        temp=start;
        printf("\nDeleted item %d", temp->data);
        start=start->rlink;
        start->llink=NULL;
```

```
        free(temp);
        return start;
}

NODE insert_rear(int item, NODE start)
{
        NODE temp, cur;
        temp=getnode();
        temp->llink=NULL;
        temp->data=item;
        temp->rlink=NULL;

        if(start==NULL)
                return temp;

        cur=start;
        while(cur->rlink!=NULL)
                cur=cur->rlink;

        cur->rlink=temp;
        temp->llink=cur;
        return start;
}

void main()
{
        int opt, ch, item;
        NODE start=NULL;
        printf("\n1. Stack Implementation\n2.Queue Implementation\n 3.Exit\n");
        printf("Enter your choice:");
        scanf("%d",&ch);

        switch(ch)
        {
                case 1: printf("\n***Stack using Double linked list****\n");
                        for(;;)
                        {
                                printf("\n1. Push \n2.Pop \n3.Display \n4.Exit");
                                printf("\nEnter you option:");
                                scanf("%d",&opt);

                                switch(opt)
                                {
                                        case 1: printf("\nEnter item to be inserted:");
```

```
                                scanf("%d",&item);
                                start=insert_front(item, start);
                                 break;
                        case 2: start=delete_front(start);
                                 break;
                        case 3: display(start);
                                break;
                        default: exit(0);
                }
        }
        break;

case 2: printf("\n***Queue using Double linked list****\n");
         for(;;)
         {
                printf("\n1.Insert \n2.Delete \n3.Display \n4.Exit");
                printf("\nEnter you option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item to be inserted:");
                                scanf("%d",&item);
                                start=insert_rear(item, start);
                                break;
                        case 2: start=delete_front(start);
                                break;
                        case 3: display(start);
                                break;
                        default: exit(0);
                }
        }
        break;
        default: exit(0);
    }
}
```

**9. Create a binary tree and implement the tree traversal techniques of inorder, preorder and post order.**

```
#include<stdio.h>
struct node
{
```

```
        struct node *llink;
        struct node *rlink;
        int data;
};
typedef struct node *NODE;

NODE getnode()          //Function to get memory from heap
{
        NODE x;
        x = (NODE) malloc (sizeof (struct node));

        if (x == NULL)
        {
                printf("No memory space\n");
                exit(0);
        }
        return x;
}

/* Function to create binary search tree by inserting elements */
NODE insert (int item, NODE root)
{
        NODE temp,prev,cur;

        temp = getnode();
        temp->data = item;
        temp->rlink = NULL;
        temp->llink = NULL;

        if(root == NULL)
                return temp;

        prev = NULL;
        cur = root;

        while (cur != NULL)
        {
                prev = cur;
                cur = (item < cur->data) ? cur->llink : cur->rlink;
        }
        if (item < prev->data)
                prev->llink = temp;
        else
                prev->rlink = temp;
```

```
        return root;
}

void preorder(NODE root)          //Function to traverse tree in pre-order
{
        if(root != NULL)
        {
                printf("%d\t",root->data);
                preorder(root->llink);
                preorder(root->rlink);
        }
}

void inorder(NODE root)           //Function to traverse tree in in-order
{
        if(root != NULL)
        {
                inorder(root->llink);
                printf("%d\t",root->data);
                inorder(root->rlink);
        }
}

void postorder(NODE root)         //Function to traverse tree in post-order
{
        if(root != NULL)
        {
                postorder(root->llink);
                postorder(root->rlink);
                printf("%d\t",root->data);
        }
}

void main()
{
        NODE root = NULL;
        int opt,item;

        for(; ;)
        {
                printf("\nCreating a Binary Tree and Traversing the tree\n");
                printf("Enter your option\n");
                printf("1:Insert an element to tree\n");
```

