Ammini College of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CS09 407(P)

DATA STRUCTURE LAB

LIST OF EXPERIMENTS

INTRODUCTION

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures are used in almost every program or software system. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and internet indexing services. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address — a bit string that can be itself stored in memory and manipulated by the program. Thus the record and array data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles

STACK

The stack is a very common data structure used in programs. By data structure, we mean something that is meant to hold data and provides certain operations on that data. Stacks hold objects, usually all of the same type. Most stacks support just the simple set of operations we introduced above; and thus, the main property of a stack is that objects go on and come off of the top of the stack.

Here are the minimal operations we'd need for an abstract stack

Push: Places an object on the *top* of the stack.

Pop: Removes an object from the *top* of the stack and produces that object.

IsEmpty: Reports whether the stack is empty or not.

QUEUES

A queue is a first-in first-out data structure (FIFO). Conceptually, the queue data structure behaves like a line. New data is placed at the rear of the queue and when data is removed it is taken from the front. A printer maintains a list of jobs in a queue; the oldest job, the one that has been in the queue the longest, is serviced first.

The operations supported by a queue are as follows:

- void enqueue(Object o) - place object o on the rear of the queue
- Object dequeue() - remove and return the object at the front of the queue
- Object front() - return the object at the front of the queue without removing it
- int size() - return the size of the queue
- boolean isEmpty() - return true if the queue contains no elements, false otherwise

SINGLY-LINKED LISTS

The singly-linked list is the most basic of all the linked data structures. A singly-linked list is simply a sequence of dynamically allocated objects, each of which refers to its successor in the list. Despite this obvious simplicity, there are myriad implementation variations.

DOUBLY-LINKED LIST

A more sophisticated kind of linked list is a doubly-linked list or two-way linked list. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node. A doubly-linked list containing three integer values: the value, the link forward to the next node, and the link backward to the previous node.

1. IMPLEMENTATION OF STACK USING ARRAY

Aim

To write a C program to implement stack using array implementation

Algorithm

```
Step1 :     Define the maximum size of stack as 5
        Structure
            Define a structure stac as follows
            Struct stack
            {
                int s[size];
                int top;
            }st;
        main() function
step 1:     start
step 2:     Initialize the top of st to -1
step 3:     Display the list of choice available for the user to choose
             from  :
            1.  Push 2. Pop 3. Display 4.exit
Step 4 :    Read the choice from the user as choice
Step 5 :    If choice ==1 then read an item to be pushed from user as
            item.
            If stack is full then display "stack is full" else call the
            function push() with item as argument.
```

```
step 6 :    If choice==2 then check whether "stack is empty".
            If stack is empty then display "stack is empty" else call the
            Function pop().
Step 7 :    If choice ==3 then call the display() function
Step 8 :    If choice==4 then go to step9
Step 9 :    Stop.
```

<u>int stfull()</u> – Function to check whether stack is full.

```
Step 1 :    If top of stack st is greater than or equal to size-1 then
             return1else return 0.
```

<u>void push(int item)</u> – Function to push an item into stack.

```
Step 1:     Increment the value of top of st by 1.
Step 2 :    Store the item in the top position of the s as follows:
            St.s[st.top=item
```

<u>int stempty()</u> – Function to check whether stack is empty.

```
Step 1 :    if top of stack st is -1 then return 1 else return 0.
```

Int pop() – Function to pop an element from stack.

```
Step 1:     Store the element at top of the st to item
Step 2 :    Decrement the value of top by 1
Step 3:     Return the value of item.
```

<u>void display()</u> – Function to display the elements present in the stack

```
Step 1 :    Check whether the stack is empty by calling the function
            stempty()
            If it is empty then display "stack is empty" else go to step2
Step 2 :    Display the elements in the stack using for loop shown
```

below:

```
        for(i=st.top;i>=0;i--)
        printf(“\n\t%d”,st.s[i]);
```

2. IMPLEMENTATION OF QUEUE USING ARRAY

Aim

To write a C program to implement queue using array

Algorithm

```
Step 1 :    Define the maximum size of the queue as 3.
            #define size 3

Main() function

Step 1 :    Start
Step 2 :    Display the list of choices available to the user to choose
            From:
            1. Enqueue
            2. Dequeue
            3. Display
            4. Exit
Step 3 :    If ch==1, then call the function enqueue() and display()
Step 4 :    If ch==2 then call the function dequeue() and display()
Step 5 :    If ch==3 then call the function display()
Step 6 :    If ch==4 then go to step 5
Step 7 :    Stop.

Void enqueue() – function to insert an element in to Queue

Step 1:     If rear=size-1 then display "Queue is Full"
Step 2:     Increment rear by 1.
Step 3:     Read the element from the user and store it in arr[rear]
Step 4:     If front==-1 then display "Queue is empty"
```

Void deueue() – Function to remove an element from queue

```
Step 1:    If front==-1 then display “Queue is empty”
Step 2:    Dequeue the element at arr[front]
Step 3:    If front==rear then assign -1 to front and rear else increment
           Front by 1.
```

Void display()- Function to display the element in queue

```
Step 1:    If front==-1 then display “Empty Queue”
Step 2:    Display the element in the queue using for loop as follows.
           For(i=front;i<=rear;i++)
                Printf(“-%d”, arr[i]);
```

3. SINGLY LINKED LIST

Aim

To write a program to create and perform requested operations on singly linked list.

