1.Implementation of Single Linked list

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int Element;
	struct node *Next;
};
struct node *List;
void insert_beg(struct node *List, int x);
void insert_mid(struct node *List, int pos, int x);
void insert_last(struct node *List, int x);
void display(struct node *List);
void delete_beg(struct node *List);
void delete_mid(struct node *List, int x);
void delete_last(struct node *List);
struct node *Find(struct node *List, int x);
int main()
{
int ch,e,pos;
List=malloc(sizeof(struct node));
List->Next=NULL;
while(1)
{
printf("\n1.Insert begin\n2.Insert Mid \n3.Inseret Last \n4.Delete Begin\n5.Delete Mid
\n6.Delete Last  \n7.Display \n8.Exit\n");
printf("\nEnter your choice\n");
scanf("%d",&ch);
switch(ch)
{
	case 1:
		printf("\nEnter an element to insert\n");
		scanf("%d",&e);
		insert_beg(List, e);
		display(List);
		break;
	case 2:
		printf("\nEnter an element to insert\n");
		scanf("%d",&e);
		printf("Enter the position to insert\n");
		scanf("%d",&pos);
		insert_mid(List, pos, e);
		display(List);
		break;
	case 3:
		printf("\nEnter an element to insert\n");
		scanf("%d",&e);
		insert_last(List, e);
		display(List);
		break;
```

```
		case 4:
			delete_beg(List);
			display(List);
			break;
		case 5:
			printf("\nEnter the element to be deleted\n");
			scanf("%d",&e);
			delete_mid(List, e);
			display(List);
			break;
		case 6:
			delete_last(List);
			display(List);
			break;
		case 7:
			display(List);
			break;
		case 8:
			exit(0);
	}
	}
	return 0;
	}

	void insert_beg(struct node *List, int x)
	{
		struct node *NewNode;
		NewNode = malloc(sizeof(struct node));
		NewNode->Element = x;
		if(List->Next == NULL)
				NewNode->Next = NULL;
		else
				NewNode->Next = List->Next;
		List->Next = NewNode;
	}
	void display(struct node *List)
	{
		if(List->Next != NULL)
		{
		struct node *Position;
		Position=List;
		printf("\nList is ");
		while(Position->Next != NULL)
		{
		Position = Position->Next;
		printf("%d ", Position->Element);
		}
		printf("\n");
		}
		else
			printf("\nEmpty List\n");
	}
```

```
void insert_mid(struct node *List, int pos, int x)
{
	struct node *NewNode, *Position;
	NewNode = malloc(sizeof(struct node));
	NewNode->Element = x;
	Position=Find(List, pos);
	NewNode->Next = Position->Next;
	Position->Next = NewNode;
}
struct node *Find(struct node *List, int x)
{
	struct node *Position;
	Position = List->Next;
	while(Position != NULL && Position->Element != x)
		Position = Position->Next;
	return Position;
}
struct node *FindPrevious(struct node *List, int x)
{
struct node *Position;
Position = List;
while(Position->Next != NULL && Position->Next->Element != x)
	Position = Position->Next;
return Position;
}

void insert_last(struct node *List, int e)
{
	struct node *NewNode, *Position;
	NewNode= malloc(sizeof(struct node));
	NewNode->Element = e;
	NewNode->Next = NULL;
	if(List->Next==NULL)
		List->Next = NewNode;
	else
	{
	Position = List;
	while(Position->Next != NULL)
		Position = Position->Next;
	Position->Next = NewNode;
	}
}
void delete_beg(struct node *List)
{
	if(List->Next != NULL)
	{
		struct node *TempNode;
		TempNode = List->Next;
		List->Next = TempNode->Next;
		printf("The deleted item is %d\n", TempNode->Element);
		free(TempNode);
	}
```

```
        else
                printf("\nEmpty List\n");
}
void delete_mid(struct node *List, int e)
{
        if(List->Next != NULL)
        {
                 struct node *TempNode, *Position;
                 Position = FindPrevious(List, e);
                 if(Position->Next != NULL)
                 {
                         TempNode = Position->Next;
                         Position->Next = TempNode->Next;
                         printf("The deleted item is %d\n", TempNode->Element);
                         free(TempNode);
                 }

        }
        else
                printf("\nEmpty List\n");
}
void delete_last(struct node *List)
{

        if(List->Next != NULL)
        {
                struct node *TempNode, *Position;
                Position = List;
                while(Position->Next->Next != NULL)
                        Position = Position->Next;
                TempNode = Position->Next;
                Position->Next = NULL;
                printf("The deleted item is %d\n", TempNode->Element);
                free(TempNode);
        }
        else
        {
                printf("\nEmpty List\n");
        }
}
```

OUTPUT:

```
1.Insert Beg
2.Insert Middle
3.Insert End
4.Delete Beg
5.Delete Middle
6.Delete End
7.Find
8.Traverse
9.Exit
Enter your choice : 1
Enter the element : 57
57
Enter your choice : 3
Enter the element : 32
57  32
Enter your choice : 1
Enter the element : 42
42  57  32
Enter your choice : 3
Enter the element : 48
42  57  32  48
Enter your choice : 4
The deleted item is 42
57  32  48
```

