Table of Content

BEYOND THE SYLLABUS

19.Program for List implementation through Array
20.Program for List implementation using dynamic memory allocation
21.Program for Implementation of Quick Sort Algorithm.

Experiment No. 1

Title: Program for Addition & Multiplication of 2D array.

Matrix Addition: The usual matrix addition is defined for two matrices of the same dimensions. The sum of two *m* × *n* (pronounced "m by n") matrices **A** and **B**, denoted by **A** + **B**, is again an *m* × *n* matrix computed by adding corresponding elements:[1][2]

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

For example:

Matrix Mutiplication: Arithmetic process of multiplying numbers (solid lines) in row *i* in matrix **A** and column *j* in matrix **B**,*then* adding the terms (dashed lines) to obtain entry *ij*in the final matrix.

If **A** is an *n* × *m* matrix and **B** is an *m* × *p* matrix,

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

the **matrix product AB** (denoted without multiplication signs or dots) is defined to be the *n* × *p* matrix[3][4][5][6]

$$\mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

.

Code:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
   int i,j,c,r,k;
   int a[20][20],b[20][20],ma[20][20],ms[20][20];
   int mm[20][20];
   clrscr();
   printf("\n\t\tINPUT:");
   printf("\n\t\t------");
   printf("\n\t\tEnter the value for row and column:  ");
   scanf("%d%d",&c,&r);
   printf("\n\t\tEnter the value for matrix A\n");
   for(i=0;i<c;i++)
     {
      for(j=0;j<r;j++)
       {
        scanf("%d",&a[i][j]);
       }
      printf("\n");
     }
   printf("\n\t\tEnter the value for matrix B\n");
   for(i=0;i<c;i++)
     {
      for(j=0;j<r;j++)
       {
         scanf("%d",&b[i][j]);
       }
      printf("\n");
     }
   for(i=0;i<c;i++)
     {
      for(j=0;j<r;j++)
        {
         ma[i][j]=a[i][j]+b[i][j];
         ms[i][j]=a[i][j]-b[i][j];
        }
     }
   for(i=0;i<c;i++)
     {
      for(j=0;j<r;j++)
        {
          mm[i][j]=0;
          for(k=0;k<c;k++)
```

```
            {
              mm[i][j] +=a[i][k]*b[k][j];
            }
        }
    }
  printf("\n\t\tOUTPUT:");
  printf("\n\t\t-------");
  printf("\n\t\tThe addition matrix is:\n");
  for(i=0;i<c;i++)
    {
      for(j=0;j<r;j++)
        {
         printf("\t\t%d",ma[i][j]);
        }
      printf("\n");
    }

  printf("\n\t\tThe multiplication matrix is:\n");
  for(i=0;i<c;i++)
    {
      for(j=0;j<r;j++)
        {
          printf("\t\t%d",mm[i][j]);
        }
      printf("\n");
    }
  getch();
}
```

INPUT:

Enter the value for row and column: 2   2
Enter the value for matrix A
4   3
6   2
Enter the value for matrix B
8   1
0   5

OUTPUT:

The addition matrix is:
        12   4
        6   7
The multiplication matrix is:

```
32  19
48  16
```

Applications: Expression evaluation and syntax parsing

Matrix multiplication is usually used in graphics initially (scalings, translations, rotations, etc). Then there are more in-depth examples such as counting the number of **walks** between nodes in a graph using the adjacency graph's power.

Experiment No. 2

Title: Pseudo code and Program for Transpose 2DArray

This c program prints transpose of a matrix. It is obtained by interchanging rows and columns of a matrix. For example if a matrix is

1 2
3 4
5 6

then transpose of above matrix will be

1 3 5
2 4 6

When we transpose a matrix then the order of matrix changes, but for a square matrix order remains same.

Program in C:

```
#include <stdio.h>

int main()
{
  int m, n, c, d, matrix[10][10], transpose[10][10];

  printf("Enter the number of rows and columns of matrix ");
  scanf("%d%d",&m,&n);
  printf("Enter the elements of matrix \n");

  for( c = 0 ; c < m ; c++ )
  {
    for( d = 0 ; d < n ; d++ )
    {
      scanf("%d",&matrix[c][d]);
    }
  }

  for( c = 0 ; c < m ; c++ )
  {
    for( d = 0 ; d < n ; d++ )
    {
      transpose[d][c] = matrix[c][d];
    }
  }

  printf("Transpose of entered matrix :-\n");

  for( c = 0 ; c < n ; c++ )
```

```
  {
    for( d = 0 ; d < m ; d++ )
    {
      printf("%d\t",transpose[c][d]);
    }
    printf("\n");
  }

  return 0;
}
```

The **array implementation** aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, stack[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack. The stack itself can therefore be effectively implemented as a two-element structure in C.

Applications: On a computer, one can often avoid explicitly transposing a matrix in memory by simply accessing the same data in a different order. For example, software libraries for linear algebra, such as BLAS, typically provide options to specify that certain matrices are to be interpreted in transposed order to avoid the necessity of data movement.

Experiment No. 3

Title: Pseudo code and Program for Stack implementation through Array

A **Stack** is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only three fundamental operations: *push*, *pop* and *stack top*. The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed). The stack top operation gets the data from the top-most position and returns it to the user without deleting it. The same underflow state can also occur in stack top operation if stack is empty.

Pseudo-code:

```
STACK-EMPTY(S)
if top[S] = 0
return true
else return false

PUSH(S, x)
top[S] <- top[S] + 1
S[top[S]] <- x

POP(S)
if STACK-EMPTY(S)
then error ”underflow”
else top[S] <- top[S] – 1
return S[top[S] + 1]
```

Program in C:

```
#include <stdio.h>

#include<ctype.h>

# define MAXSIZE 200;

int stack[MAXSIZE];

int top;        //index pointing to the top of stack

void main()
```

```
{
void push(int);
int pop();
int will=1,i,num;
clrscr();
while(will ==1)
{ printf("MAIN MENU:
      1.Add element to stack
      2.Delete element from the stack");
scanf("%d",&will);
switch(will)
{
case 1:
      printf("Enter the data... ");
      scanf("%d",&num);
      push(num);
      break;
case 2: i=pop();
      printf("Value returned from pop function is  %d ",i);
      break;
default: printf("Invalid Choice . ");
}
printf("Do you want to do more operations on Stack ( 1 for yes, any other key to exit) ");
scanf("%d" , &will);
} //end of  outer while
```

```
}               //end of main
void push(int y)
{
if(top>MAXSIZE)
      {
      printf("STACK FULL");
      return;
      }
else
      {
      top++;
      stack[top]=y;
      }
}
int pop()
{
int a;
if(top<=0)
      {
      printf("STACK EMPTY");
      return 0;
      }
else
      {
      a=stack[top];
```

```
    top--;
    }
return(a);
}
```

The **array implementation** aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, stack[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack. The stack itself can therefore be effectively implemented as a two-element structure in C.

Applications: Expression evaluation and syntax parsing

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines.

Experiment No. 4

Title: Pseudo code and Program for Queue implementation through Array

A **queue** is a particular kind of data structure in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Pseudo-code:

```
ENQUEUE(Q, x)
Q[tail[Q]] <- x
if tail[Q] = length[Q]
then tail[Q] <- 1
else tail[Q] <- tail[Q] + 1

DEQUEUE(Q)
x <- Q[head[Q]]
if head[Q] = length[Q]
then head[Q] <- 1
else head[Q] <- head[Q] + 1
return x
```

Program in C:

```
#include <stdio.h>

#include<ctype.h>

# define MAXSIZE 200

int q[MAXSIZE];

int front, rear;

void main()

{

void add(int);

int del();
```

```
int will=1,i,num;
front =0;
rear = 0;
clrscr();
printf("Program for queue demonstration through array");

while(will ==1)
{
printf("MAIN MENU:
      1.Add element to queue
      2.Delete element from the queue");
scanf("%d",&will);
switch(will)
{
case 1:
      printf("Enter the data... ");
      scanf("%d",&num);
      add(num);
      break;
case 2: i=del();
      printf("Value returned from delete function is  %d ",i);
      break;
default: printf("Invalid Choice ... ");
}
printf(" Do you want to do more operations on Queue ( 1 for yes, any other key to exit) ");
```

```
scanf("%d" , &will);
}
}
void add(int a)
{
if(rear>MAXSIZE)
        {
        printf("QUEUE FULL");
        return;
        }
else
        {
        q[rear]=a;
        rear++;
        printf(" Value of rear = %d and the value of front is %d",rear,front);
        }
}
int del()
{
int a;
if(front == rear)
        {
        printf("QUEUE EMPTY");
        return(0);
        }
```