Algorithm

<u>Self referential structure</u>

```
Step 1:     Define a structure which contains a pointer to a structure of
            same type as follows:
            Structure list :
            Integer data – Variable that holds data in node.
            Structure list * next – pointer of type struct list next holds
            the address of the next node in the list.
            End Structure
Step 2:     Define a variable node which is structure with two members
                Struct list
                {
                    Int data;
                    Struct list *next;
                };
                typedef  struct list node;
Step 3:     Make head as the pointer of type struct set the list initially as
            NULL as given below:
                node * head=NULL;
```

<u>create (int n)</u> – Function for list creation

```
Step 1:     Declare a pointer variable new of the type node.
Step 2:     Dynamically allocate the memory to the new using malloc
            function.
                    New=(node *)malloc(size of node));
Step 3:     The n is assigned to the data of new
                    new→ data=n
Step 4:     Assign NULL to the next of new
                    new→next=NULL
Step 5:     new is assigned as the head node of the list.
                    Head=new
```

<u>Insbeg(int n)</u> – Adds the node at the beginning of the list.

```
Step 1:     Declare a pointer variable new of the node.
Step 2:     Dynamically allocate the memory to the new using malloc
            Function
                    New=(node *)malloc(sizeof(node))
Step 3:     The n is assigned to the data of new
                    New→data=n
Step 4:     if head==NULL (i.e., if the list is empty 0 then perform step
            5 else goto step 6.
Step 5:
            5.1:  The new is assigned to the head.
                            Head=new
            5.2: Assign NULL to the next of head
                            Head→ next=NULL
```

```
Step 6:
        6.1: Assign head to the next of new
                New→next=head
        6.2: Assign new to head
                Head=new
```

Insend(int n) – Adds the node at the end.

```
Step 1:     Declare the pointer variables new and temp of type node.
Step 2:     Dynamically allocate the memory to new with malloc
            function
                new=(node *)malloc(size of(node));
Step 3:     Assign NULL to the next of new
                New→next=NULL
Step 4:     n is assigned to the data of new
                New→data=n
Step 5:     Assign the head to temp
                Temp=head
Step 6:     Perform step 7 until next of temp is not equal to NULL
                (temp→next!=NULL) else go to step 8.
Step 7:     Assign next of temp to temp
                Temp=temp→next
Step 8:     Assign new of the next of temp
                Temp→next=new
```

Inspos(int n) – Adding node at specified position

```
Step 1:     Declare the pointer variables new and temp of type node.
Step 2:     Declare two variables pos, i
Step 3:     Dynamically allocate the memory to the new using malloc
            Function
                New=(node *)malloc(sizeof (node));
Step 4:     n is assigned to the data of new
                New=data=n
Step 5:     Read the position from the user (pos)
Step 6:     If pos==1 then call the function insbeg() with n as parameter
            Else perform step7
Step 7:
            7.1: head is assigned to the temp       temp=head
            7.2: for(i=2;i<pos;i++)   Perform step 7.3
            7.3:
            7.3.1: If next of temp is not equal to NULL
                    (i.e., temp→next!=NULL)perform 7.3.2
            7.3.2: next of temp is assigned to temp
                    Temp=temp→next.
            7.4: Assign next of new
                    New→next=temp→next
            7.5: Assign new to the next of temp
                    Temp→next=new
```

<u>delbeg()-</u> Deletion of node at beginning.

```
Step 1:     Declare a pointer variable temp of type node.
Step 2:     If head==NULL then display "List is empty" else go to step3
Step 3:
        3.1: head is assigned to temp.        Temp=head.
        3.2: Assign the next of temp to head
            Head=temp→next.
Step 4: Deallocate the memory of temp using "true" function
        free(temp);
```

Delend()- Deletion of node at end

```
Step 1:   Declare two pointer variable s temp and prev of type node
Step 2:   Assign the head to temp.        (temp=head)
Step 3:   Perform step-4 until next of temp is not equal to NULL.
          (i.e., temp→next!=NULL)
Step 4:
          4.1: Assign temp to prev        (prev=temp0
          4.2: next of temp is assigned to temp.
                    Temp=temp→next
Step 5:   Assign NULL to the next of prev
              Prev→next=NULL
Step 6:   Deallocate the memory of temp using free function
              free(temp)
```

delpos()- Deletion of node at specified position.

```
Step 1:   Declare two pointer variables temp and prev of type node
Step 2:   Declare two variables pos and i
Step 3:   If head==NULL hen display "List is empty" else go to step 4
Step 4:   Assign head to temp. => temp=head
Step 5:    Read the position for deletion from the user (pos)
Step 6:    If pos==1 then call the function delbeg() else go to step 7
Step 7:
           7.1:  for(i=2;i<=pos;i++)
           7.2:
           7.2.1: if next of temp is not equal to NULL
                  (i.e., temp→next!=NULL) perform step 7.2.2
           7.2.2: Assign the temp to prev (prev=temp)
           7.3: Assign the next of temp to next of prev
                        Prev→next=temp→next
           7.4: Deallocate the memory of temp using free function
                        free(temp);
```

<u>search()</u> – Search for an element

```
Step 1:    Declare the pointer variable temp of type node
Step 2:    Declare a variable item and initialise i as 1
Step 3:    Read the data to be searched from the user (item)
Step 4:    Assign the head to temp (temp=head)
Step 5:    Perform Step 6 until next of temp is not equal to NULL
           (i.e., temp→next!=NULL) and data of temp is not equal to
           item. (temp→data!=item)
Step 6:
           6.1: Assign of temp to temp
                  Temp=temp→next
           6.2: i is incremented by 1
```

```
Step 7:     If data of temp is equal to item (temp→data==item)
            Display "Item is found at i" else display "Item is not found"
```

Display() – Display the list contents.

```
Step 1:     Display a pointer variable temp of type node.
Step 2:     Assign head to temp. (temp=head0
Step 3:     Display "head→"
Step 4:     Perform step -5 until temp is not equal to NULL.
Step 5:
            5.1: Display the data of temp
            5.2: Assign the next of temp to temp
                    Temp=temp→next
Step 6:     Display "NULL"
```

Main() function

```
Step 1:     Start
Step 2:     Display a list of operation for the user to choose from
            1. Single list creation
            2. Insert at beginning
            3. Insert at end
            4. Insert at position
            5. Delete at beginning
            6. Delete at end
            7. Delete at position
            8. Search for an element
            9. Exit
Step 3:     Read the choice entered by the user (choice).
Step 4:     If choice==1 then call the function create(no) and display()
            Function
```

```
Step 5:     If choice==2 then call the function insbeg(no) and display()
            Function
Step 6:     If choice==3 then call the function insend(no) and display()
            Function
Step 7:     If choice==4 then call the function inspos(no) and display()
            Function
Step 8:     If choice==5 then call the function delbeg(no) and display()
            Function
Step 9:     If choice==6 then call the function delbeg(no) and display()
            Function
Step 10:    If choice==7 then call the function delpos(no) and display()
            Function
Step 11:    If choice==8 then call the function search()
Step 12:    If choice==8 then go to step13.
Step 13:    Stop.
```

4. STACK USING LINKED LIST

Aim

To write a program to perform operation on stack using linked list.