```
Enter your choice : 4
The deleted item is 42
57  32  48
Enter your choice : 3
Enter the element : 3
57  32  48  3
Enter your choice : 5
Enter the element : 32
The deleted item is 32
57  48  3
Enter your choice : 6
The deleted item is 3
57  48
Enter your choice : 3
Enter the element : 89
57  48  89
Enter your choice : 1
Enter the element : 86
86  57  48  89
Enter your choice : 8
```

2. IMPLEMENTATION OF DOUBLE LINKED LIST:

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
  struct node *Prev;
  int Element;
  struct node *next;
};
struct node *find(struct node *list, int x);
void insertbeg(struct node *list, int e);
void InsertLast(struct node *list, int e);
void InsertMid(struct node *list, int p, int e);
void DeleteBeg(struct node *list);
void deleteend(struct node *list);
void deletemid(struct node *list, int e);
void display(struct node *list);
```

```
int main()
{
struct node *list = malloc(sizeof(struct node));
list->Prev = NULL;
list->next = NULL;
struct node *position;
int ch, e, p;
printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");
printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");
printf("\n7.find \n8.display \n9.Exit\n");
while(1)
{
printf("Enter your choice : ");
scanf("%d", &ch);
switch(ch)
{
	case 1:
	printf("Enter the element : ");
	scanf("%d", &e);
	insertbeg(list, e);
	display(list);
	break;
	case 2:
	printf("Enter the position element : ");
	scanf("%d", &p);
	printf("Enter the element : ");
	scanf("%d", &e);
	InsertMid(list, p, e);
	display(list);
	break;
	case 3:
	printf("Enter the element : ");
	scanf("%d", &e);
	InsertLast(list, e);
	display(list);
	break;
	case 4:
	DeleteBeg(list);
	display(list);
```

```
        break;
        case 5:
        printf("Enter the element : ");
        scanf("%d", &e);
        deletemid(list, e);
        display(list);
        break;
        case 6:
        deleteend(list);
        display(list);
        break;
        case 7:
        printf("Enter the element : ");
        scanf("%d", &e);
        position = find(list, e);
        if(position != NULL)
        printf("Element found...!\n");
        else
        printf("Element not found...!\n");
        break;
        case 8:
        display(list);
        break;
    case 9:
        exit(0);
        break;
        }
        }
        return 0;
}
struct node *find(struct node *list, int x)
{
struct node *position;
position = list->next;
while(position != NULL && position->Element != x)
position = position->next;
return position;
}
void insertbeg(struct node *list, int e)
```

```
{
	struct node *newnode = malloc(sizeof(struct node));
	newnode->Element = e;
	if(list->next==NULL)
	newnode->next = NULL;
	else
	{
		newnode->next = list->next;
		newnode->next->Prev = newnode;
	}
	newnode->Prev = list;
	list->next = newnode;
}
void InsertLast(struct node *list, int e)
{
	struct node *newnode = malloc(sizeof(struct node));
	struct node *position;
	newnode->Element = e;
	newnode->next = NULL;
	if(list->next==NULL)
	{
		newnode->Prev = list;
		list->next = newnode;
	}
	else
	{
		position = list;
		while(position->next != NULL)
		position = position->next;
		position->next = newnode;
		newnode->Prev = position;
	}
}
void InsertMid(struct node *list, int p, int e)
{
	struct node *newnode = malloc(sizeof(struct node));
	struct node *position;
	position = find(list, p);
	newnode->Element = e;
```

```
        newnode->next = position->next;
        position->next->Prev = newnode;
        position->next = newnode;
        newnode->Prev = position;
}
void DeleteBeg(struct node *list)
{
if(list->next!=NULL)
{
        struct node *tempnode;
        tempnode = list->next;
        list->next = tempnode->next;
        if(list->next != NULL)
        {
                tempnode->next->Prev = list;
                printf("The deleted item is %d\n", tempnode->Element);
                free(tempnode);
        }
        else
        printf("list is empty...!\n");
}
void deleteend(struct node *list)
{
        if(list->next!=NULL)
        {
                struct node *position;
                struct node *tempnode;
                position = list;
                while(position->next != NULL)
                position = position->next;
                tempnode = position;
                position->Prev->next = NULL;
                printf("The deleted item is %d\n", tempnode->Element);
                free(tempnode);
        }
        else
        printf("list is empty...!\n");
}
void deletemid(struct node *list, int e)
```

```
{
	if(list->next!=NULL)
	{
		struct node *position;
		struct node *tempnode;
		position = find(list, e);
		if(position->next!=NULL)
		{
			tempnode = position;
			position->Prev->next = position->next;
			position->next->Prev = position->Prev;
			printf("The deleted item is %d\n", tempnode-
>Element);
			free(tempnode);
		}
	}
	else
	printf("list is empty...!\n");

}
void display(struct node *list)
{
if(list->next!=NULL)
{
	struct node *position;
	position = list;
	while(position->next!= NULL)
	{
	position = position->next;
	printf("%d\t", position->Element);
	}
	printf("\n");
	}
	else
	printf("list is empty...!\n");
}
}
```

OUTPUT:

```
1.Insert Beg
2.Insert Middle
3.Insert End
4.Delete Beg
5.Delete Middle
6.Delete End
7.Find
8.Traverse
9.Exit
Enter your choice : 1
Enter the element : 48
48
Enter your choice : 3
Enter the element : 59
48  59
Enter your choice : 1
Enter the element : 84
84  48  59
Enter your choice : 3
Enter the element : 28
84  48  59  28
Enter your choice : 4
The deleted item is 84
48  59  28
```

```
Enter your choice : 5
Enter the element : 32
The deleted item is 32
74  52  74
Enter your choice : 6
The deleted item is 74
74  52
Enter your choice : 8
74  52
```