```
                printf("2:Pre-Order Traversal\n");
                printf("3:In-Order Traversal\n");
                printf("4:Post-Order Traversal\n");
                printf("5:Exit\n");
                scanf("%d",&opt);
                switch(opt)
                {
                  case 1: printf("Enter the element to be inserted \n");
                          scanf("%d",&item);
                          root = insert(item,root);
                          break;
                  case 2: printf("PREORDER TRAVERSAL\n");
                          preorder(root);
                          break;
                  case 3: printf("INORDER TRAVERSAL\n");
                          inorder(root);
                          break;
                  case 4: printf("POSTORDER TRAVERSAL\n");
                          postorder(root);
                          break;
                  case 5:
                  default: exit(0);
                }
        }
}
```

**10. Implement quick sort**

```
#include<stdio.h>
quick_sort(int x[], int low, int high)
{
        int pos;
        if (low < high)
        {
                pos = partition(x,low,high);
                quick_sort(x,low,pos-1);
                quick_sort(x,pos+1,high);
        }
        return;
}

int partition(int x[], int low, int high)          //Function for partitioning the array
{       int key, temp, true = 1, left, right;
        key = x[low];
```

```
        left = low +1;
        right = high;

        while(true)
        {
                while ((left < high) && (key >= x[left]))
                        left++;
                while(key < x[right])
                        right--;


                if(left < right)
                {
                        temp = x[left];
                        x[left] = x[right];
                        x[right] = temp;
                }
                else
                {       temp = x[low];
                        x[low] = x[right];
                        x[right] = temp;
                        return(right);
                }
        }
        return 0;
}

void main()
{
        int a[10],n,i,low,high;

        printf("Enter array size\n");
        scanf("%d",&n);
        printf("Enter the elements\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        low = 0;
        high = n-1;
        quick_sort(a,low,high);

        printf("The sorted list is \n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
```

```
}
```

**11. Implement heap sort**

```
# include<stdio.h>
void createheap(int a[], int n)
{
     int i,j,k,item;
     for(k=1;k<n;k++)
     {
          item=a[k];
          i=k;
          j=(i-1)/2;

          while(i>0 && item>a[j])
          {
               a[i]=a[j];
               i=j;
               j=(i-1)/2;
          }
          a[i]=item;
     }
}

void heapsort(int a[],int n)
{
    int i, temp;
    createheap(a,n);
    for(i=n-1;i>0;--)
    {
         temp=a[0];
         a[0]=a[i];
         a[i]=temp;
         createheap(a,i);
    }
}

void main()
{
    int a[20],n, i;
    printf("enter number of elements in array");
    scanf("%d", &n);
    printf("enter elements:\n");
```

```
    for(i=0;i<n;i++)
        scanf("%d", &a[i]);
    heapsort(a,n);
    printf("\n sorted list:\n");
    for(i=0;i<n;i++)
        printf("%d\n", a[i]);
}
```

**12. Implement the search techniques of**

**a. Linear search**

**b. Binary search**

**Program for Linear search**

```
#include<stdio.h>


int linear(int a[], int key, int n)
{
        int i;

        for(i=0;i<n;i++)
                if(a[i]==key)
                        return i+1;

        return -1;
}
void main()
{
    int a[20], n, key, i, pos;

    printf("Enter size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");

    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    printf("\nEnter the key to be searched:");
    scanf("%d",&key);

    pos=linear(a, key, n);
    if(pos==-1)
        printf("\nUnsuccessful search!!");
    else
```

```
            printf("\nKey found at position %d",pos);
        }
```

**Program for Binary search:**

```
#include<stdio.h>

int binary(int item, int a[],int low,int high)
{
        int mid;

        if(low<=high)
        {
                mid=(low+high)/2;

                if(a[mid]==item)
                        return mid+1;
                if(a[mid]>item)
                        return binary(item,a,low,mid-1);

                return binary(item,a,mid+1,high);
        }
        return -1;
}

void main()
{
        int a[10],n,pos,item, i;

        printf("Enter the array size:");
        scanf("%d",&n);
        printf("Enter the elements in ascending order\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("Enter the key element:");
        scanf("%d",&item);

        pos=binary(item,a,0,n-1);