Algorithm

For linked list operation there is a node having 2 parts. One part is the data part and other printer part.

Top and t are node type.

Main() function

```
Step 1:    Start
Step 2:    Display the menu
Step 3:    If a=1 call the function push then go to step 8
            else go to Step 4
Step 4:    if a=2 call the function pop then go to Step 8 else go to Step5
Step 5:    if a=3 call the function display then go to Step 8
           else go to Step 6.
Step 6:    If a=4 go to Step 9
           Else go to Step 7
Step 7:    print wrong choice.
Step 8:    if a!=4 go to Step 2.
Step 9:    Stop.
```

Function push()

```
Step 1:    Start
Step 2:    t←new node
Step 3:    Read element in x
Step 4:     t→data←x
Step 5:     t→next←top
Step 6:     top←t
Step 7:     print one element is inserted
Step 8:     Return
```

<u>Function Display</u>

```
Step 1:     Start
Step 2:     t←top
Step 3:     if t!=NULL go to Step 4
                Else go to Step 6
Step 4:     print t←data
Step 5:     t←t→next and go to Step 3
Step 6:     Return
```

<u>Function pop</u>

```
Step 1:     start
Step 2:     t←top
Step 3:     x←t→data
Step 4:     top←top→next
Step 5:     free(t)
Step 6:     print the element that is poped.
Step 7:     Return
```

5. QUEUE USING LINKED LIST

Aim

To write a program to perform operation on queue using linked list.

Algorithm

For linked list operation there is a node having 2 parts. One is the data part and other is printer part, pointing to the next node.

Front, rear, t are the node type

Main() function

```
Step 1:    Start
Step 2:    Display the menu
Step 3:    if a=1 call function enqueue then go to Step 8 else gotoStep4
Step 4:    if a=2 call function dequeue then go to Step 8 else go to
           Step 5
Step 5:    if a=3 call function display then go to Step 8  else goto Step6
Step 6:    if a=4 call function exit then go to Step 9  else go to Step 7
Step 7:    Print wrong choice
Step 8:    is a!=4 go to Step 2
Step 9:    Stop
```

Function enqueue()

```
Step 1:    Start
Step 2:    Create a new node t
Step 3:    Read the element in x
Step 4:    t→data←
Step 5:    if rear=NULL   front←t, rear←t
```

```
                Else rear→next←t
Step 6:         rear←t
Step 7:         rear→next=NULL
Step 8:         print one element is inserted
Step 9:         Return
```

Function Dequeue()

```
Step 1:         Strat
Step 2:         if front=NULL Q is empty return else step 3
Step 3:         t→front
Step 4:         x←t→data
Step 5:         front←front→next
Step 6:         free(t)
Step 7:         print x
Step 8:         Return
```

6. SINGLY LINKED LIST USING POLINOMIAL ADDITION

Aim

To write a program to perform polynomial addition operation using linked list.

Algorithm

```
SELF REFRENCIIAL DTRUCTURE
Step 1:    Define a structure  which contains a pointer to a structure of same
           type as follows.
       Structure list
           Integer coeff, pow    - variables that datas in the node.
           Structure list *next  - pointer of type struct list next holds the
                                   address of the next node in the list.
           End structure
Step 2:    Define a new type node which is structure with three members
               Typedef struct list node;
Step 3:    Make head as the pointer of  type struct.
           Set the list initially as NULL
               Node *head=NULL;
Step 4:    Repeat the step 3 for poly1,poly2 and poly as given below.
               Node *poly1=NULL,*poly2=NULL,*poly=NULL;
Step 5:    Declare the function create(),display() and polyadd() with void as
           Return type as return type as given below.
               Void create(node  *);
               Void display(node  *);
```

```
                Void polyadd(node *,node *,node*);

    Main() function

    Step 1:     Start
    Step 2:     Dynamically allocate the memory to poly1 using malloc
                Function
    Step 3:     call the function create() with poly1 as argument
    Step 4:     Repeat the steps 2 and 3 for poly2
    Step 5:     Display the polynomials poly1 and poly2 by calling the
                Function  display() with poly1 and  poly2 as arguments
                Respectively.
    Step 6:     Dynamically allocate the memory to resultant polynomial
                poly using malloc function
    Step 7:     Call the function polyadd() with poly1, poly2 and poly as
                Arguments.
                        Poly(poly1,poly2,poly);
    Step 8:     Display the resultant polynomial poly by calling the
                display() function
    Step 9:     Stop.

  Create(node *p)

    Step 1:     Declare a character variable ch
    Step 2:     Read the value for coefficient from the user and store it in
                Coeff  of p as follows
                        Scanf("%d",&p→coeff);
    Step 3:     Read the value for power from the user and store it in pow of
```

```
            p as follows.
                Scanf("%d",&p→pow);
Step 4:     If  the user wants to continue then perform Step 5
            else go to Step 6
Step 5:     5.1: Dynamically allocate the memory to next of p using
                malooc() function
                p→next=(node *)malloc(size of (node));
            5.2: next of p is assigned to p
                p=p→next;
Step 6:     Assign NULL to the next of p
            p→next=NULL;
```

<u>display(node *p)</u>

```
Step 1:     Perform the Step 2 until p is not equal to NULL
                P!=NULL
Step 2:     2.1: Display the coefficient and power of the each term in
                The polynomial with the help of following statement.
                Printf("%d%d",p→coeff,p→pow);
            2.2: next of p is assigned to p
                P=p→next
            2.3: If p is not equal to NULL (i.e., p!=NULL) then
                display '+'
```