3. **Polynomial Manipulation:(Application of singly linked list)**

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct poly
{
int coeff;
int pow;
struct poly *Next;
};
typedef struct poly Poly;
void Create(Poly *List);
void Display(Poly *List);
void Addition(Poly *Poly1, Poly *Poly2, Poly *Result);
int main()
{
Poly *Poly1 = malloc(sizeof(Poly));
Poly *Poly2 = malloc(sizeof(Poly));
Poly *Result = malloc(sizeof(Poly));
Poly1->Next = NULL;
Poly2->Next = NULL;
printf("Enter the values for first polynomial :\n");
Create(Poly1);
printf("The polynomial equation is : ");
Display(Poly1);
printf("\nEnter the values for second polynomial :\n");
Create(Poly2);
printf("The polynomial equation is : ");
Display(Poly2);
Addition(Poly1, Poly2, Result);
printf("\nThe polynomial equation addition result is : ");
Display(Result);
return 0;
}
void Create(Poly *List)
{
int choice;
```

```
Poly *Position, *NewNode;
Position = List;
do
{
NewNode = malloc(sizeof(Poly));
printf("Enter the coefficient : ");
scanf("%d", &NewNode->coeff);
printf("Enter the power : ");
scanf("%d", &NewNode->pow);
NewNode->Next = NULL;
Position->Next = NewNode;
Position = NewNode;
printf("Enter 1 to continue : ");
scanf("%d", &choice);
} while(choice == 1);
}
void Display(Poly *List)
{
Poly *Position;
Position = List->Next;
while(Position != NULL)
{
printf("%dx^%d", Position->coeff, Position->pow);
Position = Position->Next;
if(Position != NULL && Position->coeff > 0)
{
printf("+");
}
}
}
void Addition(Poly *Poly1, Poly *Poly2, Poly *Result)
{
Poly *Position;
Poly *NewNode;2

Poly1 = Poly1->Next;
Poly2 = Poly2->Next;
Result->Next = NULL;
Position = Result;
```

```
while(Poly1 != NULL && Poly2 != NULL)
{
NewNode = malloc(sizeof(Poly));
if(Poly1->pow == Poly2->pow)
{
NewNode->coeff = Poly1->coeff + Poly2->coeff;
NewNode->pow = Poly1->pow;
Poly1 = Poly1->Next;
Poly2 = Poly2->Next;
}
else if(Poly1->pow > Poly2->pow)
{
NewNode->coeff = Poly1->coeff;
NewNode->pow = Poly1->pow;
Poly1 = Poly1->Next;
}
else if(Poly1->pow < Poly2->pow)
{
NewNode->coeff = Poly2->coeff;
NewNode->pow = Poly2->pow;
Poly2 = Poly2->Next;
}
NewNode->Next = NULL;
Position->Next = NewNode;
Position = NewNode;
}
while(Poly1 != NULL || Poly2 != NULL)
{
NewNode = malloc(sizeof(Poly));
if(Poly1 != NULL)
{
NewNode->coeff = Poly1->coeff;
NewNode->pow = Poly1->pow;
Poly1 = Poly1->Next;
}
if(Poly2 != NULL)
{
NewNode->coeff = Poly2->coeff;
NewNode->pow = Poly2->pow;
```

```
Poly2 = Poly2->Next;
}
NewNode->Next = NULL;
Position->Next = NewNode;
Position = NewNode;
}
}
```

output

```
Enter the values for first polynomial :
Enter the coefficient : 2
Enter the power : 3
Enter 1 to continue : 1
Enter the coefficient : 3
Enter the power : 5
Enter 1 to continue : 1
Enter the coefficient : 54
Enter the power : 4
Enter 1 to continue : 1
Enter the coefficient : 64
Enter the power : 5
Enter 1 to continue : 1
Enter the coefficient : 654
Enter the power : 0
Enter 1 to continue : 0
The polynomial equation is : 2x^3+3x^5+54x^4+64x^5+654x^0
Enter the values for second polynomial :
Enter the coefficient : 7
Enter the power : 4
Enter 1 to continue : 47
The polynomial equation is : 7x^4
The polynomial equation addition result is : 7x^4+2x^3+3x^5+54x^4+64x^5+654x
^0
C:\Users\sujit\Desktop\data structure>7
```

SUBTRACTION:

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

struct poly {
    int coeff;
    int pow;
    struct poly *Next;
};
typedef struct poly Poly;

void Create(Poly *List);
void Display(Poly *List);
void Subtraction(Poly *Poly1, Poly *Poly2, Poly *Result);
```

```
int main() {
   Poly *Poly1 = malloc(sizeof(Poly));
   Poly *Poly2 = malloc(sizeof(Poly));
   Poly *Result = malloc(sizeof(Poly));
   Poly1->Next = NULL;
   Poly2->Next = NULL;
   printf("Enter the values for first polynomial :\n");
       Create(Poly1);
   printf("The polynomial equation is : ");
       Display(Poly1);
       printf("\nEnter the values for second polynomial :\n");
       Create(Poly2); printf("The polynomial equation is : ");
        Display(Poly2); Subtraction(Poly1, Poly2, Result);
       printf("\nThe polynomial equation subtraction result is : ");
        Display(Result);
      return 0;
      }
      void Create(Poly *List)
      {
        int choice;
        Poly *Position, *NewNode;
        Position = List;
        do
        {
            NewNode = malloc(sizeof(Poly));
            printf("Enter the coefficient : ");
            scanf("%d", &NewNode->coeff);
            printf("Enter the power : ");
            scanf("%d", &NewNode->pow);
            NewNode->Next = NULL;
            Position->Next = NewNode;
            Position = NewNode;
            printf("Enter 1 to continue : ");
            scanf("%d", &choice);
         }while(choice == 1);
       }
       void Display(Poly *List)
       {
        Poly *Position; Position = List->Next;
        while(Position != NULL)
        {
           printf("%dx^%d", Position->coeff, Position->pow);
           Position = Position->Next;
           if(Position != NULL && Position->coeff > 0)
```

```
    {
      printf("+");
    }
  }
 }
void Subtraction(Poly *Poly1, Poly *Poly2, Poly *Result)
 {
    Poly *Position;
    Poly *NewNode;
    Poly1 = Poly1->Next;
    Poly2 = Poly2->Next;
    Result->Next = NULL;
    Position = Result;
    while(Poly1 != NULL && Poly2 != NULL)
     {
        NewNode = malloc(sizeof(Poly));
        if(Poly1->pow == Poly2->pow)
        {
            NewNode->coeff = Poly1->coeff - Poly2->coeff;
            NewNode->pow = Poly1->pow;
            Poly1 = Poly1->Next;
            Poly2 = Poly2->Next;
      }
      else if(Poly1->pow > Poly2->pow)
      {
        NewNode->coeff = Poly1->coeff;
        NewNode->pow = Poly1->pow;
        Poly1 = Poly1->Next;
       }
      else if(Poly1->pow < Poly2->pow)
      {
           NewNode->coeff = -(Poly2->coeff);
           NewNode->pow = Poly2->pow;
           Poly2 = Poly2->Next;
     }
           NewNode->Next = NULL;
           Position->Next = NewNode;
           Position = NewNode;
     }
     while(Poly1 != NULL || Poly2 != NULL)
     {
       NewNode = malloc(sizeof(Poly));
       if(Poly1 != NULL)
       {
```

```
                 NewNode->coeff = Poly1->coeff;
                 NewNode->pow = Poly1->pow;
                 Poly1 = Poly1->Next;
                  }
                if(Poly2 != NULL)
                {
                    NewNode->coeff = -(Poly2->coeff);
                    NewNode->pow = Poly2->pow;
                    Poly2 = Poly2->Next;
                }
                  NewNode->Next = NULL;
                  Position->Next = NewNode;
                 Position = NewNode;
               }
```