```
else
    {
    a=q[front];
    front++;
    }
    return(a);
}
```

Applications:

1) Serving requests of a single shared resource (printer, disk, CPU), transferring data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO,sockets.
2) Call centre phone systems will use a queue to hold people in line until a service representative is free.
3) Buffers on MP3 players and portable CD players, iPod playlist. Playlist for jukebox: add songs to the end, play from the front of the list.
4) When programming a real-time system that can be interrupted (e.g., by a mouse click or wireless connection), it is necessary to attend to the interrupts immediately, before proceeding with the current activity. If the interrupts should be handles in the same order they arrive, then a FIFO queue is the appropriate data structure.

Experiment No. 5

Title: Pseudo code and Program for Circular Queue implementation through Array

A circular queue is a Queue but a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insert ion and deletion are totally independent. Which means that you don't have to worry about an interrupt handler trying to do an insertion at the same time as your main code is doing deletion.

Pseudo code:

```
add_circular( item,queue,rear,front)
            { rear=(rear+1)mod n;
              if (front == rear )
             then print " queue is full "
            else {
                  queue [rear]=item;}
             }
delete_circular (item,queue,rear,front)
{
if (front = = rear)
   print ("queue is empty");
else
  {
    front= front+1;
   item= queue[front];
  }
}
```

Program in C:

```
#include <stdio.h>
#include<ctype.h>
# define MAXSIZE 200
int cq[MAXSIZE];
int front,rear;
void main()
{
void add(int,int [],int,int,int);
int del(int [],int ,int ,int );
int will=1,i,num;
front = 1;
rear = 1;
clrscr();
printf("Program for Circular Queue demonstration through array");
while(will ==1)
{
printf("MAIN MENU:
      1.Add element to Circular Queue
      2.Delete element from the Circular Queue");
scanf("%d",&will);
switch(will)
{
case 1:
      printf("Enter the data... ");
```

```
        scanf("%d",&num);
        add(num,cq,MAXSIZE,front,rear);
        break;
case 2: i=del(cq,MAXSIZE,front,rear);
        printf("Value returned from delete function is  %d ",i);
        break;
default: printf("Invalid Choice . ");
}
printf(" Do you want to do more operations on Circular Queue ( 1 for yes, any other key to exit) ");
scanf("%d" , &will);
} //end of  outer while
}               //end of main


void add(int item,int q[],int MAX,int front,int rear)
{
rear++;
rear= (rear%MAX);
if(front ==rear)
        {
        printf("CIRCULAR QUEUE FULL");
        return;
        }
else
        {
```

```
        cq[rear]=item;
        printf("Rear = %d    Front = %d ",rear,front);
        }
}
int del(int q[],int MAX,int front,int rear)
{
int a;
if(front == rear)
        {
        printf("CIRCULAR STACK EMPTY");
        return (0);
        }
else
        {
        front++;
        front = front%MAX;
        a=cq[front];
        return(a);
        printf("Rear = %d    Front = %d ",rear,front);
        }
}
```

Applications:

Linear queues require much less programming logic as (you're right) the limitation of queue-size is really about all you really need to know. However, circular queues provide much more flexibility than a linear queue. Here's a couple of examples of flexibility available with a circular queue that aren't easily implementable with a linear queue:
1. Memory allocations...(assuming no pointers)

A. (Linear Queue):
i. Allocate new, larger block of memory reflective of the entire (existing) queue and the queue-size additions.
ii. Copy the initial block of memory (i.e. the entire existing queue) to the newly allocated block of memory.
iii Reset the "high-water" variable indicating the new size of the queue.

B. (Circular Queue):
i. Allocate ONLY the new object or structure-type required.
ii. Insert the new allocation ANYWHERE you want in the queue.

Typically circular queues implement a linked-list design to manage the queue. This allows you to "insert" an artifact (e.g. object or data-structure) anywhere you like into a queue. This is very difficult within a linear queue (depending on the reference implementation). Additionally memory management becomes much easier to control and predominantly more efficient implementing a circular queue.

2. Queue Referencing...
A. (Linear Queue):
i. Reference the queue by indicies (e.g. myQueue[2], myQueue[nCurrentObject], myNumberQueue[12] = 10) but here's the double-edged blade (myFocalObject = myQueue[12]).

B. (Circular Queue):
i. Reference the queue by function (e.g. GetNext(..), GetPrev(..)) or by primary data artifact (e.g. GetByName(..), GetByID(..)) to retrieve specific artifact within the queue.

The principle advantages in using a circular queue here is no explicit reference to queue indices and the ability to search by queue element data in a way that obfuscates the queue itself. This would be very valuable if your software will be enhanced in the future or part of a larger software design.

Note: If ease-of-implementation and limited software life is more important, then use the linear queues. They'll be easier to write, read and if the software you're designing has a limited lifetime and/or is compartmentalized into a small enough function, you probably don't want to "over-engineer" your code.

Experiment No. 6

Title: Pseudo code and Implementation of Stack using dynamic memory allocation

A stack is an ordered list where insertion and deletion take place at one end called top of the stack (stack pointer)

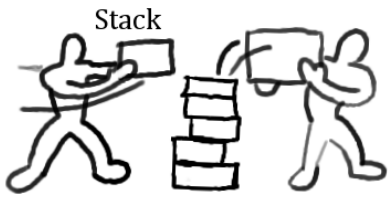
A stack is a basic data structure that is used all throughout programming. The idea is to think of your data as a stack of plates or books where you can only take the top item off the stack in order to remove things from it.

A stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data.

Stack<item-type> Operations

push(*new-item*:item-type)
Adds an item onto the stack.
top():item-type
Returns the last item pushed onto the stack.
pop()
Removes the most-recently-pushed item from the stack.
is-empty():Boolean
True if no more items can be popped and there is no top item.
is-full():Boolean
True if no more items can be pushed.
get-size():Integer
Returns the number of elements on the stack.

All operations except get-size() can be performed in *O*(1) time. get-size() runs in at worst *O*(*N*).



Pseudo code:

```
push()
{
NODE* temp;
int item;
temp = (NODE*)malloc (sizeof(NODE));
```

```
printf("Enter the item to be inserted-> ");
scanf("%d",&item);
temp->data = item;
temp->next = NULL;
if(top == NULL)
top = temp;
else
{
temp->next = top;
top = temp;
}
}

 pop()
{
int item;
NODE* temp;
if(top == NULL)
printf("\n***Stack is empty***\n");
else
{
temp = top;
top = top->next;
printf("\n\tDeleted item is-> %d\n",temp->data);
free(temp);
}
}

display()
{
NODE* temp;
if(top == NULL)
{
printf("\n***Stack is empty***\n");
return;
}
else
{
temp = top;
printf("\nThe list is-> ");
while(temp != NULL)
{
printf("\t%d",temp->data);
temp = temp->next;
}
printf("\n");
```

```
}
}
```

program in C:

```
#include<stdio.h>
#include<malloc.h>
#define MAX 10
int top = -1;
int Stack[MAX] = {0};
typedef struct list
{
  int data;
  struct list *next;
} LIST;

int main()
{
  int push(int);
  int pop();
 LIST *Stack_top = NULL;
 int stkpush(LIST **, int);
  int stkpop(LIST **);
  int data;
  push(10);
  push(11);
  push(12);
  push(13);
  push(14);
  push(15);
  push(16);
  push(17);
  push(18);
  push(19);
  push(20);
  push(11);
  push(12);
 stkpop(&Stack_top);
  printf("\n");
  stkpush(&Stack_top,10);
  stkpush(&Stack_top,11);
```

```
  stkpush(&Stack_top,12);
  stkpush(&Stack_top,13);
  stkpush(&Stack_top,14);
  stkpush(&Stack_top,15);
  while( data = stkpop(&Stack_top) )
  {
    printf("\t%d",data);
  }

  printf("\n stack2 output:");
  while(data = pop())
  {
    printf("\t%d",data);
  }


  return 0;
}
int push(int data)
{
  if(top == (MAX-1))
  {
    printf("\nStack is full");

  }
  else
  {
    Stack[++top] = data;
  }

  return 0;
}

int pop()
{
  int retv;

  if(top == -1)
  {
    printf("\n empty stack");
```

```
    retv = 0;
  }
  else
  {
    retv = Stack[top--];
  }

  return retv;
}

int stkpush(LIST **top, int data)
{
  LIST *temp;

  if( temp = (LIST *)malloc(sizeof(LIST)) )
  {
    temp->data = data;
    temp->next = *top;
    *top = temp;
    return 1;
  }
  else
  {
    return 0;
  }
}


int stkpop(LIST **top)
{
  LIST *temp;
  int retv = 0;

  if(*top == NULL)
  {
    printf("\n empty stack");

  }
  else
  {
```

```
    temp = *top;
    retv = (*top)->data;
    (*top) = (*top)->next;
    free(temp);
  }
  return retv;
}
```

Applications:

The **linked-list** implementation is equally simple and straightforward. In fact, a simple singly linked list is sufficient to implement a stack -- it only requires that the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node.

Unlike the array implementation, our structure typedef corresponds not to the entire stack structure, but to a single node:

Experiment No. 7

Title: Pseudo code and Program for Queue implementation Using Dynamic Memory allocation

For Queue implementation using dynamic memory allocation operations, we can maintain two pointers – qfront and qback as we had done for the case of array implementation of queues.
For the Enqueue operation, the data is first loaded on a new node. If the queue is empty, then after insertion of the first node, both qfront and qback are made to point to this node, otherwise, the new node is simply appended and qback updated.
In Dequeue function, first of all check if at all there is any element. If there is none, we would have *qfront as NULL, and so report queue to be empty, otherwise return the data element, update the *qfront pointer and free the node. Special care has to be taken if it was the only node in the queue.

Pseudo code:

```
struct node
{
 int value;
 struct node *next;
}
struct node *queue, *front, *rear;

insert(int value)
{
 struct node *new;

 new = (struct node *)malloc(sizeof(node));
 new->value = value;
 new->next = NULL;

 if(front == NULL)
   {
    queue = new;
    front = rear = queue;
   }
 else
   {
    rear->next = new;
    rear = new;
   }
}
```

```
delete( )
{
 int delval = 0;
 if(front == NULL) printf("queue empty");
   else
     {
      delval = front->value;
      if(front->next == NULL)
       {
        free(front);
        queue = front = rear = NULL;
       }
      else
       {
        front = front->next;
        free(queue);
        queue = front;
       }
     }
}
```

Program in C:

```
#include<stdio.h>

#include<conio.h>

struct node

{

int data;

struct node  *link;

}     ;

struct node *front, *rear;

void main()

{

int wish,will,a,num;

void add(int);

wish=1;
```

```
clrscr();
front=rear=NULL;
printf("Program for Queue as Linked List demo..");
while(wish == 1)
    {
    printf("Main Menu 1.Enter data in queue \n 2.Delete from queue");
    scanf("%d",&will);
    switch(will)
        {
        case 1:
            printf("Enter the data");
            scanf("%d",&num);
            add(num);
            //display();
            break;
        case 2:
            a=del();
            printf("Value returned from front of the queue is %d",a);
            break;
        default:
            printf("Invalid choice");
        }
    printf("Do you want to continue, press 1");
    scanf("%d",&wish);
    }
```

```
getch();
}
void add(int y)
{
struct node *ptr;
ptr=malloc(sizeof(struct node));
ptr->data=y;
ptr->link=NULL;
if(front ==NULL)
        {
        front = rear= ptr;
        }
else
        {
        rear->link=ptr;
        rear=ptr;
        }
}
int  del()
{
int num;
if(front==NULL)
        {
        printf("QUEUE EMPTY");
        return(0);
```

```
        }
    else
        {
        num=front->data;
        front = front->link;
        printf(" Value returned by delete function is %d ",num);
        return(num);
        }
    }
```

Application:

Features of queue:

1. A list structure with two access points called the **front** and **rear.**
2. All insertions (enqueue) occur at the rear and deletions (dequeue) occur at the front.
3. Varying length (dynamic).
4. Homogeneous components
5. Has a First-In, First-Out characteristic (FIFO)

Experiment No. 8

Title: Pseudo code and Program for implementation of Circular Queue using dynamic memory allocation.

A circular queue is a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insertion and deletion are totally independent, which means that you don't have to worry about an interrupt handler trying to do an insertion at the same time as your main code is doing a deletion.

Pseudo code:

```
En_queue(int *arr,  int data, int *front, int *rear)
{
If(*rear==size-1)
*rear=0;
Else
(*rear)++;
Arr[*rear]=data;
If(front==-1)
*front=0;
}
De_queue(int *arr,  int data, int *front, int *rear)
{
Data=arr[*front];
If(*front==*rear)
*front=*rear=-1;
Else if(*front==size-1)
*front=0;
Else
(*front)++;
Return(data)l
}
```

Program in C:

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

struct node
```

```
{
int info;
struct node *link;
}*rear=NULL;

main()
{
int choice;
while(1)
{
printf("1.Insert \n");
printf("2.Delete \n");
printf("3.Display\n");
printf("4.Quit\n");
printf("Enter your choice : ");
scanf("%d",&choice);

switch(choice)
{
case 1:
insert();
break;
case 2:
del();
break;
case 3:
display();
break;
case 4:
exit();
default:
printf("Wrong choice\n");
}/*End of switch*/
}/*End of while*/
}/*End of main()*/

insert()
{
int num;
struct node *q,*tmp;
```

```
printf("Enter the element for insertion : ");
scanf("%d",&num);
tmp= malloc(sizeof(struct node));
tmp->info = num;
if(rear == NULL) /*If queue is empty */
{
rear = tmp;
tmp->link = rear;
}
else
{
tmp->link = rear->link;
rear->link = tmp;
rear = tmp;
}
}/*End of insert()*/

del()
{
struct node *tmp,*q;
if(rear==NULL)
{
printf("Queue underflow\n");
return;
}
if( rear->link == rear ) /*If only one element*/
{
tmp = rear;
rear = NULL;
free(tmp);
return;
}
q=rear->link;
tmp=q;
rear->link = q->link;
printf("Deleted element is %d\n",tmp->info);
free(tmp);
}/*End of del()*/

display()
```

```
{
struct node *q;
if(rear == NULL)
{
printf("Queue is empty\n");
return;
}
q = rear->link;
printf("Queue is :\n");
while(q != rear)
{
printf("%d ", q->info);
q = q->link;
}
printf("%d\n",rear->info);
}
```

Applications:

As above experiment number 3**.**

Experiment No. 9

Title: Pseudo code and program for Implementation of Binary tree.

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side.

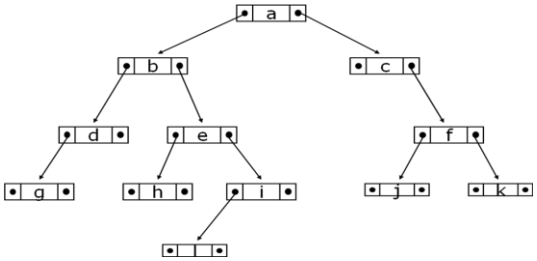
A binary tree is composed of zero or more nodes each node contains:

1. A value (some sort of data item)
2. A reference or pointer to a left child (may be null), and
3. A reference or pointer to a right child (may be null)

A binary tree may be *empty* (contain no nodes) If not empty, a binary tree has a root node.

Every node in the binary tree is reachable from the root node by a *unique* path.

A node with neither a left child nor a right child is called a leaf. In some binary trees, only the leaves contain a value.



Pseudo code:

```
struct NODE
{
 struct NODE *left;
 int value;
 struct NODE *right;
}

create_tree( struct NODE *curr, struct NODE *new )
{
 if(new->value <= curr->value)
  {
   if(curr->left != NULL)
     create_tree(curr->left, new);
   else
     curr->left = new;
  }
 else
  {
   if(curr->right != NULL)
```

```
    create_tree(curr->right, new);
   else
    curr->right = new;
  }
}
```