<u>void polyadd(node *p1,node *p2,node *p)</u>

```
Step 1:     Perform Step 2 to Step 9 until p1 not equal to NULL and p2
            not equal to NULL
```

```
                (i.e., p1!=NULL&&p2!=NULL)
    Step 2:     If pow of p2 equal to pow of  p1 (i.e., p1→pow==p2→pow)
                then perform Step 3 else go to Step 4.
    Step 3:     3.1: Assignpow of p1 to pow of p
                        p→pow=p1→pow
                3.2: Assign the sum of the coeff of p1 and p2 to coeff of p
                        p→coeff=p1→coeff+p2→coeff;
                3.3: Assign next of p1 to p1.     (p1=p1->next)
                3.4: Assign next of p2 to p2.     (p2=p2→next)
    Step 4:     If pow of p1 is greater than the pow of p2
                (i.e., p1→pow>p2→pow) then perform Step 5
                else go to Step 6
    Step 5:     5.1:    pow of p1 is assigned to the pow of p
                        p→pow=p1→pow
                5.2:    coeff of p1 is assigned to the coeff of p
                        p→coeff=p1→coeff
                5.3:    Assign next of p1 to p1
                        P1=p1→next

Step 6:         6.1:    Assign pow of p2 to pow of p
                        p→pow=p2→pow;
                6.2:    Assign coeff of p2 to coeff of p
                        p→coeff=p2→coeff;
                6.3:    Assign next of p2 to p2
                        P2=p2→next;
    Step 7:     If p1 is not equal to NULL or p2 is not equal to NULL
                (i.e., p1!=NULL||p2!=NULL) then perform Step 8
                 else go to Step 9
    Step 8:     8.1:   Dynamically allocate the memory to next of p using
                 malloc() function.
                        p→next=(node *)malloc(sizeof(node));
                8.2:   Assign next of p to p
                        P=p→next;
    Step 9:     Assign NULL to next of p
                p→next=NULL;
    Step 10:    Perform Steps 11 to Step 17 untill p1 is not equal to NULL
                or p2 is not equal to NULL.
                (i.e., p1!=NULL||p2!=NULL)
    Step 11:    If p1 not equal to NULL then perform Step 12.
                Else go to Step 13


    Step 12:    12.1:  Assign pow of p1 to pow of p
                        p→pow=p1→pow
                12.2: Assign coeff of p1 to coeff of p
                        p→coeff=p1→coeff
                12.3: Assign next of p1 to p1    (p1=p1→next)
    Step 13:    If p2 is not equal to NULL then perform Step 14
                 else go to Step 15
```

```
Step 14:    14.1: Assign pow of p2 to pow of p
                  p→pow=p2→pow
            14.2: Assign coeff of p2 to coeff of p
                  p→coeff=p2→coeff
            14.3: Assign next of p2 to p2
                  P2=p2→next
Step 15:    If p1 not equal to NULL or p2 not equal to NULL
            (i.e., p1!=NULL||p2!=NULL) then perform step 16
            Else go to Step 17.
Step 16:    16.1: Dynamically allocate the memory to next of p using
                  Malloc() function
                  p→next=(node *)malloc(sizeof(node));
            16.2: Assign next of p to p
                  P=p→next;
Step 17:    Assign NULL to next of p
            p→next=NULL
```

7. DOUBLY LINKED LIST

Aim

To write a program to create and perform various operations, such as insertion, deletion and searching for an element on doubly linked list.

Algorithm

SELF- REFERENTIAL STRUCTURE

```
Step 1:     Define a structure which contains a pointer to a structure of
            same as follows:
            Structure list
            Integer data – Variable that holds data in node.
            Structure list * next, *prev- pointer of type struct list next
            holds the address of the next node in the list.
            End Structure
Step 2:     Define a variable node which is structure with two members
            Struct list
            {
            Int data;
            Struct list *prev, *next;
            };
            typedef struct list node;
Step 3:     Make head as the pointer of type struct. Set the list initially
            as NULL as given below.
            Node *head=NULL
```

<u>Create(int n)</u> - Function for list creation.

```
Step 1:    Create a pointer variable new of type node
Step 2:    Dynamically allocate the memory to new using malloc()
           New=(node *)malloc(sizeof(node));
Step 3:    Assign n to the data of new
           New→data=n
Step 4:    Assign NULL to the next of new
           New→next=NULL
Step 5:    Assign NULL to the prev of new
           New→prev=NULL
Step 6:    Assign new to head
           Head=new
```

<u>Void insbeg(int n)</u> - Function for insertion at beginning

```
Step 1:    Declare a pointer variable new of type node.
Step 2:    Dynamically allocate the memory to new using malloc()
               New= (node *)malloc(sizeof node));
Step 3:    Assign n to the data of new
           New→data=n
Step 4:    If head==NULL then perform Step 5 else go to Step 6
Step 5:
           5.1:  Assign new to head        head=new
           5.2:  Assign NULL to the prev of new
                     New→prev=NULL
           5.3:  Assign  NULL to the nextof new
                     New→next=NULL
```

```
Step 6:
            6.1:   Assign head to the next of new
                        New→next=head
            6.2:   Assign new to the prev of head
                        Head→prev=new
            6.3:   Assign new to head        head=new
            6.4:   Assign NULL to prev of head
                        Head→prev=NULL
```

<u>Void insend(int n)</u> - Function to perform insertion at end

```
Step 1:     Declare two pointer variables new and temp of type node
Step 2:     Dynamically allocate the memory to new using malloc()
            New= (node *)malloc(sizeof (node));
Step 3:     Assign  n  to the data of new
            New→data=n
Step 4:     Assign  NULL to the next of new
            New→next=NULL
Step 5:     Assign head to temp
            Temp=head
Step 6:     Perform Step 7 until next of temp is not equal to NULL
            (i.e., temp→next!=NULL)
Step 7:     next of temp is assigned to temp
            Temp=temp→next
Step 8:     Assign new to the next of temp
            Temp→next=new
```

```
  Step 9:     Assign temp tothe prev of new
              New→prev=temp
```

<u>Void inspos(int n)</u>- Function to insert at given position

```
  Step 1:     Declare  a variable x of integer type.
  Step 2:     Declare a pointer variable new,temp and temp1 of type node.
  Step 3:     Dynamically allocate the memory to new using malloc()
  Step 4:     Assign n to the data of new
                  New→data=n;
  Step 5:     Read the position from the user (x)
  Step 6:     If x==1 then call the function insbeg(n) else go to Step 7
  Step 7:     7.1: Assign head to temp          temp=head
              7.2: for(i=2;i<x;i++)        perform Step 7.3
              7.3:
              7.3.1 : If next of temp to not equal to NULL
                  (i.e., temp→next!=NULL) perform Step 7.3.2
              7.3.2 : next of temp is assigned to temp
  Step 8:     next of temp is to temp1
                  Temp1=temp→next
  Step 9:     temp is assigned to prev of new
                  New→prev=temp
  Step 10:    temp1is Assigned to next of new
                  New→next=temp1
  Step 11:    new is assigned to next of temp
                  Temp→next=new
```

```
Step 12:    new is assigned to prev of temp1
                Temp1→prev=new
```