4. **IMPLEMENTATION OF STACK USING ARRAY AND LINKED LIST IMPLEMENTATION**

PROGRAM:

```
/*stack follows last in first out principle.both deletion and insertion can be
done in one end*/
#include<stdio.h>
#include<stdlib.h>
#define size 5
int Stack[size], top = -1;
void Push(int ele);
void Pop();
void Top();
void Display();
int main()
{
 int ch, e;
while(1)
{
 printf("1.PUSH\n 2.POP\n 3.TOP\n 4.DISPLAY\n 5.EXIT ");
 printf("\nEnter your choice : ");
 scanf("%d", &ch);
switch(ch)
{
 case 1:
 printf("Enter the element : ");
 scanf("%d",&e);
 Push(e);
```

```
 break;
 case 2:
 Pop();
 break;
 case 3:
 Top();
 break;
 case 4:
 Display();
 break;
 case 5:
 exit(0);
}
}
}
void Push(int ele)
{
 if(top==size-1)
 printf("Stack Overflow\n");
 else
 {
 top = top + 1;
 Stack[top] = ele;
 }
}
void Pop()
{
 if(top == -1)
 printf("Stack Underflow\n");
 else
 {
 printf("%d\n", Stack[top]);
 top = top - 1;
 }
}
void Top()
{
 if(top == -1)
 printf("Stack Underflow\n");
 else
 printf("%d\n", Stack[top]);
```

```
}
void Display()
{
  if(top == -1)
  printf("Stack Underflow\n");
  else
  {
  for(int i = top;i>= 0; i--)
  printf("%d\t", Stack[i]);
 printf("\n");
}
```

OUTPUT:

```
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 1
Enter the element : 84
84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 1
Enter the element : 73
73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 1
Enter the element : 832
832     73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 1
Enter the element : 35
35      832     73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 2
35
832     73      84
```

```
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 1
Enter the element : 35
35      832     73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 2
35
832     73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 3
832
832     73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice : 4
832     73      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT
Enter your choice :
```

STACK USING LINKED LIST

PROGRAM:

```
/*stack follows last in first out principle.both deletion and insertion
can be done in one end in linked list*/
#include <stdio.h>
#include <stdlib.h>
struct node
{
int Element;
struct node *Next;
};
struct node *top = NULL;
void Push(int e);
void Pop();
void Top();
void Display();
int main()
{
 int ch, e;
 while(1)
 {
   printf("1.PUSH\n 2.POP\n 3.TOP\n 4.DISPLAY\n 5.EXIT\n");
   printf("\nEnter your choice : ");
   scanf("%d", &ch);
   switch(ch)
   {
     case 1:
     printf("Enter the element : ");
     scanf("%d", &e);
     Push(e);
     Display();
     break;
     case 2:
     Pop();
     Display();
     break ;
     case 3:
     Top();
     Display();
```

```
        break;
        case 4:
        Display();
        break;
        case 5:
        exit(0);
     }
  }
return 0;
}

void Push(int e)
{
struct node *NewNode = malloc(sizeof(struct node));
NewNode->Element = e;
  if(top == NULL)
  NewNode->Next = NULL;
else
  NewNode->Next = top;
  top = NewNode;
}
void Pop()
{
  if(top==NULL)
  printf("Stack is Underflow...!\n");
else
{
  struct node *tempNode;
  tempNode = top;
  top = top->Next;
  printf("%d\n", tempNode->Element);
  free(tempNode);
}
}
void Top()
{
  if(top==NULL)
  printf("Stack is Underflow...!\n");
  else
  printf("%d\n",top->Element);
}
```

```
void Display()
{
  if(top==NULL)
  printf("Stack is Underflow...!\n");
  else
  {
  struct node *position;
  position = top;
  while(position != NULL)
   {
     printf("%d\t", position->Element);
     position = position->Next;
   }
  printf("\n");
```

