prg1: RecBS and IterBS:

```
#include <stdio.h>
#define COMPARE(a, b) ((a) == (b) ? 0 : ((a) < (b) ? -1 : 1))

int RBS(int arr[], int left, int right, int target);
int IBS(int arr[], int size, int target);

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6};
    int size = sizeof(arr) / sizeof(arr[0]); // Corrected size calculation
    int target = 2;

    int res1 = RBS(arr, 0, size - 1, target); // Fixed function call syntax
    int res2 = IBS(arr, size, target); // Fixed function call syntax

    if (res1 == -1) {
        printf("Target %d not found in Recursive Binary Search\n", target);
    } else {
        printf("Target %d found at index %d in Recursive Binary Search\n", target, res1);
    }

    if (res2 == -1) {
        printf("Target %d not found in Iterative Binary Search\n", target);
    } else {
        printf("Target %d found at index %d in Iterative Binary Search\n", target, res2);
    }

    return 0;
}

int RBS(int arr[], int left, int right, int target) {
    if (left > right) {
        return -1; // Corrected base case condition
    }
    int mid = (left + right) / 2; // Fixed 'mid' declaration
    if (COMPARE(arr[mid], target) == 0) {
        return mid;
    } else if (COMPARE(arr[mid], target) < 0) {
        return RBS(arr, mid + 1, right, target); // Fixed recursive call syntax
    } else {
        return RBS(arr, left, mid - 1, target); // Fixed recursive call syntax
    }
}

int IBS(int arr[], int size, int target) {
```

```
    int left = 0, right = size - 1; // Fixed variable declarations

    while (left <= right) { // Changed 'if' to 'while' for iterative search
        int mid = (left + right) / 2; // Fixed 'mid' declaration

        if (COMPARE(arr[mid], target) == 0) {
            return mid;
        } else if (COMPARE(arr[mid], target) < 0) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // Target not found
}
```

prg 2: Fast Transpose:

```
#include <stdio.h>

typedef struct {
    int r, c, v;
} term;

void transpose(term a[], term t[]) {
    int rt[10], sp[10];
    int i, j, numcols = a[0].c, numterms = a[0].v;

    // Initialize the header of the transposed matrix
    t[0].r = numcols;
    t[0].c = a[0].r;
    t[0].v = numterms;

    if (numterms > 0) {
        // Step 1: Initialize row terms to 0
        for (i = 0; i < numcols; i++) {
            rt[i] = 0;
        }

        // Step 2: Count the number of elements in each column of the original matrix
        for (i = 1; i <= numterms; i++) {
            rt[a[i].c]++;
        }
```

```
        // Step 3: Set starting positions for each column in the transposed matrix
        sp[0] = 1;
        for (i = 1; i < numcols; i++) {
            sp[i] = sp[i - 1] + rt[i - 1];
        }

        // Step 4: Populate the transposed matrix
        for (i = 1; i <= numterms; i++) {
            j = sp[a[i].c]++;
            t[j].r = a[i].c;
            t[j].c = a[i].r;
            t[j].v = a[i].v;
        }
    }
}

int main() {
    term a[10], t[10];
    int i;

    // Input the original matrix
    printf("\nEnter the number of rows and columns: ");
    scanf("%d%d", &a[0].r, &a[0].c);
    printf("\nEnter the number of non-zero values: ");
    scanf("%d", &a[0].v);

    for (i = 1; i <= a[0].v; i++) {
        printf("\nEnter the row, column, and value for element %d: ", i);
        scanf("%d%d%d", &a[i].r, &a[i].c, &a[i].v);
    }

    // Display the original matrix
    printf("\nOriginal Matrix (in sparse format):\n");
    printf("Row\tCol\tValue\n");
    for (i = 1; i <= a[0].v; i++) {
        printf("%d\t%d\t%d\n", a[i].r, a[i].c, a[i].v);
    }

    // Perform transpose
    transpose(a, t);

    // Display the transposed matrix
    printf("\nTranspose Matrix (in sparse format):\n");
    printf("Row\tCol\tValue\n");
    for (i = 1; i <= t[0].v; i++) {
        printf("%d\t%d\t%d\n", t[i].r, t[i].c, t[i].v);
```

```
    }

    return 0;
}
```

prog 3: Circular Q operaaions:

