

CS25311 DATA STRUCTURES LABORATORY

Moderately Shortened Practical Manual

Heading | Aim | Algorithm | Program | Output | Result

Programs are shortened only where the original manual contains comments, repetitive interface code, or unnecessary formatting. The operations required by each heading are retained.

1.A ARRAY IMPLEMENTATION OF LIST ADT

AIM

To implement the List ADT using an array and perform create, insert, delete, search and display operations.

ALGORITHM

1. Start and initialize the array list.
2. Read the number of elements and their values.
3. For insertion, shift elements to the right and place the new item.
4. For deletion, remove the selected item by shifting elements left.
5. Search the list and display the elements.
6. Stop.

PROGRAM

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

int a[MAX], n;

void create()
{
    int i;
    scanf("%d", &n);
    for(i=0;i<n;i++) scanf("%d", &a[i]);
}

void insert()
{
    int pos,x,i;
    scanf("%d%d", &pos, &x);
    if(pos<0 || pos>n || n==MAX) return;
    for(i=n;i>pos;i--) a[i]=a[i-1];
    a[pos]=x; n++;
}

void deleteItem()
{
    int pos,i;
    scanf("%d", &pos);
    if(pos<0 || pos>=n) return;
    for(i=pos;i<n-1;i++) a[i]=a[i+1];
    n--;
}

void search()
{
    int x,i,found=0;
    scanf("%d", &x);
    for(i=0;i<n;i++)
        if(a[i]==x) { printf("Found at position %d\\n",i); found=1; }
    if(!found) printf("Not found\\n");
}

void display()
{
    int i;
    for(i=0;i<n;i++) printf("%d ",a[i]);
    printf("\\n");
}

int main()
{
    int ch;
    create();
    do
    {
        printf("1.Insert 2.Delete 3.Search 4.Display 5.Exit\\n");
        scanf("%d",&ch);
    }
```

```

        if(ch==1) insert();
        else if(ch==2) deleteItem();
        else if(ch==3) search();
        else if(ch==4) display();
    }while(ch!=5);
    return 0;
}

```

OUTPUT

Input: 3

10 20 30

1

1 15

4

2

4

5

Output:

10 15 20 30

10 20 30

RESULT

Thus the array implementation of List ADT and its operations were executed successfully.

1.B DESIGN AND IMPLEMENT SINGLY LINKED LIST

AIM

To design and implement a singly linked list with insertion, deletion and display operations.

ALGORITHM

1. Create a node with data and next pointer and set head to NULL.
2. Insert a node at the beginning, end or required position.
3. Delete a node from the beginning, end or required position.
4. Reconnect the links after each deletion.
5. Traverse from head to display the list.
6. Repeat operations as required and stop.

PROGRAM

```

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

struct Node { int data; struct Node *next; } *head=NULL;

struct Node *create(int x)
{
    struct Node *p=malloc(sizeof(struct Node));
    p->data=x; p->next=NULL;
    return p;
}

void insertBegin(int x)
{
    struct Node *p=create(x);
    p->next=head; head=p;
}

void insertEnd(int x)
{
    struct Node *p=create(x), *t=head;
    if(!head){ head=p; return; }
    while(t->next) t=t->next;
    t->next=p;
}

```

```

void insertPos(int x,int pos)
{
    int i;
    struct Node *p=create(x),*t=head;
    if(pos==1){ p->next=head; head=p; return; }
    for(i=1;i<pos-1 && t;i++) t=t->next;
    if(!t){ free(p); return; }
    p->next=t->next; t->next=p;
}

void deleteBegin()
{
    struct Node *t;
    if(!head) return;
    t=head; head=head->next; free(t);
}

void deleteEnd()
{
    struct Node *t=head,*p=NULL;
    if(!head) return;
    while(t->next){ p=t; t=t->next; }
    if(p) p->next=NULL; else head=NULL;
    free(t);
}

void deletePos(int pos)
{
    int i; struct Node *t=head,*p;
    if(!head) return;
    if(pos==1){ deleteBegin(); return; }
    for(i=1;i<pos-1 && t;i++) t=t->next;
    if(!t || !t->next) return;
    p=t->next; t->next=p->next; free(p);
}

void display()
{
    struct Node *t=head;
    while(t){ printf("%d -> ",t->data); t=t->next; }
    printf("NULL\\n");
}