OUTPUT:

```
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 1
Enter the element : 84
84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 1
Enter the element : 83
83      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 1
Enter the element : 26
26      83      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 1
Enter the element : 07
7       26      83      84
```

```
Enter your choice : 2
92
7       26      83      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 2
7
26      83      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 3
26
26      83      84
1.PUSH
2.POP
3.TOP
4.DISPLAY
5.EXIT

Enter your choice : 4
26      83      84
```

IMPLEMENTATION OF QUEUE USING ARRAY AND LINKED LIST IMPLEMENTATION

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 5
int queue[size], front = -1, rear = -1;
void enqueue(int ele);
void dequeue();
void display();
int main()
{
  int ch, e;
  while(1)
  {
  printf("1.ENQUEUE\n 2.DEQUEUE\n 3.DISPLAY\n 4.EXIT\n");
  printf("\nEnter your choice : ");
  scanf("%d", &ch);
  switch(ch)
  {
    case 1:
    printf("Enter the element : ");
    scanf("%d", &e);
    enqueue();
        display();
    break;
    case 2:
    dequeue();
        display();
    break;
    case 3:
    display();
    break;
        case 4:
    exit(0);
  }
  }
}
```

```
void enqueue(int ele)
{
   if(rear ==size - 1)
   printf("Queue is Overflow\n");
   else
   {
     rear = rear + 1;
     Queue[rear] = ele;
   if(front == -1)
   front = 0;
   }
}
void dequeue()
{
    if(front == -1)
    printf("Queue is Underflow\n");
    else
    {
     printf("%d\n", Queue[front]);
     if(front == rear)
     front = rear = -1;
     else
      front = front + 1;
    }
}
void display()
{
     if(front==-1)
     printf("Queue is Underflow.\n");
     else
     {
       for(int i = front; i <= rear; i++)
       printf("%d\t", Queue[i]);
       printf("\n");
     }
}
```

OUTPUT:

```
Enter your choice : 1
Enter the element : 93
79      59      93
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT

Enter your choice : 2
79
59      93
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT

Enter your choice : 3
59      93
```

QUEUE USING LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
int Element;
struct node *next;
};
struct node *list=NULL;
struct node *front = NULL;
struct node *rear = NULL;
void enqueue(int ele);
void dequeue();
void display();
int main()
{
  int ch, e;
while(1)
{
  printf("1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.EXIT");
  printf("\nEnter your choice : ");
  scanf("%d", &ch);
```

```
  switch(ch)
  {
    case 1:
    printf("Enter the element : ");
    scanf("%d", &e);
    enqueue(e);
        display();
    break;
    case 2:
    dequeue();
        display();
    break;
    case 3:
    display();
    break;
        case 4:
    exit(0);

  }
}
return 0;
}
void enqueue(int e)
{
  struct node *newnode = malloc(sizeof(struct node));
  newnode->Element = e;
  newnode->next = NULL;
  if(rear == NULL)
  front = rear = newnode;
  else
  {
  rear->next = newnode;
  rear = newnode;
  }
}
void dequeue()
{
  if(list!=NULL)
  printf("Queue is Underflow\n");
  else
  {
   struct node *tempnode;
   tempnode = front;
```

```
    if(front == rear)
    front = rear = NULL;
    else
    front = front->next;
    printf("%d\n", tempnode->Element);
    free(tempnode);
  }
}
void display()
{
    if(list!=NULL)
    printf("Queue is Underflow\n");
    else
   {
     struct node *position;
     position = front;
     while(position != NULL)
     {
      printf("%d\t", position->Element);
      position = position->next;
     }
     printf("\n");
   }
}
```

OUTPUT:

```
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice: 1
Enter the element: 42
2   32  65  42
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice: 2
Dequeued element: 2
32  65  42
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice: 3
32  65  42
```


```
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice: 1
Enter the element: 42
2   32  65  42
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice: 2
Dequeued element: 2
32  65  42
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.EXIT
Enter your choice: 3
32  65  42
```

9.Implementation Of Binary Search Tree

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

struct Node {
int data;
struct Node* left;
struct Node* right;
};

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

struct Node* insert(struct Node* root, int value)
 {
        if (root == NULL)
{
return;
createNode(value);
}
if (value < root->data)
 {
```

```
 root->left = insert(root->left, value);
}
else if (value > root>data)
{
 root->right = insert(root>right, value);
}
return root;
}
struct Node* minValueNode(struct Node* node)
{
struct Node* current = node;
while (current && current->left != NULL) {        current = current>left;
}
return current;
}
struct Node* deleteNode(struct Node* root, int value)
{
if (root == NULL)
{
return root;
}
if (value < root->data)
 {
root->left = deleteNode(root->left, value);
}
else if (value > root>data)
{
root->right = deleteNode(root->right, value);    } else {        if (root->left ==
NULL)
{
```

```
struct Node* temp = root->right;            free(root);            return temp;
}
else if (root->right == NULL)
{
 struct Node* temp = root->left;
free(root);
return temp;
}
struct Node* temp = minValueNode(root->right);
root->data = temp>data;
root->right = deleteNode(root->right, temp->data);
}
return root;
}
struct Node* search(struct Node* root, int value)
{
if (root == NULL || root>data == value)
{
return root;
}
 if (root->data < value)
{
return search(root->right, value);
}
return search(root->left, value);
}
void display(struct Node* root)
{
if (root != NULL)
{
```

```
display(root->left);
printf("%d ", root>data);
display(root->right);
}
}
int main() {
struct Node* root = NULL;
root = insert(root, 50);
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);
 printf("Binary Search Tree Inorder Traversal: ");
display(root);
printf("\n");
root = deleteNode(root, 20);
printf("Binary Search Tree Inorder Traversal after deleting 20: ");
display(root);
printf("\n");
struct Node* searchResult = search(root, 30);
if (searchResult != NULL)
{
printf("Element 30 found in the Binary Search Tree.\n");
}
Else
{
printf("Element 30 not found in the Binary Search Tree.\n");
}
```

```
return 0;
}
```

```
Binary Search Tree Inorder Traversal: 20 30 40 50 60 70 80
Binary Search Tree Inorder Traversal after deleting 20: 30 40 50 60 70 80
Element 30 found in the Binary Search Tree.