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

typedef struct {
    int *arr;
    int rear, front, size;
} cirQ;

void initQ(cirQ *q, int size) {
    q->arr = (int *)malloc(size * sizeof(int));
    if (q->arr == NULL) {
        printf("Memory allocation failed\n");
        exit(1); // Exit if memory allocation fails
    }
    q->rear = q->front = -1;
    q->size = size; // Assign the size correctly
}

int ISFULL(cirQ *q) {
    return (q->rear + 1) % q->size == q->front;
}

int ISEMPTY(cirQ *q) {
    return q->front == -1; // Fixed incorrect comparison
}

void insertQ(cirQ *q, int item) {
    if (ISFULL(q)) {
        printf("Queue is full, can't insert\n");
        return; // Exit the function if the queue is full
    }
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear = (q->rear + 1) % q->size;
    q->arr[q->rear] = item;
    printf("Inserted %d into the queue\n", item);
}
```

```
void deleteQ(cirQ *q) {
    if (ISEMPTY(q)) {
        printf("Queue is empty, can't delete\n");
        return; // Exit the function if the queue is empty
    }
    int deleteitem = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1; // Queue becomes empty
    } else {
        q->front = (q->front + 1) % q->size;
    }
    printf("Deleted %d from the queue\n", deleteitem);
}

void display(cirQ *q) {
    if (ISEMPTY(q)) {
        printf("Queue is empty, can't display\n");
        return; // Exit the function if the queue is empty
    }
    int i = q->front;
    printf("Queue elements: ");
    while (i != q->rear) {
        printf("%d ", q->arr[i]);
        i = (i + 1) % q->size;
    }
    printf("%d\n", q->arr[q->rear]); // Print the last element
}

void freeQ(cirQ *q) {
    free(q->arr);
}

int main() {
    cirQ q; // Changed to an instance instead of a pointer
    int size;
    printf("Enter the size of the queue: ");
    scanf("%d", &size);
    initQ(&q, size);

    int choice, item;
    do {
        printf("\nCircular Queue Operations:\n");
        printf("1 - Insert\n");
        printf("2 - Delete\n");
        printf("3 - Display\n");
        printf("4 - Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter item to insert: ");
                scanf("%d", &item);
                insertQ(&q, item);
                break;
            case 2:
                deleteQ(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting program\n");
                break;
            default:
                printf("Invalid choice\n");
                break;
        }
    } while (choice != 4);

    freeQ(&q); // Free memory before exiting
    return 0;
}
```

prg 4: Multiple Stacks :

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

#define MAX_STACKS 5

typedef struct
{
        int key;
} ele;

typedef struct stack *stackPtr;

typedef struct stack{
        ele data;
        stackPtr link;
} stack;
```

```
stackPtr top[MAX_STACKS];

void push(int i, int item)
{
        stackPtr temp;
        temp=(stackPtr) malloc(sizeof(stack));
        temp->data.key = item;
        temp->link = top[i];
        top[i] = temp;
}

void pop(int i)
{
        stackPtr temp = top[i];
        int item;
        item = temp->data.key;
        top[i] = temp->link;
        free(temp);
        printf("Popped %d from stack %d\n", item, i);
}

void display()
{
        int i;
        stackPtr j;

        for(i=0;i<MAX_STACKS;i++)
        {
                printf("Stack no.%d :\n",i+1);
                if(top[i] == NULL)
       printf("Stack Empty\n----------\n");
      else
                {
                        for(j = top[i]; j != NULL ; j = j->link)
          printf("%d\t",j->data.key);
                        printf("\n----------\n");
                }
   }
}


int main()
{
   int choice, i, j;
        ele x;
```

```
   for(i=0;i<MAX_STACKS;i++)
      top[i] = NULL;


   while(1)
         {
                  printf("1.push\n2.pop\n3.display\n4.exit\n");
         printf("Enter your choice\n");
      scanf("%d",&choice);

      switch(choice)
      {
         case 1:
                                    printf("Enter the stack number(0-%d) and element to be
added\n",MAX_STACKS-1);
            scanf("%d%d",&i ,&x.key);//x is the element to be pushed
            push(i,x.key);
            break;

         case 2:
                                    printf("Enter the queue number(0-%d)\n",MAX_STACKS-1);
            scanf("%d",&i);
            if(top[i] == NULL)
               printf("Queue Empty\n");
            else
               pop(i);
            break;

         case 3:
            display();
            break;

         case 4:
            exit(0);
            break;

         default :
            printf("Invalid Choice");
      }
         }
   return 0;
}
```

prg 5 : Pstfix evaluation:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define STACKSIZE 100

int stack[STACKSIZE];
int top=-1;

int pop()
{
   return stack[top--];
}

void push(int n)
{
   stack[++top] = n;
}


int result(int op1, int op2, char operator)
{
   switch(operator)
   {
      case '+':return op1+op2;
      case '-':return op1-op2;
      case '*':return op1*op2;
      case '/':return op1/op2;
      case '%':return op1%op2;
   }
}


int postfixEval(char *str)
{
   int i;
   int op1, op2;
   for(i=0;i<strlen(str);i++)
   {
      if(isdigit(str[i]))
      {
         push(str[i]-'0');
      }
      else
      {
         op2=pop();
         op1=pop();
```