Program in C:

```
#include"stdio.h"
#include"conio.h"
#include"alloc.h"
#include"stdlib.h"

struct btree
{
int n;
struct btree *left;
struct btree *right;
}typedef btree,node;
btree *root,*ptr,*prev,*bfor,*x,*y;
int num;
char ch;
void main()
{
int c;
void create(node *);
void print(node *);
void search(node *);
void insert(node *);
void modify(node *);
void delet(node *);
while(c!=7)
{
clrscr();
printf("1. CREATE");
printf("2. PRINT");
printf("3. SEARCH");
printf("4. INSERT");
printf("5. MODIFY");
printf("6. DELETE");
printf("7. EXIT");
printf("Enter your choice : ");
scanf("%d",&c);
switch(c)
{
case 1 :
```

```
{
root=(node *)malloc(sizeof(node));
printf("Enter number : ");
scanf("%d",&root->n);
ptr=root;
root->left=root->right=NULL;
printf("Enter more(y/n)? ");
ch=getch();
create(root);
break;
}
case 2 :
{
print(root);
getch();
break;
}
case 3 :
{
printf("");
search(root);
getch();
break;
}
case 4 :
{
printf("");
insert(root);
getch();
break;
}
case 5 :
{
printf("");
modify(root);
getch();
break;
}
case 6 :
{
printf("");
delet(root);
getch();
break;
}
case 7 :
```

```
exit(0);

default:
{
printf("Invalid choice");
getch();
break;
}
}
}
getch();
}

void create(node *ptr)
{
while(ch == 'y')
{
printf("");
ptr=prev=root;
printf("Enter number : ");
scanf("%d",&num);
do
{
if(num <>n)
{
prev=ptr;
ptr=ptr->left;
}
else if(num > ptr->n)
{
prev=ptr;
ptr=ptr->right;
}
else
{
prev=NULL;
break;
}
}while(ptr);

if(prev)
{
ptr=(node *)malloc(sizeof(node));
ptr->n=num;
ptr->left=ptr->right=NULL;
if(ptr->n <>n)
```

```
{
prev->left=ptr;
if(prev == root)
root=prev;
}
if(ptr->n > prev->n)
{
prev->right=ptr;
ptr->left=ptr->right=NULL;
if(prev == root)
root=prev;
}
}
else
printf("'%d' is already present..",num);

printf("Enter more(y/n)? ");
ch=getch();
}
}

void print(node *ptr)
{
void inprint(node *);
void preprint(node *);
void postprint(node *);

if(!ptr)
{
printf("Tree is empty...");
return;
}
printf("Root is '%d'",root->n);
printf("INORDER : ");
inprint(root);
printf("PREORDER : ");
preprint(root);
printf("POSTORDER : ");
postprint(root);
}


void inprint(node *ptr)
{
if(!ptr)
return;
```

```
inprint(ptr->left);
printf("%2d ",ptr->n);
inprint(ptr->right);
return;
}

void preprint(node *ptr)
{
if(!ptr)
return;

printf("%2d ",ptr->n);
preprint(ptr->left);
preprint(ptr->right);
return;
}
void postprint(node *ptr)
{
if(!ptr)
return;

postprint(ptr->left);
postprint(ptr->right);
printf("%2d ",ptr->n);
return;
}

void search(node *ptr)
{
if(!ptr)
{
if(ptr == root)
{
printf("Tree is empty... You can't search anything...");
return;
}
}
printf("Enter the number to search : ");
scanf("%d",&num);

while(ptr)
{
if(ptr->n == num)
{
printf("Success... You found the number...");
```

```
return;
}
else
if(ptr->n < ptr="ptr-">right);
else
ptr=ptr->left;
}
printf("Tree don't contain '%d'...",num);
}

void insert(node *ptr)
{
if(!ptr)
{
printf("Tree is empty...First create & then insert... ");
return;
}
printf("");
ptr=prev=root;
printf("Enter number to be inserted : ");
scanf("%d",&num);

do
{
if(num <>n)
{
prev=ptr;
ptr=ptr->left;
}
else if(num > ptr->n)
{
prev=ptr;
ptr=ptr->right;
}
else
{
prev=NULL;
break;
}
}
while(ptr);

if(prev)
{
ptr=(node *)malloc(sizeof(node));
ptr->n=num;
```

```
ptr->left=ptr->right=NULL;

if(ptr->n <>n)
{
prev->left=ptr;
if(prev == root)
root=prev;
}
if(ptr->n > prev->n)
{
prev->right=ptr;
ptr->left=ptr->right=NULL;
if(prev == root)
root=prev;
}
printf("'%d' is inserted...",num);
}
else
printf("'%d' is already present...",num);
return;
}

void modify(node *ptr)
{
int mod;
if(!ptr)
{
if(ptr == root)
{
printf("Tree is empty... You can't modify anything...");
return;
}
}
printf("Modification of particular number can't create a binary
tree...");
getch();
printf("Enter the number to get modified : ");
scanf("%d",&num);
prev=ptr;
while(ptr)
{
if(ptr->n == num)
{
x=ptr;
mod=x->n;
printf("Then enter new number : ");
```

```
scanf("%d",&num);
bfor=ptr=root;
while(ptr)
{
if(ptr->n == num)
{
printf("'%d' already present...Modification denied...,",num);
return;
}
else if(ptr->n < bfor="ptr;" ptr="ptr-">right;
}
else
{
bfor=ptr;
ptr=ptr->left;
}
}
if(x==root)
{
y=x->right;
root=y;
while(y->left)
y=y->left;
y->left=x->left;
free(x);
}
else if(!(x->left) && !(x->right))
{
if(prev->left == x)
{
prev->left=NULL;
free(x);
}
else if(prev->right == x)
{
prev->right=NULL;
free(x);
}
}
else if(!(x->left))
{
if(prev->left == x)
{
prev->left=x->right;
free(x);
}
```

```
else if(prev->right == x)
{
prev->right=x->right;
free(x);
}
}
else if(!(x->right))
{
if(prev->left == x)
{
prev->left=x->left;
free(x);
}
else if(prev->right == x)
{
prev->right=x->left;
free(x);
}
}
else
{
y=x->right;
while(y->left)
y=y->left;
y->left=x->left;
if(prev->left == x)
prev->left=y;
else if(prev->right == x)
prev->right=y;
free(x);
}
ptr=(node *)malloc(sizeof(node));
ptr->n=num;
ptr->left=ptr->right=NULL;
if(ptr->n <>n)
{
bfor->left=ptr;
if(bfor == root)
root=bfor;
}
if(ptr->n > bfor->n)
{
bfor->right=ptr;
ptr->left=ptr->right=NULL;
if(bfor == root)
root=bfor;
```

```
}
printf("'%d' is modified by '%d'",mod,ptr->n);
return;
}
else
if(ptr->n < prev="ptr;" ptr="ptr-">right;
}
else
{
prev=ptr;
ptr=ptr->left;
}
}
printf("Tree don't contain '%d'...",num);
}

void delet(node *ptr)
{
if(!ptr)
{
if(ptr == root)
{
printf("Tree is empty... You can't delete anything...");
return;
}
}
printf("Enter the number to get deleted : ");
scanf("%d",&num);
prev=ptr;

while(ptr)
{
if(ptr->n == num)
{
if(ptr==root)
{
x=ptr->right;
root=x;
while(x->left)
x=x->left;
x->left=ptr->left;
free(ptr);
printf("'%d' is deleted...",num);
return;
}
else if(!(ptr->left) && !(ptr->right))
```

```
{
if(prev->left == ptr)
prev->left=NULL;
else
prev->right=NULL;
free(ptr);
printf("'%d' is deleted...",num);
return;
}
else if(!(ptr->left))
{
if(prev->left == ptr)
{
prev->left=ptr->right;
free(ptr);
}
else if(prev->right == ptr)
{
prev->right=ptr->right;
free(ptr);
}
printf("'%d' is deleted...",num);
return;
}
else if(!(ptr->right))
{
if(prev->left == ptr)
{
prev->left=ptr->left;
free(ptr);
}
else if(prev->right == ptr)
{
prev->right=ptr->left;
free(ptr);
}
printf("'%d' is deleted...",num);
return;
}
else
{
x=ptr->right;
while(x->left)
x=x->left;
x->left=ptr->left;
if(prev->left == ptr)
```

```
prev->left=ptr->right;
else if(prev->right == ptr)
prev->right=ptr->right;
free(ptr);
printf("'%d' is deleted...",num);
return;
}
}
else if(ptr->n < prev="ptr;" ptr="ptr-">right;
}
else
{
prev=ptr;
ptr=ptr->left;
}
}
printf("Tree don't contain '%d'...",num);
}
```

Applications:

Used in almost every high-bandwidth router for storing router-tables. Binary trees are used to construct max and min heaps to perform heap sort on an array. Binary tree having important role in language parsing. An example is the construction of the binary expression tree.

Experiment No. 10

Title : Pseudo code and Program for implementation of Tree Operations Inorder, preorder, postorder traversal.

Pseudo code:

```
Preorder (tree *root)
{
Visit root;
If (root−>left is not equals NULL) preorder (root−>left);
If (root−>right is not equals NULL) preorder (root−>right);
}
Inorder (tree *root)
{
If (root−>left is not equals NULL) inorder (root−>left);
Visit root;
If (root−>right is not equals NULL) inorder (root−>right);
}
Postorder (tree *root)
{
If (root−>left is not equals NULL) postorder (root−>left);
If (root−>right is not equals NULL) postorder (root−>right);
Visit root;
}
```

Program in C:

```
# include<stdio.h>
# include <conio.h>
# include <malloc.h>

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

void main()
{
```

```
void insert(struct node **,int);
void inorder(struct node *);
void postorder(struct node *);
void preorder(struct node *);
struct node *ptr;
int will,i,num;
ptr = NULL;
ptr->data=NULL;
clrscr();

printf("Enter the number of terms you want to add to the tree.");
scanf("%d",&will);
/* Getting Input */
for(i=0;i<will;i++)
      {
      printf("Enter the item");
      scanf("%d",&num);
      insert(&ptr,num);
      }

getch();
printf("INORDER TRAVERSAL");
inorder(ptr);
getch();
printf("PREORDER TRAVERSAL");
preorder(ptr);
getch();
printf("POSTORDER TRAVERSAL");
postorder(ptr);
getch();
}
void insert(struct node  **p,int num)
{


if((*p)==NULL)
{      printf("Leaf node created.");
      (*p)=malloc(sizeof(struct node));
      (*p)->left = NULL;
      (*p)->right = NULL;
```

```
        (*p)->data = num;
        return;
}
else
{       if(num==(*p)->data)
                {
                printf("REPEATED ENTRY ERROR VALUE REJECTED");
                return;
                }
        if(num<(*p)->data)
                {
                        printf("Directed to left link.");
                        insert(&((*p)->left),num);
                }
        else
                {
                printf("Directed to right link.");
                insert(&((*p)->right),num);
                }
}
return;
}


void inorder(struct node *p)
{
        if(p!=NULL)
                {
                inorder(p->left);
                printf("
Data :%d",p->data);
                inorder(p->right);
                }
        else
                return;
}


void preorder(struct node *p)
{
```

```
    if(p!=NULL)
            {
            printf("
Data :%d",p->data);
            preorder(p->left);
            preorder(p->right);
            }
    else
            return;
}


void postorder(struct node *p)
{
    if(p!=NULL)
            {
            postorder(p->left);
            postorder(p->right);
            printf("
Data :%d",p->data);
            }
    else
            return;
}
```

Applications:

Inorder traversal:It is particularly common to use an inorder traversal on a binary search tree because this will return values from the underlying set in order, according to the comparator that set up the binary search tree (hence the name). To see why this is the case, note that if *n* is a node in a binary search tree, then everything in *n* 's left subtree is less than *n*, and everything in *n* 's right subtree is greater than or equal to *n*. Thus, if we visit the left subtree in order, using a recursive call, and then visit *n*, and then visit the right subtree in order, we have visited the entire subtree rooted at *n* in order. We can assume the recursive calls correctly visit the subtrees in order using the mathematical principle of structural induction. Traversing in reverse inorder similarly gives the values in decreasing order.