=== Code Execution Successful ===
```

10. IMPLEMENTATION OF AVL TREE

Program:

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

typedef struct Node
{
int data;
struct Node *left;
struct Node *right;
int height;
}
Node;
 //Function to get the height of a node int height(Node *node)
 {
```

```
 if (node == NULL)
return 0;
return node->height;
}

// Function to get the balance factor of a node
int balance_factor(Node *node)
{
if (node == NULL)
return 0;
return height(node->left) - height(node->right);
}
// Function to create a new node
Node* newNode(int data) {
Node* node = (Node*)malloc(sizeof(Node));
node->data = data;
node->left = NULL;
node->right = NULL;
node->height = 1;
   return node;
}
// Function to perform a right rotation
Node* rotate_right(Node *y) {
Node *x = y->left;
Node *T2 = x->right;
// Perform rotation
 x->right = y; y->left = T2;
```

```
// Update heights
y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) :
height(y->right));
 x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) :
height(x>right));
return x;
}
// Function to perform a left rotation
Node* rotate_left(Node *x) {
Node *y = x->right;
Node *T2 = y->left;
// Perform rotation
 y->left = x;
  x->right = T2;
// Update heights
x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) :
height(x->right));
y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) :
height(y>right));
 return y;
}
// Function to insert a node into AVL tree Node* insert(Node *node,
int data) {     if (node == NULL)
return newNode(data);
 if (data < node->data)
node->left = insert(node->left, data);     else if (data > node->data)
node->right = insert(node->right, data);     else // Duplicate keys
not allowed
return node;
```

```
 // Update height of current node     node->height = 1 +
(height(node->left) > height(node->right) ? height(node->left) :
height(node->right));

// Get the balance factor    int balance = balance_factor(node);

 // Perform rotations if needed

if (balance > 1 && data < node->left->data)

return rotate_right(node);

if (balance < -1 && data > node->right->data)       return
rotate_left(node);    if (balance > 1 && data> node->left->data)

{

node->left = rotate_left(node->left);

return rotate_right(node);

}

if (balance < -1 && data < node->right->data) {       node->right =
rotate_right(node->right);       return rotate_left(node);

}

return node;

}

 // Function to find the node with minimum value

Node* minValueNode(Node *node)

 {    Node* current = node;

  while (current->left != NULL)

 current = current->left;     return current;

}

// Function to delete a node from AVL tree Node* deleteNode(Node
*root, int data)

{

if (root == NULL)

return root;
```

```
 if (data < root->data)
root->left = deleteNode(root->left, data);
else if (data > root->data)
root->right = deleteNode(root->right, data);
 else
{
if (root->left == NULL || root->right == NULL)
{
Node *temp = root->left ? root->left : root->right;
if (temp == NULL) {
temp = root;
root = NULL;
 } else
*root = *temp; // Copy the contents of the non-empty child
free(temp);
} else
{
 Node *temp = minValueNode(root->right);
 root->data = temp->data;
root->right = deleteNode(root->right, temp->data);
 }
}
if (root == NULL)
      return root;
   // Update height of current node
 root->height = 1 + (height(root->left) > height(root->right) ?
height(root->left) : height(root->right));
```

```
    // Get the balance factor
    int balance = balance_factor(root);

    // Perform rotations if needed
    if (balance > 1 && balance_factor(root->left) >= 0)
 return rotate_right(root);
    if (balance > 1 && balance_factor(root->left) < 0) {
        root->left = rotate_left(root->left);
return rotate_right(root);
    }
    if (balance < -1 && balance_factor(root->right) <= 0)
 return rotate_left(root);
    if (balance < -1 && balance_factor(root->right) > 0) {
        root->right = rotate_right(root->right);
  return rotate_left(root);
    }
return root;
}

// Function to print AVL tree inorder
 void inorder(Node *root)
 {    if (root != NULL) {
inorder(root->left);
printf("%d ", root->data);
 inorder(root->right);
}
```

```
}
int main() {
Node *root = NULL;
// Inserting nodes
root = insert(root, 10);
 root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
 root = insert(root, 25);
printf("Inorder traversal of the constructed AVL tree: ");
 inorder(root);
 printf("\n");
// Deleting node
printf("Delete node 30\n");
 root = deleteNode(root, 30);
printf("Inorder traversal after deletion: ");
 inorder(root);
printf("\n");
return 0;
}
```

```
Inorder traversal of the constructed AVL tree: 10 20 25 30 40 50
Delete node 30
Inorder traversal after deletion: 10 20 25 40 50


=== Code Execution Successful ===
```