```
            push(result(op1, op2, str[i]));
        }
    }
    return pop();//since top of the stack has the answer
}

int main()
{
    char str[100];
    printf("Enter the Postfix Expression :\n");
    scanf("%s", str);

    printf("Result = %d\n", postfixEval(str));
    return 0;
}
```

prog 6 : kmp search :

```
#include<stdio.h>
#include<string.h>
int failure[20];

void fail(char *pat)
{
        int i,j;
        int n=strlen(pat);
        failure[0]=-1;
        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;
        }
}

int match(char *string, char *pat)
{
        int i=0,j=0;
        int lens=strlen(string);
        int 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);
}

int main()
{
        int i;
        char str[30],pat[20];
        printf("\nEnter a string\n");
        scanf("%s",str);
        printf("\nEnter a substring\n");
        scanf("%s",pat);
        fail(sub);
        i=match(str,pat);
        if(i==-1)
                printf("\nPattern %s Not found", pat);
        else
                printf("\nPattern %sFound at position %d",pat,i+1);

        return 0;
}
```

Prog 7 : multiple queues:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXQUEUES 10

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

nodePtr front[MAXQUEUES];
```

```
nodePtr rear[MAXQUEUES];

void push(int i, int data)
{
   nodePtr newNode = (nodePtr)malloc(sizeof(node));
   newNode->data = data;
   newNode->link =NULL;

   if(front[i]==NULL)
      front[i] = newNode;
   else
      rear[i]->link = newNode;

   rear[i] = newNode;
}

void pop(int i)
{
   if(front[i])
   {
      nodePtr temp = front[i];
      printf("Popped : %d from Queue no.%d\n", front[i]->data, i);

      front[i] = front[i]->link;
      free(temp);
   }
   else
   {
      printf("Queue no.%d is EMPTY\n", i);
   }
}

void display(int i)
{
   printf("\nQueue no.%d\n", i);
   if(front[i])
   {
      nodePtr temp = front[i];
      for(; temp!=NULL; temp = temp->link)
         printf("%5d", temp->data);
   }
   else
   {
      printf("Queue %d Empty", i);
   }
   printf("\n");
```

```
}

int main()
{
   for(int i=0;i<MAXQUEUES; i++)
   {
      front[i] = NULL;
      rear[i] = NULL;
   }

   int choice, i, data;
   printf("MENU\n1.push\n2.pop\n3.display\n4.exit\n\n\n");

   do {
      printf("choice : ");
      scanf("%d", &choice);

      switch(choice)
      {
         case 1:
         printf("Queue no(0-9) : ");
         scanf("%d", &i);

         printf("Element : ");
         scanf("%d", &data);

         push(i, data);
         break;

         case 2:
         printf("Queue no(0-9) : ");
         scanf("%d", &i);

         pop(i);
         break;

         case 3:
         printf("Queue no(0-9) : ");
         scanf("%d", &i);

         display(i);
         break;

         case 4:
         printf("Exit\n");
         break;
```

```
            default:printf("Invalid\n");
        }
        printf("\n");

    } while(choice!=4);

    return 0;
}
```

Prog 8 : Circular poly addition:

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

// Structure for a node in the circular linked list
typedef struct Node {
    int coeff;
    int exp;
    struct Node *next;
} Node;

// Function to create a new node
Node* createNode(int coeff, int exp) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = newNode;  // Circular linked list
    return newNode;
}

// Function to insert a term into the polynomial
void insertTerm(Node* head, int coeff, int exp) {
    Node* newNode = createNode(coeff, exp);
    Node* temp = head;

    while (temp->next != head && temp->next->exp > exp) {
        temp = temp->next;
    }

    if (temp->next->exp == exp) {
        temp->next->coeff += coeff;
        free(newNode);
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
```

```
    }
}

// Function to create a polynomial with predefined values
Node* createPolynomial(int terms[][2], int n) {
    Node* head = createNode(0, -1);  // Header node
    head->next = head;

    for (int i = 0; i < n; i++) {
        insertTerm(head, terms[i][0], terms[i][1]);
    }
    return head;
}