int main()
{
    int ch,x,pos;
    do
    {
        printf("1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete
Position 7.Display 8.Exit\\n");
        scanf("%d",&ch);
        if(ch==1){ scanf("%d",&x); insertBegin(x); }
        else if(ch==2){ scanf("%d",&x); insertEnd(x); }
        else if(ch==3){ scanf("%d%d",&x,&pos); insertPos(x,pos); }
        else if(ch==4) deleteBegin();
        else if(ch==5) deleteEnd();
        else if(ch==6){ scanf("%d",&pos); deletePos(pos); }
        else if(ch==7) display();
    }while(ch!=8);
    return 0;
}

```

OUTPUT

Input: 1 10

1 20

2 30

3 25 2

7

8

Output:

10 -> 25 -> 20 -> 30 -> NULL

RESULT

Thus the singly linked list and its operations were executed successfully.

1.C DESIGN AND IMPLEMENT DOUBLY LINKED LIST

AIM

To design and implement a doubly linked list with insertion, deletion and forward/backward display operations.

ALGORITHM

1. Create nodes containing data, prev and next pointers.
2. Initialize head as NULL.
3. Insert at the beginning, end or a required position by updating both links.
4. Delete the selected node and reconnect its neighbours.
5. Traverse in forward and backward directions.
6. Stop.

PROGRAM

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

struct Node
{
    int data;
    struct Node *prev, *next;
} *head=NULL;

struct Node *create(int x)
{
    struct Node *p=malloc(sizeof(struct Node));
    p->data=x; p->prev=p->next=NULL;
    return p;
}

void insertBegin(int x)
{
    struct Node *p=create(x);
    p->next=head;
    if(head) head->prev=p;
    head=p;
}

void insertEnd(int x)
{
    struct Node *p=create(x), *t=head;
    if(!head){ head=p; return; }
    while(t->next) t=t->next;
    t->next=p; p->prev=t;
}

void insertPos(int x,int pos)
{
    int i;
    struct Node *p=create(x), *t=head;
    if(pos==1){ insertBegin(x); free(p); return; }
    for(i=1; i<pos-1 && t; i++) t=t->next;
    if(!t){ free(p); return; }
    p->next=t->next; p->prev=t;
    if(t->next) t->next->prev=p;
    t->next=p;
}

void deletePos(int pos)
{
    int i; struct Node *t=head;
    if(!head) return;
    for(i=1; i<pos && t; i++) t=t->next;
    if(!t) return;
```

```

        if(t->prev) t->prev->next=t->next; else head=t->next;
        if(t->next) t->next->prev=t->prev;
        free(t);
    }

void display()
{
    struct Node *t=head,*last=NULL;
    printf("Forward: ");
    while(t){ printf("%d ",t->data); last=t; t=t->next; }
    printf("\n\nBackward: ");
    while(last){ printf("%d ",last->data); last=last->prev; }
    printf("\n\n");
}

int main()
{
    int ch,x,pos;
    do
    {
        printf("1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Position 5.Display 6.Exit\n\n");
        scanf("%d",&ch);
        if(ch==1){ scanf("%d",&x); insertBegin(x); }
        else if(ch==2){ scanf("%d",&x); insertEnd(x); }
        else if(ch==3){ scanf("%d%d",&x,&pos); insertPos(x,pos); }
        else if(ch==4){ scanf("%d",&pos); deletePos(pos); }
        else if(ch==5) display();
    }while(ch!=6);
    return 0;
}

```

OUTPUT

```

Input: 1 10
2 20
2 30
3 15 2
5
Output:
Forward: 10 15 20 30
Backward: 30 20 15 10

```

RESULT

Thus the doubly linked list and its operations were executed successfully.

1.D DESIGN AND IMPLEMENT CIRCULARLY LINKED LIST

AIM

To design and implement a circularly linked list with insertion, deletion, count and display operations.

ALGORITHM

1. Create a node with data and next pointer and initialize head as NULL.
2. For the first node, make next point to head itself.
3. Insert at beginning, end or position while preserving circular links.
4. Delete the required node and reconnect the list to head.
5. Count and display nodes until traversal reaches head again.
6. Stop.

PROGRAM

```

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

struct Node { int data; struct Node *next; } *head=NULL;

struct Node *create(int x)

```

```

{
    struct Node *p=malloc(sizeof(struct Node));
    p->data=x; p->next=NULL;
    return p;
}

int count()
{
    int c=0; struct Node *t=head;
    if(!head) return 0;
    do{ c++; t=t->next; }while(t!=head);
    return c;
}

void insertBegin(int x)
{
    struct Node *p=create(x),*t=head;
    if(!head){ head=p; p->next=head; return; }
    while(t->next!=head) t=t->next;
    p->next=head; t->next=p; head=p;
}

void insertEnd(int x)
{
    struct Node *p=create(x),*t=head;
    if(!head){ head=p; p->next=head; return; }
    while(t->next!=head) t=t->next;
    t->next=p; p->next=head;
}

void insertPos(int x,int pos)
{
    int i;
    struct Node *p=create(x),*t=head;
    if(pos==1){ insertBegin(x); free(p); return; }
    for(i=1;i<pos-1 && t->next!=head;i++) t=t->next;
    if(i!=pos-1){ free(p); return; }
    p->next=t->next; t->next=p;
}

void deleteBegin()
{
    struct Node *t=head,*last=head;
    if(!head) return;
    if(head->next==head){ free(head); head=NULL; return; }
    while(last->next!=head) last=last->next;
    head=head->next; last->next=head; free(t);
}

void deleteEnd()
{
    struct Node *t=head,*p=NULL;
    if(!head) return;
    if(head->next==head){ free(head); head=NULL; return; }
    while(t->next!=head){ p=t; t=t->next; }
    p->next=head; free(t);
}

void deletePos(int pos)
{
    int i; struct Node *t=head,*p;
    if(!head) return;
    if(pos==1){ deleteBegin(); return; }
    for(i=1;i<pos-1 && t->next!=head;i++) t=t->next;
    if(i!=pos-1) return;
    p=t->next; t->next=p->next; free(p);
}

void display()
{
    struct Node *t=head;
    if(!head) return;
    do{ printf("%d -> ",t->data); t=t->next; }while(t!=head);

```

```

    printf("HEAD\\n");
}

int main()
{
    int ch,x,pos;
    do
    {
        printf("1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete
Position 7.Count 8.Display 9.Exit\\n");
        scanf("%d",&ch);
        if(ch==1){ scanf("%d",&x); insertBegin(x); }
        else if(ch==2){ scanf("%d",&x); insertEnd(x); }
        else if(ch==3){ scanf("%d%d",&x,&pos); insertPos(x,pos); }
        else if(ch==4) deleteBegin();
        else if(ch==5) deleteEnd();
        else if(ch==6){ scanf("%d",&pos); deletePos(pos); }
        else if(ch==7) printf("Count = %d\\n",count());
        else if(ch==8) display();
    }while(ch!=9);
    return 0;
}

```

OUTPUT

```

Input: 1 10
2 20
2 30
8
7
5
8
9
Output:
10 -> 20 -> 30 -> HEAD
Count = 3
20 -> 30 -> HEAD

```

RESULT

Thus the circularly linked list and its operations were executed successfully.