11.IMPLEMENTATION OF BFS,DFS

Program:

```
#include <stdio.h>
#define MAX_VERTICES 10
int graph[MAX_VERTICES][MAX_VERTICES] = {0};
int visited[MAX_VERTICES] = {0}; int vertices;
void createGraph()
 {
    int i, j;
    printf("Enter the number of vertices: ");    scanf("%d",
&vertices);
   printf("Enter the adjacency matrix:\n");
for (i = 0; i < vertices; i++)
 {
   for (j = 0; j < vertices; j++)
   {
    scanf("%d", &graph[i][j]);
   }
}
  }
   void dfs(int vertex)
   {
     int i;
    printf("%d ", vertex);
    visited[vertex] = 1;
    for (i = 0; i < vertices; i++)
    {
     if (graph[vertex][i] && !visited[i])
     dfs(i);
    }
   }

int main() {
    int i;
    createGraph();
```

```
    printf("Ordering of vertices after DFS traversal:\n");
    for (i = 0; i < vertices; i++)
    {
      if (!visited[i])
        {
            dfs(i);
        }
    }
    return 0;
```

```
Graph:
Vertex 0: 2 -> 2 -> 1 -> NULL
Vertex 1: 2 -> 0 -> NULL
Vertex 2: 3 -> 0 -> 1 -> 0 -> NULL
Vertex 3: 3 -> 3 -> 2 -> NULL

BFS Traversal:
Visited 2

DFS Traversal:
Visited 2
Visited 3
Visited 0
Visited 1


=== Code Execution Successful ===
```

```
}
```

12. PERFORMING TOPOLOGICAL SORTING

Program:

```
#include <stdio.h>
#define MAX_VERTICES 10
int graph[MAX_VERTICES][MAX_VERTICES] = {0};
int visited[MAX_VERTICES] = {0}; int vertices;
void createGraph()
 {
    int i, j;
    printf("Enter the number of vertices: ");    scanf("%d",
&vertices);
```

```
  printf("Enter the adjacency matrix:\n");
for (i = 0; i < vertices; i++)
 {
  for (j = 0; j < vertices; j++)
  {
   scanf("%d", &graph[i][j]);
  }
}
 }
  void dfs(int vertex)
  {
    int i;
   printf("%d ", vertex);
   visited[vertex] = 1;
   for (i = 0; i < vertices; i++)
   {
    if (graph[vertex][i] && !visited[i])
    dfs(i);
   }
   }

int main() {
   int i;
   createGraph();
   printf("Ordering of vertices after DFS traversal:\n");
   for (i = 0; i < vertices; i++)
   {
     if (!visited[i])
      {
          dfs(i);
     }
   }
   return 0;
}
```

```
Enter the number of vertices: 2
Enter the adjacency matrix:
26 26 26 2 6 26
Ordering of vertices after DFS traversal:
0 1

=== Code Execution Successful ===
```

13. IMPLEMENTATION OF PRIM'S ALGORITHM

Program:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 10
#define INF 999999
int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;
void createGraph()
 {
   int i, j;
   printf("Enter the number of vertices: ");
   scanf("%d", &vertices);
   printf("Enter the adjacency matrix:\n");
   for (i = 0; i < vertices; i++)
   {
```

```
        for (j = 0; j < vertices; j++)
          scanf("%d", &graph[i][j]);
    }
}
int findMinKey(int key[], bool mstSet[])
 {
     int min = INF, min_index;
     for (int v = 0; v < vertices; v++)
     {
          if (mstSet[v] == false && key[v] < min)
           {
          min = key[v];
          min_index = v;
           }
     }
     return min_index;
}

void printMST(int parent[])
{
   printf("Edge \tWeight\n");
   for (int i = 1; i < vertices; i++
   {
       printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}
```

```
void primMST()
 {
      int parent[vertices];
      int key[vertices];
   bool mstSet[vertices];
   for (int i = 0; i < vertices; i++)
   {
      key[i] = INF;
      mstSet[i] = false;
   }
    key[0] = 0;
   // Make key 0 so that this vertex is picked as the first vertex
   parent[0] = -1;
  // First node is always root of MST
  for (int count = 0; count < vertices - 1; count++)
  {
    int u = findMinKey(key, mstSet);
    mstSet[u] = true;
    for (int v = 0; v < vertices; v++)
    {
     if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
     {
         parent[v] = u;
         key[v] = graph[u][v];
      }
    }
  }
```

```
    printMST(parent);
  }
 int main()
 {
    createGraph();
    primMST();
    return 0;
}
```

```
Enter the adjacency matrix:

0 2 0 6 0

2 0 3 8 5

0 3 0 0 7

6 8 0 0 9

0 5 7 9 0


Enter the adjacency matrix:
Edge    Weight
0 - 1   0
-128 - 2    6
-1 - 3  0
Segmentation fault


=== Code Exited With Errors ===
```

14.IMPLEMENTATION OF DIJIKSTRA'S ALGORITHM

Program:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 10
#define INF 999999
```

```
int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;
void createGraph()
{
   int i, j;
   printf("Enter the number of vertices: ");    scanf("%d",
&vertices);
   printf("Enter the adjacency matrix:\n");
   for (i = 0; i < vertices; i++)
   {
      for (j = 0; j < vertices; j++)
     scanf("%d", &graph[i][j]);
   }
}
int minDistance(int dist[], bool sptSet[])
 {
int min = INF,
min_index;
 for (int v = 0; v < vertices; v++)
 {
  if (sptSet[v] == false && dist[v] <= min)
  {
     min = dist[v];
     min_index = v;
   }
  }
  return min_index;
```

```
}
void printSolution(int dist[])
{
  printf("Vertex \t Distance from Source\n");
  for (int i = 0; i < vertices; i++)
  printf("%d \t %d\n", i, dist[i]);
 }
void dijkstra(int src)
{
    int dist[vertices];
    bool sptSet[vertices];
    for (int i = 0; i < vertices; i++)
    {
       dist[i] = INF;
      sptSet[i] = false;
    }
       dist[src] = 0;
  for (int count = 0; count < vertices - 1; count++)
  {
      int u = minDistance(dist, sptSet);
      sptSet[u] = true;
      for (int v = 0; v < vertices; v++)
      {
         if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
      }
```

```
  }
  printSolution(dist);
 }
int main()
{
   createGraph();
   int source;
     printf("Enter the source vertex: ");
     scanf("%d", &source);
     dijkstra(source);
       return 0;
```

```
Enter the adjacency matrix:

0 10 0 5 0

0 0 1 2 0

0 0 0 0 4

0 3 9 0 2

7 0 6 0 0


Enter the adjacency matrix:
Enter the source vertex: Vertex      Distance from Source
0    0
1    999999
2    999999
3    999999
4    999999


=== Code Execution Successful ===
```

```
}
```

