DATA STRUCTURES LAB

# Lab Manual

# Lab Manual

* This experiment is mapped to the CO1 (Course Outcome 1) - Develop Program to implement Array Data Structures.
* Operations on Linear Array.  
    
  **Write a menu-driven program that implement following operations (using separate functions) on a linear array:  
  1. Insert a new element at end as well as at a given position.  
  2. Delete an element from a given whose value is given or whose position is given.  
  3. To find the location of a given element.  
  4. To display the elements of the linear array.**   
  **PROGRAM CODE:**   
  #include<stdlib.h>  
  #include<stdio.h>  
  using namespace std;  
     
  int a[20],b[20],c[40];  
  int n,val,i,j,pos;  
     
  void create();  
  void display();  
  void insert();  
  void del();  
  void search();  
  void merge();  
  void sort();  
     
  int main()  
  {  
   int choice;  
   do{  
   printf("\n\n--Menu--\n");  
   printf("1.Create\n");  
   printf("2.Display\n");  
   printf("3.Insert\n");  
   printf("4.Delete\n");  
   printf("5.Exit\n");  
   printf("-----");  
   printf("\nEnter your choice:\t");  
   scanf("%d",&choice);  
   switch(choice)  
   {  
   case 1: create();  
   break;  
   case 2: display();  
   break;  
   case 3: insert();  
   break;  
   case 4: del();  
   break;  
   case 5: exit(0);  
   break;  
   default: printf("\nInvalid choice:\n");  
   break;  
   }  
   }while(choice!=5);  
  return 0;  
  }  
  void create() //creating an array  
  {  
   printf("\nEnter the size of the array elements:\t");  
   scanf("%d",&n);  
   printf("\nEnter the elements for the array:\n");  
   for(i=0;i<n;i++)  
   {  
   scanf("%d",&a[i]);  
   }  
  }//end of create()  
  void display() //displaying an array elements  
  {  
   int i;  
   printf("\nThe array elements are:\n");  
   for(i=0;i<n;i++){  
   printf("%d\t",a[i]);  
   }  
   }//end of display()  
  void insert() //inserting an element in to an array  
  {  
   printf("\nEnter the position for the new element:\t");  
   scanf("%d",&pos);  
   printf("\nEnter the element to be inserted :\t");  
   scanf("%d",&val);  
   for(i=n-1;i>=pos;i--)  
   {  
   a[i+1]=a[i];  
   }  
   a[pos]=val;  
   n=n+1;  
  }//end of insert()  
  void del() //deleting an array element  
  {  
   printf("\nEnter the position of the element to be deleted:\t");  
   scanf("%d",&pos);  
   val=a[pos];  
   for(i=pos;i<n-1;i++)  
   {  
   a[i]=a[i+1];  
   }  
   n=n-1;  
   printf("\nThe deleted element is =%d",val);  
  }//end of delete()

DATA STRUCTURES LAB

# Manual 2

This experiment deals with String Creation, Operations on string.

# Manual

* **Write a program to implement the following operations on strings:  
  Read a string, pattern, replace string, perform pattern matching, find and replace any occurrences of a pattern  
     
  PROGRAM CODE:**   
  #include <stdio.h>  
  #include <string.h>  
     
  int main()  
  {  
   char str[100], ch, Newch;  
   int i;  
     
   printf("\n Please Enter any String : ");  
   gets(str);  
     
   printf("\n Please Enter the Character that you want to Search for : ");  
   scanf("%c", &ch);  
     
   getchar();  
     
   printf("\n Please Enter the New Character : ");  
     
   scanf("%c", &Newch);  
     
   for(i = 0; i <= strlen(str); i++)  
   {  
   if(str[i] == ch)  
   {  
   str[i] = Newch;  
   }  
   }  
   printf("\n Final String after Replacing All Occurrences of '%c' with '%c' = %s ", ch, Newch, str);  
     
   return 0;  
  }

DATA STRUCTURES LAB

# Lab Manual 3

This practical is about the Stack and its operation implementation.

# Lab Manual

* **Write a program to implement the functions on a stack:  
  1. PUSH  
  2. POP  
  3. OVERFLOW and UNDERFLOW  
     
  PROGRAM CODE:**   
  #include <stdio.h>  
  #include <stdlib.h>  
  #include <limits.h> // For INT\_MIN  
     
  #define SIZE 100  
     
  // Create a stack with capacity of 100 elements  
  int stack[SIZE];  
     
  // Initially stack is empty  
  int top = -1;  
     
  /\* Function declaration to perform push and pop on stack \*/  
  void push(int element);  
  int pop();  
     
  int main()  
  {  
   int choice, data;  
     
   while(1)  
   {  
   /\* Menu \*/  
   printf("------------------------------------\n");  
   printf(" STACK IMPLEMENTATION PROGRAM \n");  
   printf("------------------------------------\n");  
   printf("1. Push\n");  
   printf("2. Pop\n");  
   printf("3. Size\n");  
   printf("4. Exit\n");  
   printf("------------------------------------\n");  
   printf("Enter your choice: ");  
     
   scanf("%d", &choice);  
     
   switch(choice)  
   {  
   case 1:  
   printf("Enter data to push into stack: ");  
   scanf("%d", &data);  
     