Preorder traversal:Traversing a tree in preorder while inserting the values into a new tree is common way of making a complete *copy* of a binary search tree.

Experiment No. 11

Title: Pseudo code and Program for implementation of Binary Search Tree

A **binary search tree** (**BST**) is a node based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.

Generally, the information represented by each node is a **record** rather than a single data element. However, for sequencing purposes, nodes are compared according to their **keys** rather than any part of their their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

Pseudo Code:

```
bst_node *CreateANode(int val) {
bst_node *newnode;
newnode = malloc(sizeof(bst_node);
if( newnode == NULL) {
return NULL;
}
newnode->data = val;
newnode->right = newnode->left = NULL;
return newnode;}
```

Program in C:

```
#include<stdio.h>

#include<conio.h>
```

```
struct tree
{
   int data;
   struct tree *left;
   struct tree *right;
};
struct tree *create();
void preorder(struct tree *);
void inorder(struct tree *);
void postorder(struct tree *);
struct tree *create()
{
   struct tree *p,*root;
   int m,x;
   char s;
   root=(struct tree *)malloc(sizeof(struct tree));
   printf("\nenter the value of the main root");
   scanf("%d",&m);
   root->data=m;
   root->left=NULL;
   root->right=NULL;
   printf("\nenter n to stop creation of the binary search tree");
```

```
fflush(stdin);
scanf("%c",&s);
while(s!='n')
{
   p=root;
   printf("\nenter the value of the newnode");
   fflush(stdin);
   scanf("%d",&x);
   while(1)
   {
      if(x<p->data)
      {
         if(p->left==NULL)
         {
            p->left=(struct tree *)malloc(sizeof(struct tree));
            p=p->left;
            p->data=x;
            p->right=NULL;
            p->left=NULL;
            break;
         }
         else
```

```
            p=p->left;
        }
        else
        {
            if(p->right==NULL)
            {
                p->right=(struct tree *)malloc(sizeof(struct tree));
                p=p->right;
                p->data=x;
                p->right=NULL;
                p->left=NULL;
                break;
            }
            else
            p=p->right;
        }
    }
    printf("\nwant to continue");
    fflush(stdin);
    scanf("%c",&s);
}
return(root);
```

```
}
void preorder(struct tree *p)
{
   if(p!=NULL)
   {
      printf("%d ",p->data);
      preorder(p->left);
      preorder(p->right);
   }
}
void inorder(struct tree *p)
{
   if(p!=NULL)
   {
      inorder(p->left);
      printf("\t%d",p->data);
      inorder(p->right);
   }
}
void postorder(struct tree *p)
{
   if(p!=NULL)
```

```
    {
        postorder(p->left);
        postorder(p->right);
        printf("\t%d",p->data);
    }
}
void main()
{
    int h;
    struct tree *root;
    while(1)
    {
        printf("\nenter 1. for creation of the binary search tree");
        printf("\nenter 2. for preorder traversal");
        printf("\nenter 3. for inorder traversal");
        printf("\nenter 4. for postorder traversal");
        printf("\nenter 5. for exit");
        printf("\nenter your choice");
        scanf("%d",&h);
        switch(h)
        {
            case 1:
```

```
            root=create();
            break;
            case 2:
            preorder(root);
            break;
            case 3:
            inorder(root);
            break;
            case 4:
            postorder(root);
            break;
            case 5:
            exit(0);
            default:
            printf("\nentered a wrong choice");
        }
    }
}
```

Application :

Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries. Binary Search Tree structures can be used in cases where you want to minimize the number of node accesses .They is used extensively in operating system design.

Experiment No. 12

Title: Algorithm and Program for Searching Algorithm "Binary Search" .

A **Binary Search** or **half-interval search** algorithm finds the position of a specified value (the input "key") in an array sorted into order on the values of the key. At each stage, the algorithm compares the sought key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub array to the left of the middle element or, if the input key is greater, on the sub array to the right. If the array span to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

Algorithm is quite simple. It can be done either recursively or iteratively:

1. get the middle element;
2. if the middle element equals to the searched value, the algorithm stops;
3. otherwise, two cases are possible:
    1. searched value is less, than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    2. searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Now we should define, when iterations should stop. First case is when searched element is found. Second one is when subarray has no elements. In this case, we can conclude, that searched value doesn't present in the array.

Program in C:

```
#include <stdio.h>
#define TRUE  0
#define FALSE 1
int main(void) {
 int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 int left = 0;
 int right = 10;
 int middle = 0;
 int number = 0;
 int bsearch = FALSE;
 int i = 0;
 printf("ARRAY: ");
 for(i = 1; i <= 10; i++)
```

```
 printf("[%d] ", i);
printf("\nSearch for Number: ");
scanf("%d", &number);
while(bsearch == FALSE && left <= right) {
 middle = (left + right) / 2;
 if(number == array[middle]) {
  bsearch = TRUE;
  printf("** Number Found **\n");
 } else {
  if(number < array[middle]) right = middle - 1;
  if(number > array[middle]) left = middle + 1;
 }
}

if(bsearch == FALSE)
 printf("-- Number Not found --\n");
return 0;
}
```

Applications:

Binary search can be used to access ordered data quickly *when memory space is tight*. Suppose you want to store a set of 100.000 32-bit integers in a searchable, ordered data structure but you are not going to change the set often. You can trivially store the integers in a sorted array of 400.000 bytes, and you can use binary search to access it fast. But if you put them e.g. into a B-tree, RB-tree or whatever "more dynamic" data structure, you start to incur memory overhead. To illustrate, storing the integers in any kind of tree where you have left child and right child pointers would make you consume at least 1.200.000 bytes of memory (assuming 32-bit memory architecture). Sure, there are optimizations you can do, but that's how it works in general. Because it is very slow to update an ordered array (doing insertions or deletions), binary search is not useful when the array changes often.

Experiment No. 13

Title: Pseudo code and Program for implementation of Bubble sort.

Bubble sort, also known as **sinking sort**, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, it is not efficient for sorting large lists; other algorithms are better.

Pseudo code:

```
BubbleSort( int a[], int n)
Begin
for i = 1 to n-1
sorted = true
for j = 0 to n-1-i
if a[j] > a[j+1]
temp = a[j]
a[j] = a[j+1]
a[j+1] = temp
sorted = false
end for
if sorted
break from i loop
end for
End
```

Program in C:

```
#include <stdio.h>
void bubble_sort(int a[], int size);
int main(void) {
 int arr[10] = {10, 2, 4, 1, 6, 5, 8, 7, 3, 9};
 int i = 0;
 printf("before:\n");
 for(i = 0; i < 10; i++) printf("%d ", arr[i]);
 printf("\n");
 bubble_sort(arr, 10);
 printf("after:\n");
 for(i = 0; i < 10; i++) printf("%d ", arr[i]);
```

```
 printf("\n");
 return 0;
}
void bubble_sort(int a[], int size) {
 int switched = 1;
 int hold = 0;
 int i = 0;
 int j = 0;
 size -= 1;
 for(i = 0; i < size && switched; i++) {
  switched = 0;
  for(j = 0; j < size - i; j++)
   if(a[j] > a[j+1]) {
    switched = 1;
    hold = a[j];
    a[j] = a[j + 1];
    a[j + 1] = hold;
   }
 }
}
```

Application:

Bubble sort is (provably) the fastest sort available under a *very* specific circumstance. It originally became well known primarily because it was one of the first algorithms (of any kind) that was rigorously analyzed, and the proof was found that it was optimal under its limited circumstance.

Consider a file stored on a tape drive, and so little random access memory (or such large keys) that you can only load *two* records into memory at any given time. Rewinding the tape is slow enough that doing random access within the file is generally impractical -- if possible, you want to process records sequentially, no more than two at a time.

Back when tape drives were common, and machines with only a few thousand (words|bytes) of RAM (of whatever sort) were common, that was sufficiently realistic to be worth studying. That circumstance is now rare, so studying bubble sort makes little sense at all -- but even worse, the circumstance when it's optimal isn't taught anyway, so even when/if the right situation arose, almost nobody would *realize* it.