        if(pos==-1)
                printf("\nItem not found");
        else
                printf("\nItem found at position %d",pos);
}
```

**13. Write a program to**

**a. Create AVL Tree**
**b. Insert element to AVL tree**
**c. Find the height of the AVL tree**

**Logic:**
The creation of AVL tree requires every number to be read one after the other. When each element is read from the keyboard, it has to be inserted into the existing AVL tree at a proper position (left subtree or right subtree). Then, the balance factor for every node has to be evaluated. If any imbalance is found, we need to perform rotation. The type of rotation depends on the value of balance factor (negative – left rotation, positive – right rotation). LR and RL rotation have to be made when the height of imbalanced node is more than 2. But, observe that LR rotation is left rotation and right rotation in that order. Whereas, RL rotation is right rotation and then left rotation.

One can check the insertion by displaying the contents of the AVL tree. For this, inorder traversal is used. Since inorder traversal of AVL tree (which is also a binary search tree) gives the elements in an ascending order, it will be easy to verify the proper insertion/working of the program.

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

struct node
{
        int data;
        struct node *llink;
        struct node *rlink;
        int ht;
};
typedef struct node *NODE;

NODE getnode()
{
   NODE x;
   x=(NODE) malloc(sizeof(struct node));
   if(x==NULL)
   {
        printf("\nAllocation failed!!");
        exit(0);
   }
   return x;
```

```
}

int height(NODE N)
{
        if (N == NULL)
                return 0;
        return N->ht;
}

// Get Balance factor of node N
int getBalance(NODE N)
{
        if (N == NULL)
                return 0;
        return height(N->llink) - height(N->rlink);
}

// Function to right rotate subtree rooted with y
NODE rightRotate(NODE y)
{
        NODE x = y->llink;
        NODE T2 = x->rlink;

        // Perform rotation
        x->rlink = y;
        y->llink = T2;

        // Update heights
        y->ht = max(height(y->llink), height(y->rlink))+1;
        x->ht = max(height(x->llink), height(x->rlink))+1;

        // Return new root
        return x;
}

// Function to left rotate subtree rooted with x
NODE leftRotate(NODE x)
{
        NODE y = x->rlink;
        NODE T2 = y->llink;

        // Perform rotation
        y->llink = x;
        x->rlink = T2;
```

```
        //  Update heights
        x->ht = max(height(x->llink), height(x->rlink))+1;
        y->ht = max(height(y->llink), height(y->rlink))+1;

        // Return new root
        return y;
}

NODE insert(NODE root, int item)
{
        NODE temp;
        int balance;

        temp=getnode();
        temp->llink=NULL;
        temp->data=item;
        temp->rlink=NULL;
        temp->ht=1;

        if (root == NULL)
          return temp;

        if (item < root->data)
         root->llink  = insert(root->llink, item);
        else
         root->rlink = insert(root->rlink, item);

         /* Update height of this ancestor node */
         root->ht=max(height(root->llink), height(root->rlink)) +1;

         /// Get the balance factor
         balance = getBalance(root);

        // If this node becomes unbalanced, then there are 4 cases

        // Right rotate
         if (balance > 1 && item < root->llink->data)
                return rightRotate(root);

        // Left rotate
         if (balance < -1 && item > root->rlink->data)
                return leftRotate(root);
```

```
            // Left Right rotate
               if (balance > 1 && item > root->llink->data)
               {      root->llink =  leftRotate(root->llink);
                      return rightRotate(root);
               }

            // Right Left rotate
               if (balance < -1 && item < root->rlink->data)
               {      root->rlink=rightRotate(root->rlink);
                      return leftRotate(root);
               }

               return root;
}

void inorder(NODE root)
{
        if(root != NULL)
        {
                inorder(root->llink);
                printf("%d\t", root->data);
                inorder(root->rlink);
        }
}
void main()
{
        NODE root = NULL;
        int opt, item;

        for(;;)
        {
                printf("\n*****AVL Tree Operations*****");
                printf("\n1.Insert \n2.Display using Inorder \n3.Find height \n4.Exit");
                printf("\nEnter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1:printf("\nEnter item to be inserted:");
                               scanf("%d",&item);
                               root=insert(root, item);
                               break;
                        case 2: printf("\nInorder traversal of AVL tree is:\n");
                                inorder(root);
```

```
                    break;
            case 3: printf("\nHeight is %d", height(root));
                    break;
            case 4:
            default: exit(0);
        }
    }
}
```