<u>Void disbeg()</u> – function to delete at beginning

```
Step 1:     Declare a pointer variable temp of type node
Step 2:     If head==NULL then display " List is empty"
             else go to Step 3
Step 3:
            3.1:  head is assigned to temp  =>temp=head
            3.2:  next of head is assigned to head
                  Head=head→next
            3.3:  Assign NULL to the prev of head
                      Head→prev=NULL
Step 4:
            4.1:  Perform Step 4.2 until temp→next!=NULL
            4.2:  next of temp is assigned to temp
                      Temp=temp→next
            4.3:  prev of temp is assigned to temp1
                      Temp1=temp→prev
            4.4:  Assign NULL to prev of temp
                      Temp→prev=NULL
Step 5:     Assign NULL to the next of temp1
                      Temp1→next=NULL
Step 6:     Deallocate the memory of temp using free() function
                      Free(temp);
```

Void delpos() – Function to delete at a given position

```
Step 1:    Declare pointer variables temp, temp1, temp2 of type node.
Step 2:    Declare two variables i and x of integer type
Step 3:    if head==NULL then display "List is empty"
           Else go to Step 4
Step 4:
           4.1:  Assign head to temp              =>temp=head
           4.2:  Read the position from the user (x)
           4.3:  If x==1 then call the function delbeg()
                 Else perform Step 4.4
           4.4:
                 4.4.1:  for(i=2;i<=x;i++) perform steps 4.4.2 to 4.4.5
                 4.4.2: If temp→next!=NULL perform Step 4.4.3
                        Else go to Step 4.4.4
                 4.4.3: Assign next of temp to temp
                        Temp=temp→next
                 4.4.4: Assign prev of temp to temp1
                        Temp1=temp→prev
                 4.4.5: Assign next of temp to temp2
                        Temp2=temp→next
                 4.4.6  :Assign temp1 to prev of temp2
                        Temp2→prev=temp1
                 4.4.7: Assign temp2 to next of temp1
                        Temp→next=temp2
```

```
                    4.4.8: Assign NULL to tje next of temp
                           Temp→next=NULL
                    4.4.9: Assign NULL to the prev of temp
                           Temp→prev=NULL
  Step 5:      Deallocate the memory of temp using free()
                           Free(temp0;
```

<u>Void search()</u>- Function to search for an element

```
  Step 1:      Declare a pointer variable temp of the type node.
  Step 2:      Declare two variables item of i of integer type.
  Step 3:      Read the data to be searched from the user
  Step 4:      Assign head to temp        temp=head
  Step 5:      Perform Step 6 until
                (temp→next!=NULL &&temp->data!=item)
               Else go tp Step 7
  Step 6:
               6.1:    Assign next of temp to temp
                       Temp=temp→next
               6.2:    Increment the variable i by 1
  Step 7:      If (temp→data==item) display "Itemis found at i"
               Else go to Step 8
  Step 8:      Display "Item not found"
```

<u>Void display()</u>- Function to display the list

```
  Step 1:      declare a pointer variable temp of type  node
  Step 2:      Assign head to temp        temp=head
```

```
Step 3:     Display head
Step 4:     Perform step 5 until temp!=NULL
            Else go to Step 6
Step 5:
            5.1:   Display the data
            5.2:   Assign next temp to temp
                   Temp=temp→next
Step 6:     Display "NULL"
```

<u>Main() Function</u>

```
Step 1:     Start
Step 2:     Display a list of operations for the user to choose from
            1. Single list creation
            2. Insert at beginning
            3. Insert at end
            4. Insert at position
            5. Delete at beginning
            6. Delete at end
            7. Delete at position
            8. Search for an element
            9. Exit
Step 3:     Read the choice from the user
Step 4:     If choice=1 call the function create(no) and display()
Step 5:     If choice=2 call the function insbeg(no) and display()
Step 6:     If choice=3 call the function insend(no) and display()
Step 7:     If choice=4 call the function inspos(no) and display()
Step 8:     If choice=5 call the function delbeg() and display()
Step 9:     If choice=6 call the function delend() and display()
Step 10:    If choice=7 call the function delpos() and display()
Step 11:    If choice=8 call the function search()
Step 12:    If choice=9 then go to Step 13
Step 13:    Stop.
```

8. QUICK SORT

Aim

To sort n number in ascending order using quick sort algorithm

Algorithm

```
Step 1:    Start
Step 2:    Read n numbers to be sorted in to an array
Step 3:    Call the function quick sort passing variable
Step 4:    print the numbers
Step 5:    stop
```

Function Quick sort()

```
Initial state m←0,n←n-1
Step 1:    Start
Step 2:    If m<n go to Step 3
Step 3:    i←m,j←n,p←a[m]
Step 4:    i←i+1
Step 5:    if((a[j]>=p)&&(j>m)) go to Step 7
           Else go to Step 8
Step 6:    if((a[i]<=p)&&(j<=n) go to Step 4
           Else go to Step 6
Step 7:    j←j-1 go to Step 6
Step 8:    if i<j go to Step 9
           Else go to Step 10
Step 9:    swap a[i]&a[j]
Step 10:   if i<=j go to Step 4
```

```
            Else go to Step 11
Step 11:    swap a[m]&a[j]
Step 12:    Call function quick sort by passing a,m,j-1
Step 13:    Call function quick sort by passing a,j+1,n
Step 14:    Return.
```

9. MERGE SORT

Aim

To sort n numbers in ascending order using merge sort

Algorithm

```
Step 1:     Start
Step 2:     Enter the range and numbers to be sorted
Step 3:     Call the function merge sort and pass x,y,n
Step 4:     if((n/2)%2==0)
                Print elements in  x
                Else print elements in y
Step 5:     Stop
```