Experiment No. 14

Title: Pseudo code and Program for implementation of Selection Sort

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n^2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited. The algorithm works as follows:

1. Find the minimum value in the list.
2. Swap it with the value in the first position.
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

Effectively, the list is divided into two parts: the sublist of items already sorted, which is built up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Pseudo code:

```
SelectionSort(A)
// GOAL: place the elements of A in ascending order
1  n := length[A]
2  for i := 1 to n
3     // GOAL: place the correct number in A[i]
4     j := FindIndexOfSmallest( A, i, n )
5     swap A[i] with A[j]
      // L.I. A[1..i] the i smallest numbers sorted
6  end-for
7  end-procedure

FindIndexOfSmallest( A, i, n )
// GOAL: return j in the range [i,n] such
//       that A[j]<=A[k] for all k in range [i,n]
1  smallestAt := i ;
2  for j := (i+1) to n
3     if ( A[j] < A[smallestAt] ) smallestAt := j
      // L.I. A[smallestAt] smallest among A[i..j]
4  end-for
5  return smallestAt
6  end-procedure
```

Program in C:

```
#include <stdio.h>
void selection_sort(int a[], int size);
int main(void) {
 int arr[10] = {10, 2, 4, 1, 6, 5, 8, 7, 3, 9};
 int i = 0;
 printf("before:\n");
 for(i = 0; i < 10; i++) printf("%d ", arr[i]);
 printf("\n");
 selection_sort(arr, 10);
printf("after:\n");
 for(i = 0; i < 10; i++) printf("%d ", arr[i]);
 printf("\n");
 return 0;
}
void selection_sort(int a[], int size) {
 int i = 0;
 int j = 0;
 int large = 0;
 int index = 0;
 for(i = size - 1; i > 0; i--) {
  large = a[0];
  index = 0;
  for(j = 1; j <= i; j++)
   if(a[j] > large) {
    large = a[j];
    index = j;
   }
   a[index] = a[i];
   a[i] = large;
 }
}
```

Application:

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n^2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

Experiment No. 15

Title: Pseudo code and Program for implementation of Merge Sort.

Merge sort is an $O(n \log n)$ comparison-based sorting algorithm. Most implementations produce a stable sort, meaning that the implementation preserves the input order of equal elements in the sorted output. It is a divide and conquer algorithm.
Conceptually, a merge sort works as follows

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sublists of about half the size.
3. Sort each sublist recursively by re-applying the merge sort.
4. Merge the two sublists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than from two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge

Pseudo code:

```
mergesort(int a[], int low, int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
merge(a,low,high,mid);
}
return(0);
}

merge(int a[], int low, int high, int mid)
{
int i, j, k, c[50];
i=low;
j=mid+1;
k=low;
while((i<=mid)&&(j<=high))
{
if(a[i]<a[j])
```

```
{
c[k]=a[i];
k++;
i++;
}
else
{
c[k]=a[j];
k++;
j++;
}
}
while(i<=mid)
{
c[k]=a[i];
k++;
i++;
}
while(j<=high)
{
c[k]=a[j];
k++;
j++;
}
for(i=low;i<k;i++)
{
a[i]=c[i];
}
}
```

Program in C:

```
#include <stdio.h>
#include <stdlib.h>
#define MAXARRAY 10
void mergesort(int a[], int low, int high);
int main(void) {
 int array[MAXARRAY];
 int i = 0;
 /* load some random values into the array */
 for(i = 0; i < MAXARRAY; i++)
  array[i] = rand() % 100;
 /* array before mergesort */
 printf("Before    :");
```

```
 for(i = 0; i < MAXARRAY; i++)
  printf(" %d", array[i]);
 printf("\n");
 mergesort(array, 0, MAXARRAY - 1);
 /* array after mergesort */
 printf("Mergesort :");
 for(i = 0; i < MAXARRAY; i++)
  printf(" %d", array[i]);
 printf("\n");
 return 0;
}
void mergesort(int a[], int low, int high) {
 int i = 0;
 int length = high - low + 1;
 int pivot  = 0;
 int merge1 = 0;
 int merge2 = 0;
 int working[length];
 if(low == high)
  return;
 pivot  = (low + high) / 2;
 mergesort(a, low, pivot);
 mergesort(a, pivot + 1, high);
  for(i = 0; i < length; i++)
  working[i] = a[low + i];
 merge1 = 0;
 merge2 = pivot - low + 1;
 for(i = 0; i < length; i++) {
  if(merge2 <= high - low)
   if(merge1 <= pivot - low)
    if(working[merge1] > working[merge2])
     a[i + low] = working[merge2++];
    else
     a[i + low] = working[merge1++];
   else
    a[i + low] = working[merge2++];
  else
   a[i + low] = working[merge1++];
 }
}
```

Application:

- Sort a list of names.
- Organize an MP3 library.
- Display Google Page Rank results.
- List RSS news items in reverse chronological order.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Experiment No. 17

Title: Pseudo code and Program that implements Breadth First Search .

Breadth-first search (**BFS**) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspect all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

Pseudo code:

BFS(V, E, *s*)

```
for each u in V − {s}            ▷ for each vertex u in V[G] except s.
    do color[u] ← WHITE
       d[u] ← infinity
       π[u] ← NIL
 color[s] ← GRAY                 ▷ Source vertex discovered
 d[s] ← 0                        ▷ initialize
 π[s] ← NIL                      ▷ initialize
 Q ← {}                          ▷ Clear queue Q
 ENQUEUE(Q, s)
 while Q is non-empty
    do u ← DEQUEUE(Q)                  ▷ That is, u = head[Q]
       for each v adjacent to u        ▷ for loop for every node along with edge.
            do if color[v] ← WHITE     ▷ if color is white you've never seen it before
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       DEQUEUE(Q)
 color[u] ← BLACK
```

Program in C:

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

#define TRUE 1
#define FALSE 0
#define MAX 8
```

```
struct node
{
int data ;
struct node *next ;
} ;
int visited[MAX] ;
int q[8] ;
int front, rear ;
void bfs ( int, struct node ** ) ;
struct node * getnode_write ( int ) ;
void addqueue ( int ) ;
int deletequeue( ) ;
int isempty( ) ;
void del ( struct node * ) ;
void main( )
{
struct node *arr[MAX] ;
struct node *v1, *v2, *v3, *v4 ;
int i ;
clrscr( ) ;
v1 = getnode_write ( 2 ) ;
arr[0] = v1 ;
v1 -> next = v2 = getnode_write ( 3 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 1 ) ;
arr[1] = v1 ;
v1 -> next = v2 = getnode_write ( 4 ) ;
v2 -> next = v3 = getnode_write ( 5 ) ;
v3 -> next = NULL ;
v1 = getnode_write ( 1 ) ;
arr[2] = v1 ;
v1 -> next = v2 = getnode_write ( 6 ) ;
v2 -> next = v3 = getnode_write ( 7 ) ;
v3 -> next = NULL ;
v1 = getnode_write ( 2 ) ;
arr[3] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 2 ) ;
arr[4] = v1 ;
```

```
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 3 ) ;
arr[5] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 3 ) ;
arr[6] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;

v1 = getnode_write ( 4 ) ;
arr[7] = v1 ;
v1 -> next = v2 = getnode_write ( 5 ) ;
v2 -> next = v3 = getnode_write ( 6 ) ;
v3 -> next = v4 = getnode_write ( 7 ) ;
v4 -> next = NULL ;
front = rear = -1 ;
bfs ( 1, arr ) ;
for ( i = 0 ; i < MAX ; i++ )
del ( arr[i] ) ;
getch( ) ;
}

void bfs ( int v, struct node **p )
{
struct node *u ;
visited[v - 1] = TRUE ;
printf ( "%d\t", v ) ;
addqueue ( v ) ;
while ( isempty( ) == FALSE )
{
v = deletequeue( ) ;
u = * ( p + v - 1 ) ;
while ( u != NULL )
{
if ( visited [ u -> data - 1 ] == FALSE )
{
addqueue ( u -> data ) ;
visited [ u -> data - 1 ] = TRUE ;
```

```
printf ( "%d\t", u -> data ) ;
}
u = u -> next ;
}
}
}
struct node * getnode_write ( int val )
{
struct node *newnode ;
newnode = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
newnode -> data = val ;
return newnode ;
}
void addqueue ( int vertex )
{
if ( rear == MAX - 1 )
{
printf ( "\nQueue Overflow." ) ;
exit( ) ;
}
rear++ ;
q[rear] = vertex ;
if ( front == -1 )
front = 0 ;
}
int deletequeue( )
{
int data ;
if ( front == -1 )
{
printf ( "\nQueue Underflow." ) ;
exit( ) ;
}
data = q[front] ;
if ( front == rear )
front = rear = -1 ;
else
front++ ;
return data ;
}
```

```
int isempty( )
{
if ( front == -1 )
return TRUE ;
return FALSE ;
}
void del ( struct node *n )
{
struct node *temp ;
while ( n != NULL )
{
temp = n -> next ;
free ( n ) ;
n = temp ;
}
}
```

Applications:

Breadth-first search can be used to solve many problems in graph theory, for example:

1. Finding all nodes within one connected component
2. Copying Collection, Cheney's algorithm
3. Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges)
4. Testing a graph for bipartiteness
5. (Reverse) Cuthill–McKee mesh numbering
6. Ford–Fulkerson method for computing the maximum flow in a flow network
7. Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

Experiment No. 18

Title: Pseudo code and Program for Implementation of Depth First Search.

Depth-first search (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Pseudo code:

```
DFS(G,v)   ( v is the vertex where the search starts )
      Stack S := {};   ( start with an empty stack )
      for each vertex u, set visited[u] := false;
      push S, v;
      while (S is not empty) do
        u := pop S;
        if (not visited[u]) then
          visited[u] := true;
          for each unvisited neighbour w of u
            push S, w;
        end if
      end while
    END DFS()
```

Program in C:

```
#include <stdio.h>
typedef struct node {
int value;
struct node *right;
struct node *left;
} mynode;
mynode *root;
add_node(int value);
void levelOrderTraversal(mynode *root);
int main(int argc, char* argv[]) {
root = NULL;

add_node(5);
add_node(1);
add_node(-20);
add_node(100);
```

```
add_node(23);
add_node(67);
add_node(13);
printf("\n\n\nLEVEL ORDER TRAVERSAL\n\n");
levelOrderTraversal(root);
getch();
}
// Function to add a new node...
add_node(int value) {
mynode *prev, *cur, *temp;
temp = malloc(sizeof(mynode));
temp->value = value;
temp->right = NULL;
temp->left = NULL;
if(root == NULL) {
printf("\nCreating the root..\n");
root = temp;
return;
}
prev = NULL;
cur = root;
while(cur != NULL) {
prev = cur;
//cur = (value < cur->value) ? cur->left:cur->right;
if(value < cur->value) {
cur = cur->left;
} else {
cur = cur->right;
}
}
if(value < prev->value) {
prev->left = temp;
} else {
prev->right = temp;
}
}
// Level order traversal..
void levelOrderTraversal(mynode *root) {
mynode *queue[100] = {(mynode *)0}; // Important to initialize!
int size = 0;
```

```
int queue_pointer = 0;
while(root) {
printf("[%d] ", root->value);
if(root->left) {
queue[size++] = root->left;
}
if(root->right) {
queue[size++] = root->right;
}
root = queue[queue_pointer++];
}
}
```

Applications:

Algorithms that use depth-first search as a building block include:

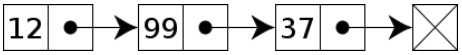
1. Finding connected components.
2. Topological sorting.
3. Finding 2-(edge or vertex)-connected components.
4. Finding 3-(edge or vertex)-connected components.
5. Finding the bridges of a graph.
6. Finding strongly connected components.
7. Planarity Testing[4][5]
8. Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
9. Maze generation may use a randomized depth-first search.
10. Finding biconnectivity in graphs.

BEYOND THE SYLLABUS

Experiment No. 19

Title: Pseudo code and Program for List Implementation using Array

A **Linked list** is a data structure used for collecting a sequence of objects, which allows efficient addition, removal and retrieval of elements from any position in the sequence. It is implemented as nodes, each of which contains a reference (i.e., a *link*) to the next and/or previous node in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

Pseudo-code:

```
LIST-INSERT(L, x)
next[x] <- head[L]
if head[L] != NIL
then prev[head[L]] <- x
head[L] <- x
prev[x] <- NIL

LIST-DELETE(L, x)
if prev[x] != NIL
then next[prev[x]] <- next[x]
else head[L] <- next[x]
if next[x] != NIL
then prev[next[x]] <- prev[x]

LIST-SEARCH(L, k) (k = key)
x <- head[L]
while x != NIL and key[x] != k
do x <- next[x]
return x
```

Program in C:

```
#include<stdio.h>
#include<conio.h>
#define MAX 20 //maximum no of elements in the list
//user defined datatypes
struct
{
int list[MAX];
int element;  //new element to be inserted
int pos;   //position of the element to be inserted or deleted
int length;   //total no of elements
}l;
enum boolean { true, false };
typedef enum boolean boolean;   //function prototypes
int menu(void);    //function to display the list of operations
void create(void);    //function to create initial set of elements
void insert(int, int); //function to insert the given element at specified position
void delet(int);     //function to delete the element at given position
void find(int);    //function to find the position of the given element, if exists
void display(void); //function to display the elements in the list
boolean islistfull(void);   //function to check whether the list is full or not
boolean islistempty(void);   //function to check whether the list is empty or not
void main()
{
int ch;
int element;
int pos;
l.length = 0;
while(1)
{
ch = menu();
switch (ch)
{
case 1:
l.length = 0;
create();
break;
case 2:
if (islistfull() != true)
{
```

```
printf("\tEnter the New element : ");
scanf("%d", &element);
printf("\tEnter the Position : ");
scanf("%d", &pos);
insert(element, pos);
}else
{
printf("\tList if Full. Cannot insert");
printf("\nPress any key to continue...");
getch();
}break;
  case 3:
if (islistempty() != true)
{
printf("Enter the position of element to be deleted : ");
scanf("%d", &pos);
delet(pos);
}else
{
printf("List is Empty.");
printf("\nPress any key to continue...");
getch();
}break;
case 4:
printf("No of elements in the list is %d", l.length);
printf("\nPress any key to continue...");
getch();
break;
case 5:
printf("Enter the element to be searched : ");
scanf("%d", &element);
find(element);
break;
case 6:
display();
break;
case 7:
exit(0);
break;
default:printf("Invalid Choice");
```

```
printf("\nPress any key to continue...");
getch();
}
}
}

//function to display the list of elements
int menu()
{
int ch;
clrscr();
printf("\n\t\t**********************************************\n");
printf("\t\t******LIST Implementation Using Arrays******\n");
printf("\t\t**********************************************\n\n");
printf("\t1. Create\n\t 2. Insert\n\t 3. Delete\n\t 4. Count\n\t 5. Find\n\t 6. Display\n\t
7. Exit\n\n\t Enter your choice : ");
scanf("%d", &ch);
printf("\n\n");
return ch;
}//function to create initial set of elements
void create(void)
{
int element;
int flag=1;
while(flag==1)
{
printf("Enter an element : ");
scanf("%d", &element);
l.list[l.length] = element;
l.length++;
printf("To insert another element press '1' : ");
scanf("%d", &flag);
}
}//function to display the elements in the list
void display(void)
{
int i;
for (i=0; i<l.length; i++)
printf("Element %d : %d \n", i+1, l.list[i]);
printf("Press any key to continue...");
getch();
```

```
}//function to insert the given element at specified position
void insert(int element, int pos)
{
int i;
if (pos == 0)
{
printf("\n\nCannot insert at zeroth position");
getch();

}
```

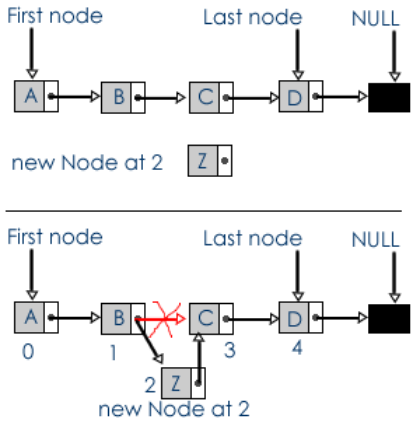
Application: Linked list overcome the disadvantages of array. And the main disadvantages of array are

1) The size of the array is fixed — 100 elements in this case. Most often this size is specified at compile time with a simple declaration such as in the example above . With a little extra effort, the size of the array can be deferred until the array is created at runtime, but after that it remains fixed. You can go to the trouble of dynamically allocating an array in the heap and then dynamically resizing it with realloc(), but that requires some real programmer effort.
2) Because of (1), the most convenient thing for programmers to do is to allocate arrays which seem "large enough" (e.g. the 100 in the scores example). Although convenient, this strategy has two disadvantages: (a) most of the time there are just 20 or 30 elements in the array and 70% of
the space in the array really is wasted. (b) If the program ever needs to process more than 100 scores, the code breaks. A surprising amount of commercial code has this sort of naive array allocation which wastes space most of the time and crashes for special occasions. For relatively large arrays (larger than 8k bytes), the virtual memory system may partially compensate for this problem, since the "wasted" elements are never touched.
3) Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

Experiment No. 20

Title: Linked List implementation using dynamic memory allocation

Linked list is a data structure with the following specifics::
1.Data is dynamically added or removed.
2. Every data object has two parts-a data part and a link part. Together they constitute a node.
3. The list can be traversed only through pointers.
4. Every node is an important constituent of the data.
5. The end of the data is always a leaf end.
6. Tree structures (branched link lists) are used to store data into disks.