2.A DESIGN AND IMPLEMENT STACK AND ITS OPERATIONS USING LIST

AIM

To design and implement stack operations using a linked list.

ALGORITHM

1. Initialize top as NULL.
2. Push: create a node and insert it at top.
3. Pop: remove the top node if the stack is not empty.
4. Peek: display the top element.
5. Display all elements from top to bottom and check whether the stack is empty.
6. Stop.

PROGRAM

```

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

struct Node { int data; struct Node *next; } *top=NULL;

void push(int x)
{
    struct Node *p=malloc(sizeof(struct Node));
    p->data=x; p->next=top; top=p;
}

```

```

void pop()
{
    struct Node *p;
    if(!top){ printf("Stack Underflow\\n"); return; }
    p=top; printf("Popped: %d\\n",p->data);
    top=top->next; free(p);
}

void peek(){ if(top) printf("Top = %d\\n",top->data); else printf("Stack is empty\\n"); }

void display()
{
    struct Node *p=top;
    while(p){ printf("%d ",p->data); p=p->next; }
    printf("\\n");
}

int main()
{
    int ch,x;
    do
    {
        printf("1.Push 2.Pop 3.Peek 4.Display 5.IsEmpty 6.Exit\\n");
        scanf("%d",&ch);
        if(ch==1){ scanf("%d",&x); push(x); }
        else if(ch==2) pop();
        else if(ch==3) peek();
        else if(ch==4) display();
        else if(ch==5) printf(top?"Not empty\\n":"Empty\\n");
    }while(ch!=6);
    return 0;
}

```

OUTPUT

Input: 1 10

1 20

1 30

4

3

2

4

6

Output:

30 20 10

Top = 30

Popped: 30

20 10

RESULT

Thus the stack and its operations using a list were executed successfully.

2.B DESIGN AND IMPLEMENT QUEUE AND ITS OPERATIONS USING LIST

AIM

To design and implement queue operations using a linked list.

ALGORITHM

1. Initialize FRONT and REAR as NULL.
2. Enqueue: create a node and insert it at REAR.
3. Dequeue: remove the node at FRONT and update REAR if needed.
4. Peek: display the FRONT element.
5. Display the queue from FRONT to REAR.
6. Stop.

PROGRAM

```

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

struct Node { int data; struct Node *next; } *front=NULL,*rear=NULL;

void enqueue(int x)
{
    struct Node *p=malloc(sizeof(struct Node));
    p->data=x; p->next=NULL;
    if(!rear) front=rear=p;
    else{ rear->next=p; rear=p; }
}

void dequeue()
{
    struct Node *p;
    if(!front){ printf("Queue Underflow\\n"); return; }
    p=front; printf("Deleted: %d\\n",p->data);
    front=front->next;
    if(!front) rear=NULL;
    free(p);
}

void peek(){ if(front) printf("Front = %d\\n",front->data); else printf("Queue is empty\\n"); }

void display()
{
    struct Node *p=front;
    while(p){ printf("%d ",p->data); p=p->next; }
    printf("\\n");
}

int main()
{
    int ch,x;
    do
    {
        printf("1.Enqueue 2.Dequeue 3.Peek 4.Display 5.Exit\\n");
        scanf("%d",&ch);
        if(ch==1){ scanf("%d",&x); enqueue(x); }
        else if(ch==2) dequeue();
        else if(ch==3) peek();
        else if(ch==4) display();
    }while(ch!=5);
    return 0;
}

```

OUTPUT

Input: 1 10

1 20

1 30

4

3

2

4

5

Output:

10 20 30

Front = 10

Deleted: 10

20 30

RESULT

Thus the queue and its operations using a list were executed successfully.

3.A DESIGN AND IMPLEMENT POLYNOMIAL ADT USING LIST

AIM

To design and implement polynomial ADT using a linked list and perform polynomial addition.

ALGORITHM

1. Create linked-list nodes for coefficient and exponent.
2. Insert terms in descending exponent order and combine like terms.
3. Read and display the two polynomials.
4. Compare exponents and add coefficients when they match.
5. Append remaining terms and display the sum.
6. Stop.

PROGRAM

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

struct Node
{
    int coeff,exp;
    struct Node *next;
};

struct Node *insert(struct Node *h,int c,int e)
{
    struct Node *p=malloc(sizeof(struct Node)), *t=h, *q=NULL;
    p->coeff=c; p->exp=e;
    if(!h || e>h->exp){ p->next=h; return p; }
    while(t && t->exp>e){ q=t; t=t->next; }
    if(t && t->exp==e){ t->coeff+=c; free(p); return h; }
    p->next=t; q->next=p;
    return h;
}

struct Node *readPoly()
{
    struct Node *h=NULL;
    int n,c,e,i;
    scanf("%d",&n);
    for(i=0;i<n;i++){ scanf("%d%d",&c,&e); h=insert(h,c,e); }
    return h;
}

struct Node *add(struct Node *a,struct Node *b)
{
    struct Node *r=NULL;
    while(a && b)
    {
        if(a->exp>b->exp){ r=insert(r,a->coeff,a->exp); a=a->next; }
        else if(b->exp>a->exp){ r=insert(r,b->coeff,b->exp); b=b->next; }
        else{ r=insert(r,a->coeff+b->coeff,a->exp); a=a->next; b=b->next; }
    }
    while(a){ r=insert(r,a->coeff,a->exp); a=a->next; }
    while(b){ r=insert(r,b->coeff,b->exp); b=b->next; }
    return r;
}

void display(struct Node *p)
{
    while(p){ printf("%dx^%d",p->coeff,p->exp); if(p->next) printf(" + "); p=p->next; }
    printf("\n");
}

int main()
{
    struct Node *p1=readPoly(), *p2=readPoly();
    struct Node *sum=add(p1,p2);
    display(p1); display(p2); display(sum);
    return 0;
}
```

OUTPUT

Input:

```

3
2 3
3 2
4 1
2
5 2
2 0

```

Output:

```

2x^3 + 8x^2 + 4x^1
5x^2 + 2x^0
2x^3 + 13x^2 + 4x^1 + 2x^0

```

RESULT

Thus the polynomial ADT using list and polynomial addition were executed successfully.

3.B USE STACK OPERATIONS TO CONVERT INFIX EXPRESSION INTO POSTFIX

AIM

To convert an infix expression into postfix expression using stack operations.

ALGORITHM

1. Scan the infix expression from left to right.
2. Append operands directly to the postfix expression.
3. Push opening brackets onto the stack.
4. For a closing bracket, pop until the opening bracket is found.
5. For operators, pop operators of higher or equal precedence, then push the current operator.
6. Pop remaining operators and display the postfix expression.

PROGRAM

```

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

char stack[100];
int top=-1;

int prec(char c)
{
    if(c=='^') return 3;
    if(c=='*' || c=='/') return 2;
    if(c=='+' || c=='-') return 1;
    return 0;
}

int main()
{
    char in[100], out[100], c;
    int i=0, j=0;
    scanf("%s", in);

    while((c=in[i++])!='\\0')
    {
        if(isalnum(c)) out[j++]=c;
        else if(c=='(') stack[++top]=c;
        else if(c==')')
        {
            while(top>=0 && stack[top]!='(') out[j++]=stack[top--];
            if(top>=0) top--;
        }
        else
        {
            while(top>=0 && stack[top]!='(' && prec(stack[top])>=prec(c))
                out[j++]=stack[top--];
            stack[++top]=c;
        }
    }
}

```

```

while(top>=0) out[j++]=stack[top--];
out[j]='\0';
printf("Postfix: %s",out);
return 0;
}

```

OUTPUT

Input: A+B*(C-D)

Output:
Postfix: ABCD-*+

RESULT

Thus the infix expression was converted to postfix expression successfully.

3.C USE STACK OPERATIONS FOR EVALUATING THE POSTFIX EXPRESSION

AIM

To evaluate a postfix expression using stack operations.

ALGORITHM

1. Scan the postfix expression from left to right.
2. Push operands onto the stack.
3. When an operator is found, pop the top two operands.
4. Perform the operation and push the result.
5. After all symbols are processed, pop the final result.
6. Display the result and stop.

PROGRAM

```

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

int stack[100],top=-1;

int main()
{
    char s[100],c;
    int i,a,b;
    scanf("%s",s);

    for(i=0;s[i];i++)
    {
        c=s[i];
        if(isdigit(c)) stack[++top]=c-'0';
        else
        {
            b=stack[top--]; a=stack[top--];
            if(c=='+') stack[++top]=a+b;
            else if(c=='-') stack[++top]=a-b;
            else if(c=='*') stack[++top]=a*b;
            else if(c=='/') stack[++top]=a/b;
            else if(c=='%') stack[++top]=a%b;
        }
    }
    printf("Result = %d",stack[top]);
    return 0;
}

```

OUTPUT

Input: 23*54*+9-

Output:
Result = 17

RESULT

Thus the postfix expression was evaluated successfully using stack operations.

4.A DESIGN AND IMPLEMENT BINARY SEARCH TREE

AIM

To design and implement a Binary Search Tree and display its structure.

ALGORITHM

1. Start with an empty BST.
2. Read the values and insert smaller values to the left and larger values to the right.
3. Find the height of the tree.
4. Visit each level from root to the last level.
5. Display the tree level by level.
6. Stop.

PROGRAM

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

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

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

struct Node *insert(struct Node *r,int x)
{
    if(!r) return newNode(x);
    if(x<r->data) r->left=insert(r->left,x);
    else if(x>r->data) r->right=insert(r->right,x);
    return r;
}

int height(struct Node *r)
{
    int l,h;
    if(!r) return 0;
    l=height(r->left); h=height(r->right);
    return (l>h?l:h)+1;
}

void printLevel(struct Node *r,int level)
{
    if(level==1) printf("Level 1: ",r->data);
    else
    {
        if(r){ printLevel(r->left,level-1); printLevel(r->right,level-1); }
        else { printLevel(NULL,level-1); printLevel(NULL,level-1); }
    }
}

int main()
{
    struct Node *root=NULL;
    int n,x,i,h;
    scanf("%d",&n);
    for(i=0;i<n;i++){ scanf("%d",&x); root=insert(root,x); }
    h=height(root);
    for(i=1;i<=h;i++){ printLevel(root,i); printf("\n"); }
```

```

    return 0;
}

```

OUTPUT

Input: 7
50 30 70 20 40 60 80

Output:
50
30 70
20 40 60 80

RESULT

Thus the Binary Search Tree was designed and implemented successfully.