// Function to display a polynomial
void displayPolynomial(Node* head) {
    Node* temp = head->next;
    while (temp != head) {
        printf("%dx^%d", temp->coeff, temp->exp);
        temp = temp->next;
        if (temp != head) {
            printf(" + ");
        }
    }
    printf("\n");
}

// Function to add two polynomials
Node* addPolynomials(Node* p1, Node* p2) {
    Node* result = createNode(0, -1); // Header node
    result->next = result;
    Node* temp1 = p1->next;
    Node* temp2 = p2->next;

    while (temp1 != p1 || temp2 != p2) {
        if (temp1 == p1) {
            insertTerm(result, temp2->coeff, temp2->exp);
            temp2 = temp2->next;
        } else if (temp2 == p2) {
            insertTerm(result, temp1->coeff, temp1->exp);
            temp1 = temp1->next;
        } else if (temp1->exp > temp2->exp) {
            insertTerm(result, temp1->coeff, temp1->exp);
            temp1 = temp1->next;
        } else if (temp1->exp < temp2->exp) {
            insertTerm(result, temp2->coeff, temp2->exp);
```

```
            temp2 = temp2->next;
        } else {
            insertTerm(result, temp1->coeff + temp2->coeff, temp1->exp);
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
    }
    return result;
}

int main() {
    int poly1_terms[][2] = {{5, 3}, {4, 2}, {2, 0}}; // 5x^3 + 4x^2 + 2x^0
    int poly2_terms[][2] = {{3, 3}, {1, 1}, {6, 0}}; // 3x^3 + 1x^1 + 6x^0
    int n1 = sizeof(poly1_terms) / sizeof(poly1_terms[0]);
    int n2 = sizeof(poly2_terms) / sizeof(poly2_terms[0]);

    Node* poly1 = createPolynomial(poly1_terms, n1);
    Node* poly2 = createPolynomial(poly2_terms, n2);

    printf("First Polynomial: ");
    displayPolynomial(poly1);

    printf("Second Polynomial: ");
    displayPolynomial(poly2);

    Node* result = addPolynomials(poly1, poly2);
    printf("Resultant Polynomial: ");
    displayPolynomial(result);

    return 0;
}
```

Prog 9: doubly linked list:

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

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

nodePtr head;

void dinsert()
```

```
{
        int n;
        nodePtr temp;
        printf("Enter the info for the new node");
        scanf("%d", &n);
        temp=(nodePtr)malloc(sizeof(node));
        temp->data=n;
   temp->llink = head;
   temp->rlink = head->rlink;
   head->rlink-> llink = temp;
   head->rlink = temp;
}

void ddelete()
{
        nodePtr temp=head->rlink;
        if (head->rlink == head)
                printf("Deletion of head node not permitted.\n");
        else
        {
                head->rlink = temp->rlink;
                temp->rlink->llink = head;
                printf("removing node with data %d\n",temp->data);
                free(temp);
   }
}

void displayRight()
{
        nodePtr temp;
        if (head->rlink == head)
                printf("Empty list.\n");
        else
        {
                for(temp=head->rlink; temp->rlink != head; temp = temp->rlink)
                        printf("%d\t", temp->data);
                printf("%d\t", temp->data);
                printf("\n\n");
        }
}

void displayLeft()
{
        nodePtr temp;
        if (head->llink == head)
                printf("Empty list.\n");
```

```
        else
        {
                for(temp=head->llink; temp->llink != head; temp = temp->llink)
                        printf("%d\t", temp->data);
                printf("%d\t", temp->data);
                printf("\n\n");
        }
}

int main()
{
        unsigned int choice;
        head=(nodePtr)malloc(sizeof(node));
        head->rlink=head;
        head->llink=head;

        while(1)
        {
                printf("1:insert a node in DLL \n2:delete a node from DLL \n3:display the
DLL forward\n4:display the DLL forward\n5:exit\n");
                scanf("%u", &choice);
                switch(choice)
                {
                        case 1: dinsert();
                                        break;
                        case 2: ddelete();
                                        break;
                        case 3: displayRight();
                                        break;
                        case 4: displayLeft();
                                        break;
                        case 5: exit(0);
                                        break;
                        default: printf("Invalid choice... try again\n");
                }
        }
        return 0;
}
```

Prog 10: Max heap :