```
#include"m_list.h"
void main()
  {
    list *first=NULL,*second=NULL,*third=NULL;
    int choice,i;
    char ch='y';
    while(1)
   {
      clrscr();
      printf("case 1:  Create list");
      printf("  case 2:  Add in the list");
      printf("case 3:  Delete in the list");
      printf("case 4:  Append two list");
      printf(" case 5:  show list");
      printf(" case 6:  Exit");
      printf(" Enter your choice : ");
      scanf("%d",&choice);
```

```
 switch(choice)
{
  case 1:    //create list
     while(ch!='n')
      {
     printf("Enter element : ");
     scanf("%d",&i);
     create(&first,i);
     printf("Enter element (y/n) : ");
     fflush(stdin);
     scanf("%c",&ch);
      }
  break;
  case 2:   //add in the list
      int c;
      clrscr();
      printf(" case 1:  Add in Beginning");
      printf(" case 2:  Add in End");
      printf("case 3:  Add After a given element");
      printf(" case 4:  Return to main menu");
      printf(" Enter your choice : ");
      scanf("%d",&c);
      switch(c)
      {
        case 1: add_at_beg(&first);
           break;
        case 2: add_at_end(&first);
           break;
        case 3: add_after_given_element(&first);
           break;
        case 4: break;
      }
  break;
  case 3:
      clrscr();
      printf(" case 1:  Delete in Beginning");
      printf(" case 2:  Delete in End");
      printf(" case 3:  Delete a specified element");
      printf("case 4:  Return to main menu");
      printf(" Enter your choice : ");
```

```
    scanf("%d",&c);
    switch(c)
    {
      case 1: del_at_beg(&first);
        break;
      case 2: del_at_end(&first);
        break;
      case 3: del_specified_element(&first);
        break;
      case 4: break;
    }
break;
case 4:
     char ch='y';
     printf("Enter element in second list : ");
     while(ch!='n')
     {
       printf("Enter element : ");
       scanf("%d",&i);
       create(&second,i);
       printf("Enter element (y/n) : ");
       fflush(stdin);
       scanf("%c",&ch);
     }
     append(&third,first,second);

break;
case 5:    //show list
    clrscr();
    printf("case 1:  List 1");
    printf(" case 2:  List 2");
    printf("  case 3:  List 3");
    printf(" Enter choice : ");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1: show(first);break;
      case 2: show(second);break;
      case 3: show(third);break;
   }
```

```
        break;
        case 6:    exit(0);

     }
    }
   }


**********************************
#include<conio.h>
#include<stdio.h>
#include<alloc.h>
#include<stdlib.h>
typedef struct list
  {
    int info;
    struct list *next;
  };
//.................Function Declaration .....

void create(struct list **p,int i)
   {
     struct list *temp,*q=*p;
     temp=(struct list*)malloc(sizeof(struct list));
     temp->info=i;
     temp->next=NULL;
     if(*p==NULL)
     *p=temp;
     else
   {
     while(q->next!=NULL)
       q=q->next;
     q->next=temp;
   }
   }
int append(struct list **t,struct list *f,struct list *s)
   {
   struct list *temp=*t;
   if(f==NULL && s==NULL)
     return 0;
```

```
  while(f)
   {
    create(t,f->info);
    f=f->next;
   }
  while(s)
   {
    create(t,s->info);
    s=s->next;
   }

  return 0;
  }
void show(struct list *p)
  {
   if(p==NULL)
  printf(" List is Empty");
   else
   while(p)
     {
   printf("%d  ",p->info);
   p=p->next;
     }
   getch();
  }
void add_at_beg(struct list **l)
   {
    struct list *temp=(struct list *)malloc(sizeof(struct list));
    printf("Enter element : ");
    scanf("%d",&temp->info);
    temp->next=NULL;
    if(*l==NULL)
       *l=temp;
    else
       {
     temp->next=*l;
     *l=temp;
       }
   }
void del_at_beg(struct list **l)
```

```
    {
     list *temp;
     if(*l==NULL)
       {
     printf("
List is empty");
     getch();
       }
     else
       {
     temp=*l;
     *l=(*l)->next;
     free(temp);
       }
    }
void add_at_end(struct list **l)
     {
      list *temp,*p;
      temp=(struct list *)malloc(sizeof(struct list));
      printf("Enter element : ");
      scanf("%d",&temp->info);
      temp->next=NULL;
      if(*l==NULL)
     *l=temp;
      else
     {
       p=*l;
       while(p->next!=NULL)
          p=p->next;
       p->next=temp;
     }
     }

void del_at_end(struct list **l)
     {
      list *temp,*p;
      if(*l==NULL)
     {
      printf("
List is Empty");
```

```
        getch();
      }
       else if((*l)->next==NULL)
      {
        temp=*l;
        *l=NULL;
        free(temp);
      }
       else
      {
       p=*l;
       while(p->next->next!=NULL)
          p=p->next;
       temp=p->next->next;
       p->next=NULL;
       free(temp);
      }
      }
void add_after_given_element(list **l)
   {
      list *temp,*p;
      int m;
      temp=(struct list *)malloc(sizeof(struct list));
      printf("Enter element : ");
      scanf("%d",&temp->info);
      printf("Enter position after which element inserted  : ");
      scanf("%d",&m);
      temp->next=NULL;
      if(*l==NULL)
  *l=temp;
       else
   {
     p=*l;
     while(p->next!=NULL)
        if(p->info==m)
     break;
        else
     p=p->next;

     temp->next=p->next;
```

```
      p->next=temp;

    }
    }
void del_specified_element(list **l)
    {
       list *temp,*p,*q;
       int m;
       printf("
Enter element which is deleted : ");
       scanf("%d",&m);
       if(*l==NULL)
     {
       printf("List is Empty");
       getch();
     }
       else if((*l)->next!=NULL && (*l)->info==m)
     {
         temp=*l;
         *l=(*l)->next;
         free(temp);
     }
       else if((*l)->next==NULL && (*l)->info==m)
      {
         temp=*l;
         *l=NULL;
         free(temp);
      }
       else
    {
      p=*l;
      while(p!=NULL)
         if(p->info==m)
       break;
         else
       {
         q=p;
         p=p->next;
       }
      temp=p;
```

```
    q->next=p->next;
    free(temp);
  }
  }
```

Applications:

A linked list can be used to manage a dynamically growing/shrinking list of data. It's primary advantage is for sorting: if each piece of "data" consists of a large data structure, you only have to adjust the links (usually a pointer) rather than copying the data. The biggest disadvantage is in searching through the data, as you have to traverse through all the items on the list preceding it.

Experiment No. 21

Title: pseudo code and Program to sort elements of given array using Quick Sort Algorithm

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

- Pick an element, called a *pivot*, from the list.
- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Pseudo code:

```
Quicksort(A,p,r) {
   if (p < r) {
      q <- Partition(A,p,r)
      Quicksort(A,p,q)
      Quicksort(A,q+1,r)
   }
}

Partition(A,p,r)
   x <- A[p]
   i <- p-1
   j <- r+1
   while (True) {
      repeat
         j <- j-1
      until (A[j] <= x)
      repeat
         i <- i+1
      until (A[i] >= x)
      if (i<-=""> A[j]
      else
         return(j)
   }
}
```

Program in C:

```
#include<stdio.h>
#include<conio.h>
#define max 15
int beg,end,top,i,n,loc,left,right;
int array[max+1];    //contains the various elements.
int upper[max-1],lower[max-1];
//two stacks to store two ends of the list.

void main()
      {

        void enter(void);
        void quick(void);
        void prnt(void);
        clrscr();
        enter();     //entering elements in the array
        top=i-1;     //set top to stack
                if (top==0)
                  {
       printf(" UNDERFLOW CONDITION       ");
       getch();
       exit();
      }
         top=0;
         if(n>1)
                  {
       top++;
       lower[top]=1;upper[top]=n;
       while ( top!=NULL )
       {
         beg=lower[top];
         end=upper[top];
         top--;
         left=beg;   right=end;    loc=beg;
         quick();
         if ( beg<loc-1)
                  {
      top++;
      lower[top]=beg;
      upper[top]=loc-1;
```

```
                }
              if(loc+1<end)
                {
    top++;
    lower[top]=loc+1;
    upper[top]=end;
    }
      }              //end of while
     }               //end of if statement
              printf("Sorted elements of the array are :");
              prnt();   //to print the sorted array
     getch();
    }             //end of main


void  enter(void)
 {
    printf("Enter the no of elements in the array:");
    scanf("%d",&n);
    printf("Enter the elements of the array :");
    for(i=1;i<=n;i++)
      {
     printf(" Enter the %d element :",i);
     scanf("%d",&array[i]);
      }
 }


 void  prnt(void)
 {
    for(i=1;i<=n;i++)
      {
     printf(" The %d element  is : %d",i,array[i]);
      }
 }

 void quick()
 {
     int temp;
     void tr_fr_right(void);
```

```
    while( array[loc]<=array[right] && loc!=right)
    {
     right--;
    }

    if(loc==right)
    return ;

    if(array[loc]>array[right])
    {
            temp=array[loc];
            array[loc]=array[right];
            array[right]=temp;
    loc=right;
    tr_fr_right();
    }
    return ;
}

void  tr_fr_right()
{
     int temp;
     while( array[loc] > array[left] && loc!=left)
    {
     left++;
    }

    if(loc==left)
    return ;

    if(array[loc] < array[left])
    {
            temp=array[loc];
            array[loc]=array[left];
            array[left]=temp;
    loc=left;
    quick();
    }
    return ;
}
```

Application:

Commercial applications use Quicksort - generally it runs fast, no additional memory, this compensates for the rare occasions when it runs with $O(N^2)$. Never use in applications which require guaranteed response time:

1. Life-critical (medical monitoring, life support in aircraft and space craft).

2. Mission-critical (monitoring and control in industrial and research plants handling dangerous materials, control for aircraft, defense, etc) .Unless you assume the worst-case response time.