4.B TRAVERSE THE BINARY TREE RECURSIVELY IN PREORDER, POSTORDER AND INORDER

AIM

To traverse the binary tree recursively in preorder, inorder and postorder.

ALGORITHM

1. Create the BST using the given values.
2. Preorder: visit root, left subtree and right subtree.
3. Inorder: visit left subtree, root and right subtree.
4. Postorder: visit left subtree, right subtree and root.
5. Display all three traversals.
6. Stop.

PROGRAM

```

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

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

struct Node *insert(struct Node *r,int x)
{
    if(!r)
    {
        r=malloc(sizeof(struct Node));
        r->data=x; r->left=r->right=NULL;
        return r;
    }
    if(x<r->data) r->left=insert(r->left,x);
    else if(x>r->data) r->right=insert(r->right,x);
    return r;
}

void preorder(struct Node *r){ if(r){ printf("%d ",r->data); preorder(r->left); preorder(r->right); } }
void inorder(struct Node *r){ if(r){ inorder(r->left); printf("%d ",r->data); inorder(r->right); } }
void postorder(struct Node *r){ if(r){ postorder(r->left); postorder(r->right); printf("%d ",r->data); } }

int main()
{
    struct Node *root=NULL;
    int n,x,i;
    scanf("%d",&n);
    for(i=0;i<n;i++){ scanf("%d",&x); root=insert(root,x); }
    printf("Preorder: "); preorder(root);
    printf("\nInorder: "); inorder(root);
    printf("\nPostorder: "); postorder(root);
    return 0;
}

```

OUTPUT

Input: 5
50 30 70 20 40

Output:
Preorder: 50 30 20 40 70
Inorder: 20 30 40 50 70
Postorder: 20 40 30 70 50

RESULT

Thus the binary tree traversals were implemented successfully.

5.A DESIGN AND IMPLEMENT AVL TREE

AIM

To design and implement an AVL tree using insertion, deletion, rotations and traversal.

ALGORITHM

1. Insert values using BST insertion.
2. Update height and calculate balance factor after insertion.
3. Apply LL, RR, LR or RL rotation when necessary.
4. For deletion, remove the selected node and update heights.
5. Rebalance the affected nodes after deletion.
6. Display the tree using a traversal.

PROGRAM

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

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

int height(struct Node *p){ return p?p->h:0; }
int max(int a,int b){ return a>b?a:b; }

struct Node *newNode(int x)
{
    struct Node *p=malloc(sizeof(struct Node));
    p->data=x; p->h=1; p->left=p->right=NULL;
    return p;
}

struct Node *rightRotate(struct Node *y)
{
    struct Node *x=y->left,*t=x->right;
    x->right=y; y->left=t;
    y->h=1+max(height(y->left),height(y->right));
    x->h=1+max(height(x->left),height(x->right));
    return x;
}

struct Node *leftRotate(struct Node *x)
{
    struct Node *y=x->right,*t=y->left;
    y->left=x; x->right=t;
    x->h=1+max(height(x->left),height(x->right));
    y->h=1+max(height(y->left),height(y->right));
    return y;
}

int balance(struct Node *p){ return p?height(p->left)-height(p->right):0; }

struct Node *insert(struct Node *r,int x)
{
    int b;
    if(!r) return newNode(x);
    if(x<r->data) r->left=insert(r->left,x);
    else if(x>r->data) r->right=insert(r->right,x);
```

```

    else return r;
    r->h=1+max(height(r->left),height(r->right));
    b=balance(r);
    if(b>1 && x<r->left->data) return rightRotate(r);
    if(b<-1 && x>r->right->data) return leftRotate(r);
    if(b>1 && x>r->left->data){ r->left=leftRotate(r->left); return rightRotate(r); }
    if(b<-1 && x<r->right->data){ r->right=rightRotate(r->right); return leftRotate(r); }
    return r;
}

struct Node *minNode(struct Node *r)
{
    while(r->left) r=r->left;
    return r;
}

struct Node *deleteNode(struct Node *r,int x)
{
    struct Node *t;
    int b;
    if(!r) return r;
    if(x<r->data) r->left=deleteNode(r->left,x);
    else if(x>r->data) r->right=deleteNode(r->right,x);
    else
    {
        if(!r->left || !r->right)
        {
            t=r->left?r->left:r->right;
            if(!t){ free(r); return NULL; }
            *r=*t; free(t);
        }
        else
        {
            t=minNode(r->right);
            r->data=t->data;
            r->right=deleteNode(r->right,t->data);
        }
    }
    r->h=1+max(height(r->left),height(r->right));
    b=balance(r);
    if(b>1 && balance(r->left)>=0) return rightRotate(r);
    if(b>1){ r->left=leftRotate(r->left); return rightRotate(r); }
    if(b<-1 && balance(r->right)<=0) return leftRotate(r);
    if(b<-1){ r->right=rightRotate(r->right); return leftRotate(r); }
    return r;
}

void inorder(struct Node *r){ if(r){ inorder(r->left); printf("%d ",r->data); inorder(r->right); } }

int main()
{
    struct Node *root=NULL;
    int n,x,i,del;
    scanf("%d",&n);
    for(i=0;i<n;i++){ scanf("%d",&x); root=insert(root,x); }
    scanf("%d",&del);
    root=deleteNode(root,del);
    inorder(root);
    return 0;
}

```

OUTPUT

Input: 7
30 20 40 10 25 35 50
20

Output:
10 25 30 35 40 50

RESULT

Thus the AVL tree and its balancing operations were executed successfully.

5.B DESIGN AND IMPLEMENT HEAP TREE

AIM

To design and implement a max heap with insertion and deletion.

ALGORITHM

1. Initialize an empty heap.
2. Insert each element at the end of the heap.
3. Swap the element with its parent until the max-heap property is satisfied.
4. For deletion, replace the root by the last element.
5. Apply heapify-down until the heap property is restored.
6. Display the heap.

PROGRAM