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

#define MAX_ELEMENTS 25
```

```
int heap[MAX_ELEMENTS];
int n = 0;

void push(int item)
{
        int i;

        i= ++n;
        while((i!=1) && ( item > heap[i/2]))
        {
                heap[i] = heap[i/2];
                i = i/2;
        }
        heap[i] = item;
}

void pop()
{
        int item;
        int temp;
        int parent, child;
        if(n==0)
                printf("heap is empty\n");
        else
        {
                item = heap[1];
                temp = heap[n--];
                parent = 1;
                child = 2;
                while(child <= n)
                {
                        if(child < n && (heap[child] < heap[child+1]))
                                child++;
                        if(temp >= heap[child])
                                break;
                        heap[parent] = heap[child];
                        parent = child;
                        child *= 2;
                }
                heap[parent] = temp;
                printf("Element removed from heap is %d\n", item);
        }
}

void display()
{
```

```
        int i;
        for(i=1; i<=n; i++)
                printf("%d\t", heap[i]);
        printf("\n");
}

int main()
{
        unsigned int choice;
        int x;
        while(1)
        {
                printf("1:insert a node to heap \n2:delete a node from heap \n3:display the max
heap\n4:exit\n");
                scanf("%u", &choice);
                switch(choice)
                {
                        case 1: if(n == MAX_ELEMENTS)
                                        {
                                                printf("Heap is full\n");
                                                exit(1);
                                        }
                                        printf("Enter the element to be added to heap\n");
                scanf("%d",&x);//x is the element to be pushed
                                        push(x);
                                        break;
                        case 2: pop();
                                        break;
                        case 3: display();
                                        break;
                        case 4: exit(0);
                                        break;
                        default: printf("Invalid choice... try again\n");
                }
        }
        return 0;
}
```

Prog 11: BST :

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

typedef struct node* treeptr;
typedef struct node
{
```

```
        int data;
    treeptr left;
    treeptr right;
}node;

treeptr createNode(int value)
{
    treeptr newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}


treeptr insert(treeptr root, int data)
  {
    if (root == NULL) return createNode(data);
    if (data < root->data)
      root->left  = insert(root->left, data);
    else if (data > root->data)
      root->right = insert(root->right, data);

    return root;
  }

void search(treeptr root, int data)
  {
    if (root == NULL)
      {
            printf("key not found\n");
            return;
      }
    else if (data == root->data)
            printf("key found in the BST\n");
      else if (data < root->data)
      search(root->left, data);
    else if (data > root->data)
      search(root->right, data);
  }

void inorder(treeptr root)
{
    if(root == NULL)
            return;
      inorder(root->left);
```

```
        printf("%d ->", root->data);
        inorder(root->right);
}


int main()
{
        treeptr root = NULL;
           int key;
           char ch='y';
           while (ch == 'y')
           {
                    printf("Enter a key to insert in BST\n");
                    scanf("%d", &key);
                    getchar();
           root = insert(root, key);
                    printf("do you wish to enter another key into BST (y/n)\n");
                    scanf("%c", &ch);
           }

           printf("Keys in inorder traversal\n");
        inorder(root);
           printf("\n");
           printf("Enter the search Key\n");
           scanf("%d", &key);
           search(root, key);

}
```

Prog 12 : dfs :

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

#define TRUE 1
#define FALSE 0

typedef struct node
{
   struct node *link;
   int vertex;
}node;

node *G[20];
```

```
int visited[20];
int n;

void insert(int vi,int vj)
{
   node *p,*q;
         q=(node*)malloc(sizeof(node));
   q->vertex=vj;
   q->link=NULL;
   if(G[vi]==NULL)
      G[vi]=q;
   else
   {   for(p=G[vi];p->link!=NULL; p=p->link);
      p->link=q;
   }
}

void read_graph()
{
   int i,vi,vj,no_of_edges;
   printf("Enter number of vertices:");
         scanf("%d",&n);

         for(i=0;i<n;i++)
      G[i]=NULL;
   printf("Enter number of edges \n");
   scanf("%d",&no_of_edges);
   for(i=0;i<no_of_edges;i++)
   {
      printf("Enter an edge(u v):");
                  scanf("%d%d",&vi,&vj);
                  insert(vi,vj);
   }
}

void DFS(int i)
{
   node *p;
         printf("%5d",i);
   visited[i]=TRUE;
   for(p=G[i];p; p=p->link)
   {
           if(!visited[p->vertex])
         DFS(p->vertex);
   }
}
```

```
int main()
{
   int i;
   read_graph();
   for(i=0;i<n;i++)
      visited[i]=FALSE;
         printf("\nNodes visited in DFS order\n");
   DFS(1);
         printf("\n");
         return 0;
}
```