   // Push element to stack  
   push(data);  
   break;  
     
   case 2:  
   data = pop();  
     
   // If stack is not empty  
   if (data != INT\_MIN)  
   printf("Data => %d\n", data);  
   break;  
     
   case 3:  
   printf("Stack size: %d\n", top + 1);  
   break;  
     
   case 4:  
   printf("Exiting from app.\n");  
   exit(0);  
   break;  
     
   default:  
   printf("Invalid choice, please try again.\n");  
   }  
   printf("\n\n");  
   }  
   return 0;  
  }  
     
  /\*\*  
   \* Functiont to push a new element in stack.  
   \*/  
  void push(int element)  
  {  
   // Check stack overflow  
   if (top >= SIZE)  
   {  
   printf("Stack Overflow, can't add more element element to stack.\n");  
   return;  
   }  
     
   // Increase element count in stack  
   top++;  
     
   // Push element in stack  
   stack[top] = element;  
     
   printf("Data pushed to stack.\n");  
  }  
     
  /\*\*  
   \* Function to pop element from top of stack.  
   \*/  
  int pop()  
  {  
   // Check stack underflow  
   if (top < 0)  
   {  
   printf("Stack is empty.\n");  
   return INT\_MIN;  
   }  
   return stack[top--];  
  }

DATA STRUCTURES LAB

# Lab Manual 4

# Lab Manual

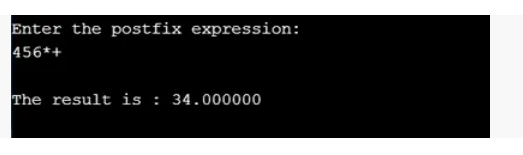
* **Write a program for converting an Infix Expression to Postfix Expression.**
* **Experiment on Application of Stack - Infix to PostFix Conversion**    
  **PROGRAM CODE:**#include<stdio.h>  
  char stack[20];  
  int top = -1;  
  void push(char x)  
  {  
   stack[++top] = x;  
  }  
     
  char pop()  
  {  
   if(top == -1)  
   return -1;  
   else  
   return stack[top--];  
  }  
     
  int priority(char x)  
  {  
   if(x == '(')  
   return 0;  
   if(x == '+' || x == '-')  
   return 1;  
   if(x == '\*' || x == '/')  
   return 2;  
  }  
     
  main()  
  {  
   char exp[20];  
   char \*e, x;  
   printf("Enter the expression :: ");  
   scanf("%s",exp);  
   e = exp;  
   while(\*e != '\0')  
   {  
   if(isalnum(\*e))  
   printf("%c",\*e);  
   else if(\*e == '(')  
   push(\*e);  
   else if(\*e == ')')  
   {  
   while((x = pop()) != '(')  
   printf("%c", x);  
   }  
   else  
   {  
   while(priority(stack[top]) >= priority(\*e))  
   printf("%c",pop());  
   push(\*e);  
   }  
   e++;  
   }  
   while(top != -1)  
   {  
   printf("%c",pop());  
   }  
  }

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

Experiment on Stack Applications

# Manual (Algorithm, Code and Analysis)

* **Evaluation of Suffix expression with single digit operands and operators: +, -, \*, /, %, ^  
  Code**#include<stdio.h>  
  #include<math.h>  
  #include<string.h>  
  float compute(char symbol, float op1, float op2)  
  {  
   switch (symbol)  
   {  
   case '+': return op1 + op2;  
   case '-': return op1 - op2;  
   case '\*': return op1 \* op2;  
   case '/': return op1 / op2;  
   case '$':  
   case '^': return pow(op1,op2);  
   default : return 0;  
   }  
  }  
  void main()  
  {  
  float s[20], res, op1, op2;  
  int top, i;  
  char postfix[20], symbol;  
  printf("\nEnter the postfix expression:\n");  
  scanf ("%s", postfix);  
  top=-1;  
  for (i=0; i<strlen(postfix) ;i++)  
  {  
  symbol = postfix[i];  
  if(isdigit(symbol))  
   s[++top]=symbol - '0';  
  else  
   {  
   op2 = s[top--];  
   op1 = s[top--];  
   res = compute(symbol, op1, op2);  
   s[++top] = res;  
   }  
  }  
  res = s[top--];  
  printf("\nThe result is : %f\n", res);  
  }   
  **Output  
  **
* **Op.jpg**
* Complexity  
  T(n) = O(n).  
  **Solving Tower of Hanoi problem with n disks**Code  
  #include<stdio.h>  
  #include<conio.h>  
  void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod)   
  {   
   if (n == 1)   
   {   
   printf("\n Move disk 1 from rod %c to rod %c", from\_rod, to\_rod);   
   return;   
   }   
   towerOfHanoi(n-1, from\_rod, aux\_rod, to\_rod);   
   printf("\n Move disk %d from rod %c to rod %c", n, from\_rod, to\_rod);   
   towerOfHanoi(n-1, aux\_rod, to\_rod, from\_rod);   
  }   
   int main()   
  {   
   int n = 4; // Number of disks   
   towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods   
   return 0;   
  }   
  **Output**
* ****
* **tohoi.jpg**
* Complexity  
  T(n) = O(2^n - 1)