```
#include <stdio.h>
#define MAX 100

int heap[MAX],size=0;

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

void insert(int x)
{
    int i=++size;
    heap[i]=x;
    while(i>1 && heap[i]>heap[i/2])
    {
        swap(&heap[i],&heap[i/2]);
        i/=2;
    }
}

void deleteMax()
{
    int i=1,l,r,m;
    if(!size) return;
    heap[1]=heap[size--];
    while(2*i<=size)
    {
        l=2*i; r=l+1; m=l;
        if(r<=size && heap[r]>heap[l]) m=r;
        if(heap[i]>=heap[m]) break;
        swap(&heap[i],&heap[m]); i=m;
    }
}

void display()
{
    int i;
    for(i=1;i<=size;i++) printf("%d ",heap[i]);
    printf("\n");
}

int main()
{
    int n,x,i;
    scanf("%d",&n);
    for(i=0;i<n;i++){ scanf("%d",&x); insert(x); }
    printf("Heap: "); display();
    deleteMax();
    printf("After deletion: "); display();
    return 0;
}
```

OUTPUT

Input: 5
40 20 50 10 30

Output:

Heap: 50 30 40 10 20

After deletion: 40 30 20 10

RESULT

Thus the heap tree was designed and implemented successfully.

6.A DESIGN AND IMPLEMENT DIJKSTRA'S ALGORITHM

AIM

To design and implement Dijkstra's algorithm for single-source shortest paths.

ALGORITHM

1. Read the weighted adjacency matrix and source vertex.
2. Set source distance to 0 and all other distances to infinity.
3. Select the unvisited vertex with minimum distance.
4. Mark it visited and update distances of its adjacent vertices.
5. Repeat until all reachable vertices are processed.
6. Display the shortest distances.

PROGRAM

```
#include <stdio.h>
#define MAX 20
#define INF 9999

int main()
{
    int n, c[MAX][MAX], d[MAX], v[MAX]={0};
    int s, i, j, k, min, u;
    scanf("%d", &n);

    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
        {
            scanf("%d", &c[i][j]);
            if(c[i][j]==0 && i!=j) c[i][j]=INF;
        }

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

    for(k=0; k<n; k++)
    {
        min=INF; u=-1;
        for(i=0; i<n; i++)
            if(!v[i] && d[i]<min){ min=d[i]; u=i; }
        if(u==-1) break;
        v[u]=1;
        for(i=0; i<n; i++)
            if(!v[i] && d[u]+c[u][i]<d[i])
                d[i]=d[u]+c[u][i];
    }

    for(i=0; i<n; i++) printf("%d : %d\\n", i+1, d[i]);
    return 0;
}
```

OUTPUT

```
Input: 4
0 5 2 0
5 0 1 3
2 1 0 6
0 3 6 0
```

1

Output:

1 : 0
2 : 3
3 : 2
4 : 6

RESULT

Thus Dijkstra's shortest path algorithm was implemented successfully.

6.B DESIGN AND IMPLEMENT FLOYD WARSHALL ALGORITHM

AIM

To design and implement Floyd-Warshall algorithm for all-pairs shortest paths.

ALGORITHM

1. Read the weighted adjacency matrix.
2. Represent unreachable edges by infinity.
3. Select each vertex as an intermediate vertex.
4. Update $D[i][j]$ if the path through k is shorter.
5. Repeat for all intermediate vertices.
6. Display the final shortest path matrix.

PROGRAM

```
#include <stdio.h>
#define MAX 20
#define INF 9999

int main()
{
    int a[MAX][MAX], n, i, j, k;
    scanf("%d", &n);

    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
        {
            scanf("%d", &a[i][j]);
            if(a[i][j]==0 && i!=j) a[i][j]=INF;
        }

    for(k=0; k<n; k++)
        for(i=0; i<n; i++)
            for(j=0; j<n; j++)
                if(a[i][k]+a[k][j]<a[i][j])
                    a[i][j]=a[i][k]+a[k][j];

    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            printf("%d\\t", a[i][j]);
        printf("\\n\\n");
    }
    return 0;
}
```

OUTPUT

Input: 3

0 4 11
4 0 2
11 2 0

Output:

0 4 6
4 0 2

Thus the Floyd-Warshall algorithm was implemented successfully.

7.A DESIGN AND IMPLEMENT KRUSKAL'S ALGORITHM

AIM

To design and implement Kruskal's algorithm to obtain a minimum spanning tree.

ALGORITHM

1. Read vertices and weighted edges.
2. Sort all edges in ascending order of weight.
3. Make each vertex a separate set.
4. Select the smallest edge and check whether it forms a cycle.
5. Add it if no cycle is formed and continue until n-1 edges are selected.
6. Display the MST and total cost.

PROGRAM