Function merge sort

```
Step 1:     Start
Step 2:     l←1
Step 3:     if l<n go to Step 4 else return
Step 4:     Callthe function merge pass &pass x,y,n,l
Step 5:     l=2*l
Step 6:     call function merge pass and pass x,y,n,l
Step 7:     l=2*l and go to Step 3
```

Function merge pass

```
Step 1:     start
Step 2:     i=0
Step 3:     if i<(n-2*l+1) go to Step 4 else go to Step 6
Step 4:     call functionmerge and pass x,y,i,i+l-1,i+2l-1
Step 5:     i=l+2*l  go to Step 3
Step 6:     if(i+i-1)<n do step 7
            Else go to Step 8
Step 7:     call function &pass x,y,i,i+l-1,n-1& go to Step 12
Step 8:     k=j
Step 9:     if k<n do step 10
            Else go to Step 12
Step 10:    y[k]=x[k]
Step 11:    i=k+1& go to Step 9
Step 12:    Return
```

Function Merge

```
Step 1:     Start
Step 2:     i=l, j=m+1, k=l
Step 3:     if  i<=m&j<=n do Step 4
            Else do Step 8
Step 4:     if x[i]<x[j] do Step 5
            Else do Step 6
Step 5:     y[k]=x[j],i=j+1 go to Step 7
Step 6:     y[k]=x[j],j=j+1
Step 7:     k=k+1 go to Step 3
Step 8:     if i>m do Step 9 else go to Step 11
Step 9:     if j<=n do Step 11
            Else go to Step 13
```

```
Step 10:    y[k]=x[j],j=j+1,k=k+1 go to Step 7
Step 11:    if i<=m do Step 12 else go to Step 13
Step 12:    y[k]=x[i]], i=i+1 k=k+1 go to Step 11
Step 13:    Return
```

10. INFIX TO POSTFIX CONVERTION

Aim

To write a program to convert a given infix expression to postfix expression

Algorithm

Main() Function

```
Step 1:   Start
Step 2:   Read the maximum number of character from the user
Step 3:   for(i=0;i<max;i++), read the characters from the user and
          strore in infix[i] and if infix[i]==’#’ then go to Step 4
Step 4:   for(i=0;i<max;i++) display the infix expression
              Putchar(infix[i])
Step 5:   Call the function infix-postfix()
Step 6:   for(i=0;i<max;i++) display the postfix expression
              Putchar(postfix[i])
Step 7:   stop
```

Infix-postfix() - function to convert an infix expression to postfix Expression

```
Step 1:   Declare a counter variable i,a character variable next and
          initialise p as 0
Step 2:   stack[top]=’#’
Step 3:   len=strlen(infix)
Step 4:   infix[len]=’#’
Step 5:   for(i=0;infix[i]=’#’;i++) perform 6 to 11
Step 6:     if infix[i]='(' then call push() with infix[i] as argument
Step 7:     if infix[i]=')' then perform Step 8
Step 8
            8.1:  next=pop()
            8.2:  postfix[p++]=next;
Step 9:     if infix[i]='+','-',*','/','%',or '^' then perform Step 10
Step 10:
            10.1: precendence=prec(infix[i])
            10.2: Perform Step 10.3 until the operator!='#' and
                  Precendence of the operator <=store in stak[]
            10.3: postfix[p++]=pop()
            10.4: call push() with infix[i] as argument
Step 11:    if infix[i]!=anyone mentioned above store the character in
            the postfix[]
Step 12:    perform Step 13 until the operator in stack[]!='#'
Step 13:
            13.1: postfix[p++]=pop()
            13.2: postfix[p]='\0'
```

<u>Void push(char symbol)</u>- Function to perform push operation in stack

```
Step 1:     If (top>max) then display "Stack overflow"
            Else go  to Step 2
Step 2:
                  2.1: top++
                  2.2:   stak[top]=symbol
```

Int pop()- Function to perform pop operation in stack

```
Step 1:    If top==-1 then display "Stack underflow"
                Else go to Step 2
Step 2:    Return the operator stored in stack[]
                Return (stack[top--])
```

Int prec(char symbol)- Function to assign precedence to an operator

```
Step 1:    If symbol=='(' then return 0
Step 2:    If symbol=='+' or '-' then return 1
Step 3:    If symbol=='*' or '/' or '%' then return 2
Step 4:    If symbol=='^' then return 3
Step 5:    Return 0
```

11. IMPLEMENTATION OF EXPRESSION TREE CONSTRUCTION AND TREE TRAVERSAL

Aim

To write a c program to create an expression tree from the given postfix expression and find the prefix and infix expression for that expression

Algorithm

Self referential structure

```
Step 1:     Define a structure which contains a pointer to a structure
            Struct tree
            {
                    Char data;
                    Struct  tree *lchild;
                            Struct tree *rchild;
                    };
    Step 2:         Define a variable node for that structure using typedef
                    Typedef struct tree node;
    Step 3:         Declare an array pointer of size 50 for the node
                    Node * stack[50];
```

<u>Void push(node *t)</u>- Function to push an element into stack

```
    Step 1:         top++
    Step 2:         stack[top]=t
```

<u>Node pop()</u> - function to pop out an element from stack

```
    Step 1:         Declare  pointer variable t of type node
                            Node *t;
```

```
Step 2:    Assign the element at stack[top--] to t and return it.
           T=stack[top--]
           Return t;
```

<u>void inorder(node *ptr)</u>- Function to display inorder expression

```
Step 1:    If ptr!=NULL then perform Step 2 to 4
Step 2:    Call the recursive function inorder() with ptr→lchild as
           Argument
Step 3:    Print the data at ptr
Step 4:    Call inorder() with ptr→rchild as argument recursively
```

<u>void preorder</u>(node *ptr) – Function to display preorder expression

```
Step 1:    If ptr!=NULL then perform steps 2 to 4
Step 2:    Print the data at ptr
Step 3:    Call preorder() with ptr→lchild as argument, recursively
Step 4:    Call preorder() with ptr→rchild as argument, recursively
```