15.PROGRAM TO PERFORM SORTING

Program:

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

void swap(int *a,
int *b) {     int temp
= *a;
*a = *b;
*b = temp;
}

int partition(int arr[], int low, int
high) {     int pivot = arr[high];
int i = (low - 1);
for (int j = low; j <= high - 1; j++)
 {
if (arr[j] < pivot)
 {
i++;
swap(&arr[i], &arr[j]);
}
 }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low,
int high)
{
if (low < high) {
int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
    }
}

void merge(int arr[], int l, int
m, int r) {     int i, j, k;     int n1
= m - l + 1;
```

```
int n2 = r - m;
int L[n1], R[n2];

for (i = 0; i < n1;
i++)        L[i] = arr[l
+ i];
for (j = 0; j < n2;
j++)        R[j] =
arr[m + 1 + j];
     i
=0;
j = 0;
    k = l;
while (i < n1 && j <
n2) {        if (L[i] <=
R[j])
 {
arr[k] = L[i];
i++;
 } else {
arr[k] = R[j];
j++;
}
 k++;
}

    while (i <
n1) {
arr[k] = L[i];
i++;
 k++;
    }
    while (j <
n2)
{
arr[k] = R[j];
j++;
k++;
    }
}
```

```
void mergeSort(int arr[], int
l, int r)
{
if (l < r) {
int m = l + (r - l) / 2;
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
   int arr[n];
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

printf("\nSorting using Quick
Sort:\n");    quickSort(arr, 0, n -
1);
 for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
   }

   printf("\n\nSorting using Merge
Sort:\n");    mergeSort(arr, 0, n -
1);
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
   }

   return 0;
}
```

```
Enter the number of elements: 5
Enter 5 elements:
36 58 14 85 69

Sorting using Quick Sort:
14 36 58 69 85

Sorting using Merge Sort:
14 36 58 69 85

=== Code Execution Successful ===
```

16.IMPLEMENTATION OF COLLISION RESOLUTION TECHNIQUES

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TABLE_SIZE 10
typedef struct Node
{
  int data;
  struct Node* next;
}
Node;
Node* createNode(int data)
```

```
{
   Node* newNode = (Node*)malloc(sizeof(Node));
   if (newNode == NULL)
   {
      printf("Memory allocation failed!\n");
      exit(1);
   }
   newNode->data = data;     newNode->next = NULL;
   return newNode;
}
int hashFunction(int key)
{
   return key % TABLE_SIZE;
}
Node* insertOpenAddressing(Node* table[], int key)
 {
       int index = hashFunction(key);
      while (table[index] != NULL)
            index = (index + 1) % TABLE_SIZE;
      table[index] = createNode(key);     return table[index];
}
void displayHashTable(Node* table[])
{
      printf("Hash Table:\n");
      for (int i = 0; i < TABLE_SIZE; i++)
      {
          printf("%d: ", i);
          Node* current = table[i];
```

```
            while (current != NULL)
            {
              printf("%d ", current->data);
              current = current->next;
            }
           printf("\n");
        }
 }

Node* insertClosedAddressing(Node* table[], int key)
 {
       int index = hashFunction(key);
       if (table[index] == NULL)
      {
          table[index] = createNode(key);
      }
     else
     {
       Node* newNode = createNode(key);
       newNode->next = table[index];
       table[index] = newNode;
     }
      return table[index];
  }
  int rehashFunction(int key, int attempt)
 {
   // Double Hashing Technique
   return (hashFunction(key) + attempt * (7 - (key % 7))) %
TABLE_SIZE;
```

```
 }
Node* insertRehashing(Node* table[], int key)
{
    int index = hashFunction(key);
    int attempt = 0;
    while (table[index] != NULL)
    {
        attempt++;
        index = rehashFunction(key, attempt);
    }
    table[index] = createNode(key);
    return table[index];
}
int main()
 {
    Node* openAddressingTable[TABLE_SIZE] = {NULL};
    Node* closedAddressingTable[TABLE_SIZE] = {NULL};
    Node* rehashingTable[TABLE_SIZE] = {NULL};
    // Insert elements into hash tables
     insertOpenAddressing(openAddressingTable, 10);
     insertOpenAddressing(openAddressingTable, 20);
     insertOpenAddressing(openAddressingTable, 5);
     insertClosedAddressing(closedAddressingTable, 10);
     insertClosedAddressing(closedAddressingTable, 20);
    insertClosedAddressing(closedAddressingTable, 5);
    insertRehashing(rehashingTable, 10);
    insertRehashing(rehashingTable, 20);
    insertRehashing(rehashingTable, 5);
```

```
  // Display hash tables
  displayHashTable(openAddressingTable);
  displayHashTable(closedAddressingTable);
 displayHashTable(rehashingTable);
 return 0;
}
```

```
Hash Table:
0:
1:
2:
3:
4:
5: 5
6:
7:
8:
9:
Hash Table:
0: 20 10
1:
2:
3:
4:
5: 5
6:
7:
8:
9:
Hash Table:
0: 10
1: 20
2:
3:
4:
5: 5
6:
7:
8:
9:


=== Code Execution Successful ===
```