```
#include <stdio.h>
#define MAX 50

int parent[MAX];

int find(int x)
{
    while(parent[x]!=x) x=parent[x];
    return x;
}

void unite(int a,int b){ parent[find(a)]=find(b); }

int main()
{
    int n,e,u[MAX],v[MAX],w[MAX],i,j,t,c=0,total=0;
    scanf("%d%d",&n,&e);

    for(i=0;i<e;i++) scanf("%d%d%d",&u[i],&v[i],&w[i]);
    for(i=0;i<n;i++) parent[i]=i;

    for(i=0;i<e-1;i++)
        for(j=i+1;j<e;j++)
            if(w[i]>w[j])
            {
                t=w[i];w[i]=w[j];w[j]=t;
                t=u[i];u[i]=u[j];u[j]=t;
                t=v[i];v[i]=v[j];v[j]=t;
            }

    for(i=0;i<e && c<n-1;i++)
        if(find(u[i])!=find(v[i]))
        {
            unite(u[i],v[i]);
            printf("%d - %d = %d\\n",u[i],v[i],w[i]);
            total+=w[i]; c++;
        }

    printf("Minimum Cost = %d",total);
    return 0;
}
```

OUTPUT

```
Input: 4 5
0 1 10
0 2 6
```

```

0 3 5
1 3 15
2 3 4

```

Output:

```

2 - 3 = 4
0 - 3 = 5
0 - 1 = 10
Minimum Cost = 19

```

RESULT

Thus Kruskal's minimum spanning tree algorithm was implemented successfully.

7.B DESIGN AND IMPLEMENT PRIM'S ALGORITHM

AIM

To design and implement Prim's algorithm to obtain a minimum spanning tree.

ALGORITHM

1. Read the cost adjacency matrix.
2. Select a starting vertex and mark it visited.
3. Find the minimum-cost edge from a visited vertex to an unvisited vertex.
4. Add the edge and mark the new vertex visited.
5. Repeat until n-1 edges are selected.
6. Display the MST and total cost.

PROGRAM

```

#include <stdio.h>
#define MAX 20
#define INF 9999

int main()
{
    int n, c[MAX][MAX], v[MAX]={0};
    int i, j, e=0, min, u=0, w=0, total=0;
    scanf("%d", &n);

    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
        {
            scanf("%d", &c[i][j]);
            if(c[i][j]==0 && i!=j) c[i][j]=INF;
        }

    v[0]=1;
    while(e<n-1)
    {
        min=INF;
        for(i=0; i<n; i++)
            if(v[i])
                for(j=0; j<n; j++)
                    if(!v[j] && c[i][j]<min)
                    { min=c[i][j]; u=i; w=j; }
        if(min==INF) break;
        printf("%d - %d = %d\\n", u, w, min);
        v[w]=1; total+=min; e++;
    }
    printf("Minimum Cost = %d", total);
    return 0;
}

```

OUTPUT

```

Input: 4
0 10 6 5
10 0 0 15

```

```
6 0 0 4
5 15 4 0
```

Output:

0 - 3 = 5

3 - 2 = 4

0 - 1 = 10

Minimum Cost = 19

RESULT

Thus Prim's minimum spanning tree algorithm was implemented successfully.

8.A IMPLEMENTATION OF DEPTH FIRST SEARCH

AIM

To implement Depth First Search (DFS) algorithm.

ALGORITHM

1. Read the graph and starting vertex.
2. Mark the starting vertex as visited.
3. Visit the vertex and recursively visit each unvisited adjacent vertex.
4. Continue until all reachable vertices are visited.
5. Display the DFS traversal.
6. Stop.

PROGRAM

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

int graph[MAX][MAX],visited[MAX],n;

void DFS(int v)
{
    int i;
    visited[v]=1;
    printf("%d ",v);
    for(i=0;i<n;i++)
        if(graph[v][i] && !visited[i]) DFS(i);
}

int main()
{
    int i,j,start;
    scanf("%d",&n);
    for(i=0;i<n;i++)
        for(j=0;j<n;j++) scanf("%d",&graph[i][j]);
    scanf("%d",&start);
    DFS(start);
    return 0;
}
```

OUTPUT

Input: 4

0 1 1 0

1 0 0 1

1 0 0 1

0 1 1 0

0

Output:

0 1 3 2

RESULT

Thus the Depth First Search algorithm was implemented successfully.

8.B IMPLEMENTATION OF BREADTH FIRST SEARCH

AIM

To implement Breadth First Search (BFS) algorithm.

ALGORITHM

1. Read the graph and starting vertex.
2. Initialize all vertices as unvisited.
3. Insert the start vertex into a queue and mark it visited.
4. Remove a vertex from the queue and visit it.
5. Insert each unvisited adjacent vertex into the queue.
6. Repeat until the queue is empty.

PROGRAM

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

int main()
{
    int graph[MAX][MAX], visited[MAX]={0}, queue[MAX];
    int n, i, j, start, front=0, rear=0, v;
    scanf("%d", &n);
    for(i=0; i<n; i++)
        for(j=0; j<n; j++) scanf("%d", &graph[i][j]);
    scanf("%d", &start);

    queue[rear++]=start; visited[start]=1;
    while(front<rear)
    {
        v=queue[front++];
        printf("%d ", v);
        for(i=0; i<n; i++)
            if(graph[v][i] && !visited[i])
                { visited[i]=1; queue[rear++]=i; }
    }
    return 0;
}
```

OUTPUT

Input: 4
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
0

Output:
0 1 2 3

RESULT

Thus the Breadth First Search algorithm was implemented successfully.

8.C IMPLEMENTATION OF TOPOLOGICAL SORTING

AIM

To implement topological sorting of a directed graph.

ALGORITHM

1. Read the adjacency matrix and calculate indegree of each vertex.
2. Insert all zero-indegree vertices into a queue.
3. Remove a vertex, add it to the order, and reduce indegrees of adjacent vertices.
4. Insert vertices whose indegree becomes zero.

5. Repeat until the queue is empty.
6. If all vertices are not processed, report a cycle.

PROGRAM

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

int main()
{
    int g[MAX][MAX], in[MAX]={0}, q[MAX], order[MAX];
    int n, i, j, f=0, r=0, c=0, v;
    scanf("%d", &n);

    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
        {
            scanf("%d", &g[i][j]);
            if(g[i][j]) in[j]++;
        }

    for(i=0; i<n; i++) if(in[i]==0) q[r++]=i;

    while(f<r)
    {
        v=q[f++]; order[c++]=v;
        for(i=0; i<n; i++)
            if(g[v][i] && --in[i]==0) q[r++]=i;
    }

    if(c!=n) printf("Cycle exists");
    else for(i=0; i<n; i++) printf("%d ", order[i]);
    return 0;
}
```