<u>void postorder(node *ptr)-</u> Function to display the postorder expression

```
Step 1:    if ptr!=NULL then Steps 2 to 4
Step 2:    Call postorder() with ptr→lchild as argument recursively
Step 3:    Call postorder() with ptr→rchild as argument recursively
Step 4:    Print the data at ptr
```

<u>node *getnode(char s)-</u> Function to store the operators in the tree

```
Step 1:    Declare a pointer variable t of type node
Step 2:    Dynamically allocate the memory to t using malloc()
           T=(node *0malloc(sizeof(node));
Step 3:    Store the s at the data of t
Step 4:    Assign the s at the data of t

Step 5:    Return t
```

<u>void logicalview</u>(node *t,int col,int row,int wid,int call)

```
Step 1:    call gotoxy() with col and row as arguments

Step 2:    if t!=NULL then perform Step 3 to 7

Step 3:    if call==0 then display"Root"

Step 4:    Call the function gotoxy with col and row+1

Step 5:    Display data of t.

Step 6:    Call logicalview() recursively with t->rchild,col+wid,
           Row+2,wid/2 and call+1 as arguments
```

<u>Main() function</u>

```
Step 1:    Start

Step 2:    Declare 2 pointer variables type node
           Node *temp,*new1

Step 3:    Read the positive expression from the user as str[50]

Step 4:    for(i=0;str[i]!='\0';i++)

Step 5:    if the character at str[i] is alphanumeric then perform Step 6
           Else go to Step 7

Step 6
           6.1:  call getnode(str[i]) and assign it to new
           6.2:  call the push() with new1 as argument

Step 7:
           7.1:  call getnode(str[i]) and assign it to new1
           7.2:  Assign new1 to temp
           7.3:  call pop() for left and right childs of temp
```

```
                    Temp→rchild=pop();
                    Temp→lchild=pop();
               7.4: call push() with temp as argument
Step 8:        Display expression tree by calling logicalview() with temp
Step 9:        Display the preorder, inorder and postorder expression by
                calling preorder(),inorder() and postorder() with temp as
               argument respectively.
Step 10:       Stop
```

12. IMPLEMENTATION OF BINARY SEARCH TREE

Aim

To write a C program to create a binary search tree and perform various operations like insertion, deletion, find an element, find minimum element, maximum element and display the tree on it.

Algorithm

```
        Self refrential structure

    Step 1:     Define a structure which contains a pointer to the structure
                of same type
                    struct tree
                    {
                        Int data;
                        Struct tree *left;
                        Struct tree *right;
                    };
    Step 2:     Define a variable node of the tree
                    Typedef struct tree node;
    Step 3:     Declare a pointer variable t for node and assign it to NULL
                    Node *t=NULL;
    Step 4:     Declare a global variable call of integer type and
                Assign it to 0
                        Int call=0;
```

Node * makeempty(node *)- Function for deleting all the element in the tree

```
Step 1:    If t!=NULL then perform Step 2 to 4
Step 2:    Call the function makeempty() recursively with t→left as
           argument to delete left child of a node
Step 3:    Call the function makeempty() recursively with t→right
                   As argument to delete right child of a node.
Step 4:    Deallocate the memory of  t using free()function
Step 5:    Return NULL
```

Node *find (int x,node *t)- Function for finding the element in tree

```
Step 1:    If t==NULL then return NULL else go to Step 2.
Step 2:    If x<t→data then return find(x,t→left)
           Else go to Step 3
Step 3:    If x>t→data then return find(x,t→right)
```

Node *findmin(node *t)- Function for finding minimum element.

```
Step 1:    If t==NULL then return NULL else go to Step 2
Step 2:    If there is no left child then return the element at t else
           Call the function findmin(t→left) recursively
```

Node *findmax(node *t)- Function for finding maximum element

```
Step 1:    If t==NULL then return NULL else go to Step 2
Step 2:    If there is no right child then return the element at t.
           Else call recursive function findmax[t→right)
```

Node *insert(int x,node *t)- Function to insert element

```
Step 1:    If t==NULL perform Step 2 else go to Step 3
Step 2:
```

```
                2.1:  Dynamically allocate the memory to t using malloc()
                      T=(node *)malloc(sizeof(node));
                2.2:  Store x in data of t
                2.3:  Make the left and right child of t to point to NULL
Step 3:
                3.1:  If x<t→data then insert x as left child of t
                3.2:  If x>t→data then insert x as right child of t
Step 4:         Return t
```

<u>Node * delete (int x, node *t)-</u> Function to delete element

```
Step 1:         Declare a pointer variable temp of type node.
Step 2:         If t==NULL then display "Element not found"
                Else go to Step 3
Step 3:         If x<t→data then traverse along left child for deleting x by
                 calling the function delete(x,t→left) and Assign  it to t→left
                 else go to Step 4.
Step 4:         If x>t→data then traverse along right child for deleting x by
                calling delete(x,t→right) and Assign it to t→right
                 else go to Step 5
Step 5:         If both left and right child exist for t then perform Step 6
                Else go to Step 7
Step 6:
                6.1:  Call findmin(t→right) and assign it to temp
                      Temp=findmin(t→right);
                6.2:  Assign data of tempto data of t
                      t→data=temp→data;
```

```
            6.3:   call delete(t→data,t→right) and assign it to t→right
                   t→right=delete(t→data,t→right);
    Step 7:
            7.1:   Assign t to temp
            7.2:   If t→left==NULL then Assign the data at right child
                   To t, else go to Step 7.3
            7.3:   If t→right==NULL then assign
                   the data at left child of t to t.
    Step 8:     Return t
```

<u>Void disply</u>(node *t,int col,int row,int wid,int call)

```
    Step 1:     Call gotoxy() with col and row as argument
    Step 2:     If t!=NULL then perform Step 3 to 7
    Step 3:     If call==0 then display ROOT
    Step 4:     Call gotoxy() with col and row+1 as arguments
    Step 5:     Display data of t
    Step 6:     call display(t→left,col-wid,row+2,wid++,call++);
    Step 7:     call display(t→right,col+wid,row+2,wid/2,call+1);
```