OUTPUT

Input: 4
0 1 1 0
0 0 0 1
0 0 0 1
0 0 0 0

Output:
0 1 2 3

RESULT

Thus topological sorting was implemented successfully.

9.A IMPLEMENTATION OF SORTING TECHNIQUES

AIM

To implement Insertion Sort, Selection Sort, Quick Sort and Merge Sort to arrange integers in ascending order.

ALGORITHM

1. Read the elements and choose the sorting technique.
2. Insertion Sort inserts each key into the sorted part.
3. Selection Sort selects the minimum from the unsorted part.
4. Quick Sort partitions around a pivot and recursively sorts the parts.
5. Merge Sort divides the array and merges the sorted parts.
6. Display the sorted array.

PROGRAM

```
#include <stdio.h>
#define MAX 50
```

```

void insertion(int a[],int n)
{
    int i,j,x;
    for(i=1;i<n;i++)
    {
        x=a[i]; j=i-1;
        while(j>=0 && a[j]>x){ a[j+1]=a[j]; j--; }
        a[j+1]=x;
    }
}

void selection(int a[],int n)
{
    int i,j,m,t;
    for(i=0;i<n-1;i++)
    {
        m=i;
        for(j=i+1;j<n;j++) if(a[j]<a[m]) m=j;
        t=a[i]; a[i]=a[m]; a[m]=t;
    }
}

void quick(int a[],int l,int r)
{
    int i=l,j=r,p=a[(l+r)/2],t;
    while(i<=j)
    {
        while(a[i]<p)i++;
        while(a[j]>p)j--;
        if(i<=j){ t=a[i]; a[i++]=a[j]; a[j--]=t; }
    }
    if(l<j) quick(a,l,j);
    if(i<r) quick(a,i,r);
}

void merge(int a[],int l,int m,int r)
{
    int t[MAX],i=l,j=m+1,k=l;
    while(i<=m && j<=r) t[k++]=a[i]<=a[j]?a[i++]:a[j++];
    while(i<=m)t[k++]=a[i++];
    while(j<=r)t[k++]=a[j++];
    for(i=l;i<=r;i++)a[i]=t[i];
}

void mergeSort(int a[],int l,int r)
{
    int m;
    if(l<r){ m=(l+r)/2; mergeSort(a,l,m); mergeSort(a,m+1,r); merge(a,l,m,r); }
}

int main()
{
    int a[MAX],n,ch,i;
    scanf("%d",&n);
    for(i=0;i<n;i++) scanf("%d",&a[i]);
    scanf("%d",&ch);
    if(ch==1) insertion(a,n);
    else if(ch==2) selection(a,n);
    else if(ch==3) quick(a,0,n-1);
    else if(ch==4) mergeSort(a,0,n-1);
    for(i=0;i<n;i++) printf("%d ",a[i]);
    return 0;
}

```

OUTPUT

Input: 6
45 12 78 23 9 31
3

Output:
9 12 23 31 45 78

RESULT

Thus the required sorting techniques were implemented successfully.

9.B IMPLEMENTATION OF HASHING TECHNIQUES

AIM

To implement hashing using Linear Probing and Quadratic Probing.

ALGORITHM

1. Initialize both hash tables as empty.
2. Read the keys and calculate the hash index using $\text{key \% SIZE}$.
3. For a collision in Linear Probing, check consecutive positions.
4. For Quadratic Probing, check $(\text{index} + i^2) \% \text{SIZE}$.
5. Insert each key into the first free position.
6. Display both hash tables.

PROGRAM

```
#include <stdio.h>
#define SIZE 10

int linear[SIZE],quad[SIZE];

void initialize()
{
    int i;
    for(i=0;i<SIZE;i++) linear[i]=quad[i]=-1;
}

void insert(int key)
{
    int p=key%SIZE, i, pos;
    for(i=0;i<SIZE;i++)
    {
        pos=(p+i)%SIZE;
        if(linear[pos]==-1){ linear[pos]=key; break; }
    }
    for(i=0;i<SIZE;i++)
    {
        pos=(p+i*i)%SIZE;
        if(quad[pos]==-1){ quad[pos]=key; break; }
    }
}

void display()
{
    int i;
    printf("Linear: ");
    for(i=0;i<SIZE;i++) printf("%d ", linear[i]);
    printf("\nQuadratic: ");
    for(i=0;i<SIZE;i++) printf("%d ", quad[i]);
}

int main()
{
    int n,x,i;
    initialize();
    scanf("%d",&n);
    for(i=0;i<n;i++){ scanf("%d",&x); insert(x); }
    display();
    return 0;
}
```

OUTPUT

Input: 5
23 43 13 27 33

Output:

Linear: -1 -1 23 43 13 33 -1 27 -1 -1
Quadratic: -1 -1 23 43 13 -1 -1 27 33 -1

RESULT

Thus the Linear Probing and Quadratic Probing hashing techniques were implemented successfully.