<u>Main() function</u>

```
    Step 1:     Start
    Step 2:     Display list of operations for the user to choose from
                   1. Insert
                   2. Delete
                   3. Find
                   4. Findmin
                   5. Findmax
                   6. Display
                   7. Makeempty
```

```
                8. Exit
Step 3:     Read the choice from the user choice
Step 4:     If choice==1 then read the data to be inserted as c and call
            the functions insert(c,t) and display (t,40,2,16,call)
Step 5:     If choice==2 then read the data to be deleted as c
            If t==NULL, then display “tree is empty”
            Else call delete(c,t) and display (t,40,2,16,call)
Step 6:     If choice==3 then read data to be searched as c
            If t==NULL then display “tree is empty”
            Else call find(c,t)
Step 7:     If choice==4 then check whether tree is empty or not.
            If tree is not empty then call findmin() and Assign it to r
Step 8:     If choice==5 then check whether tree is empty or not.
            If tree is not empty then call findmax(t) and Assign to r
Step 9:     If choice==6 then check whether tree is empty or not.
            If tree is not empty then call display(t,40,2,16,call)
Step 10:    If choice==7 then call makeempty(t) and assign it to t.
            Display “Tree is empty”
Step 11:    If choice==7 then go to Step 12
Step 12:    Stop.
```

13. IMPLEMENTATION OF HASHING TECHNIQUE USING LINEAR PROBING

Aim

To create a hash table based on modulo division method using linear probing method, in C program

Algorithm

```
Step 1:     Define max as 10 as shown below
            #define max 10
```

Create(int num) – Function to create and perform hash function

```
Step 1:     Assign the num%10 to key=> key=num%10
Step 2:     Return the value of key
```

Iprobe(int a[max],int key, int num)- Function to insert an element in to hash table

```
Step 1:     Initialise flag to 0
Step 2:     If a[key]==-1 then store num at a[key] position in the array
            else go to Step 3
Step 3:     Perform Step 4 untili<max
Step 4:     Find how many position are filled in the table using the
            Condition
            If(a[i]!=-1)
                    Count++;
            i++;
Step 5:     If count==max then display "Hash table is full"
Step 6:     for(i=key+1;i<max;i++) perform Step 7
Step 7:     If a[i]==-1 perform Step 8
Step 8:
            8.1:  Assign num to a[i]
            8.2:  Assign 1 to Flag and break the loop
Step 9:     If i==max perform Step 10
Step 10:    Initialise i to 0 and perform Step 8 until i<max
```

<u>Void display(int a[max])</u>

```
Step 1:     Display the contents in the hash table using for loop as
            follows:
                  for(i=0;i,max;i++)
                  printf("\n\t\t%d\t%d",i,a[i]);
```

<u>main() function</u>

```
Step 1:     Start
Step 2:     Intialise all elements of hashtable as -1 by using for loop as
            For(i=0;i<max;i++)
            a[i]=-1;
Step 3:     Read the number from the user as num
Step 4:     Call the function create() with num as argument and
            assign it to key
Step 5:     Call the function lprobe() with a, key and num as arguments
Step 6:     Call the function disply() with a as argument
Step 7:     Perform steps 3,4,5 and 6 until user wants to continue
Step 8:     Stop
```

14. IMPLEMENTATION OF DIJKSTRA'S ALGORITHM

Aim

To write a C program to find minimum distance from source vertex to all other vertices using Dijkstra's algorithm

Algorithm

```
Structure
Step 1:    Define a structure edge as follows
           Struct edge
           {
                Int vs,vd,wt;
           }a[20];
           Vs- Source vertex
           Vd-Destination vertex
           Wt- Weightage
```

Void di(int n)- Function to find shortest distance from single node to all nodes.

```
Step 1:    Make all the vertices unvisited using the for loop
           For(i=1;i<=n;i++)
           Str[j]=0;
Step 2:    Create edges for each non-zero element in the given
           adjacency matrix.  Edge should be created with source
           vertex vs, destination vertex vd, and weightage wt.
Step 3:    Read the starting vertexfrom the user as s and make it visited
           St[s]=1
```

```
               Let the predecessor of s as 1 and the distance be 0
Step 4:        Make the other nodes distance be 9999 and their
               predecessor as 0
               for(i=0;i<=n;i++)
               {
                   If(i!=s)
                   {
                       Pred[i]=0;
                       Dist[i]=9999;
                   }
               }
Step 5:        Find the distance from the vertex which is visited to its
               adjacent vertex using the for loop
               for(i=1;i<=k;i++)
               {
                   If((st[a[i].vs==1)&&(st[a[i].vd]==0))
                   {
                   M[++t]=(a[i].wt+dist[a[i].vs]);
                   }
               }
Step 6:        Consider the vertex with smallest distance as visited and
                its predecessor be vs and its distance be shortest distance
               between vs and vd
               for(i=1;i<=k;i++)
```

```
            {
            If((st(a[i].vs]==1)&&(st[a[i].vd]==0))
            {
            If((a[i].wt+dist[a[i].vs])==sm)
            {
            St[a[i].vd=1;
            Pred[a[i].vd]=a[i].vs;
            Dist[a[i].vd]=a[i].wt+dist[a[i].vs];
            }
            }
            }
Step 7:     Repeat steps 4,5 and 6 until all nodes are visited
Step 8:     Print the predecessor value of all nodes and their
            corresponding distance values.
```

<u>Main() function</u>

```
Step 1:     Start
Step 2:     Read no.of vertices from the user as n
Step 3:     Read the adjacentcy matrix from the user as g[i][j]
Step 4:     Call the function dj() with n as argument
Step 5:     Stop
```

15. DFS & BFS

Aim

Write an algorithm and corresponding program for depth first search and breadth first search

Algorithm

```
Step 1:     DES(v)      - v is the vertex
Step 2:     mark[v]:visited
Step 3:     for each vertex w on L[u] do
Step 4:     if mark – reverse call to DFS
Step 5:     DFS[w] – reverse call to DFS
        BFS search for each vertex v that are visited
Step 1:     BSF(v)
Step 2:     Mark[v]
Step 3:     equal(v,q)
Step 4:     Repeat step while not empty(a) do
Step 5:     x=front(q)
Step 6:     dequeue(q)
Step 7:     for each vertex adjacent to x do
Step 8:     if mark[y]=unvisited then
Step 9:     mark[s]=visited
Step 10:    enqueue(y,q)
Step 11:    insert((x,y),1)
```