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

Circular Queue

# Manual (Algorithm, Code and Analysis)

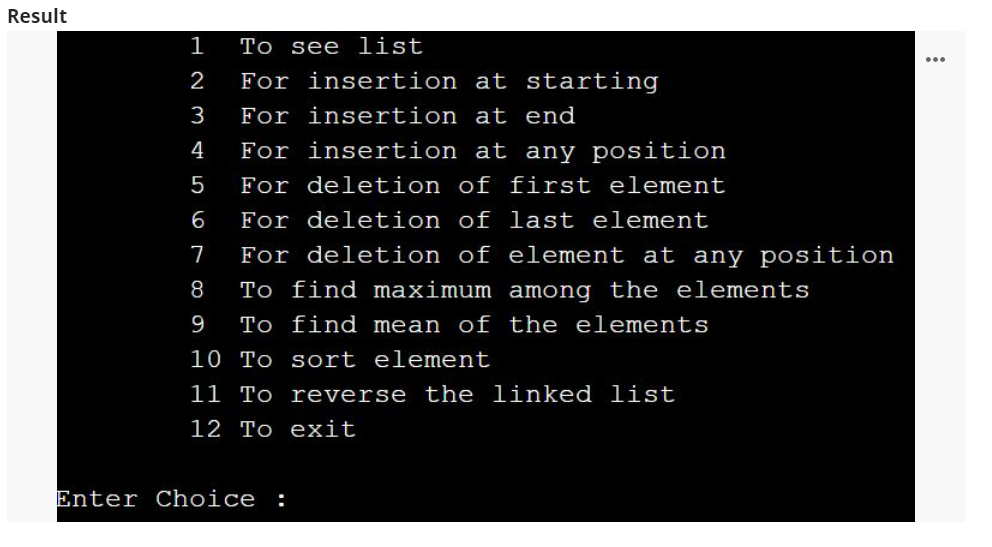
* Experiment Title  
  A menu driven Program for the operations on Circular QUEUE of Characters (Array Implementation of Queue with maximum size MAX)  
    
  Insert an Element on to Circular QUEUE  
  Delete an Element from Circular QUEUE  
  Display the status of Circular QUEUE  
  Exit  
    
  **Introduction on Circular Queue**Circular Queue is also a linear data structure, which follows the principle of FIFO(First In First Out) and the last position is connected back to the first position to make a circle. It is also called ‘Ring Buffer’., but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.  
    
  **Algorithm  
  enQueue(value)**This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.   
  Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).  
  If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.  
  **deQueue()**This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.   
  Check whether queue is Empty means check (front==-1).  
  If it is empty then display Queue is empty. If queue is not empty then step 3  
  Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.  
    
  **Code**#include<stdlib.h>   
    
  #include<conio.h>  
    
  # include<stdio.h>  
    
  # define MAX 5  
    
  int cqueue\_arr[MAX];  
    
  int front = -1;  
    
  int rear = -1;  
    
  void insert(int item)  
    
  {  
    
  if((front == 0 && rear == MAX-1) || (front == rear+1))  
    
  {  
    
  printf("Queue Overflow \n");  
    
  return;  
    
  }  
    
  if (front == -1) /\*If queue is empty \*/  
    
  {  
    
  front = 0;  
    
  rear = 0;  
    
  }  
    
  else  
    
  {  
    
  if(rear == MAX-1) /\*rear is at last position of queue \*/  
    
  rear = 0;  
    
  else  
    
  rear = rear+1;  
    
  }  
    
  cqueue\_arr[rear] = item ;  
    
  }  
    
  void del()  
    
  {  
    
  if (front == -1)  
    
  {  
    
  printf("Queue Underflow\n");  
    
  return ;  
    
  }  
    
  printf("Element deleted from queue is : %d\n",cqueue\_arr[front]);  
    
  if(front == rear) /\* queue has only one element \*/  
    
  {  
    
  front = -1;  
    
  rear=-1;  
    
  }  
    
  else  
    
  {  
    
  if(front == MAX-1)  
    
  front = 0;  
    
  else  
    
  front = front+1;  
    
  }  
    
  }  
    
  void display()  
    
  {  
    
  int front\_pos = front,rear\_pos = rear;  
    
  if(front == -1)  
    
  {  
    
  printf("Queue is empty\n");  
    
  return;  
    
  }  
    
  printf("Queue elements :\n");  
    
  if( front\_pos <= rear\_pos )  
    
  while(front\_pos <= rear\_pos)  
    
  {  
    
  printf("%d ",cqueue\_arr[front\_pos]);  
    
  front\_pos++;  
    
  }  
    
  else  
    
  {  
    
  while(front\_pos <= MAX-1)  
    
  {  
    
  printf("%d ",cqueue\_arr[front\_pos]);  
    
  front\_pos++;  
    
  }  
    
  front\_pos = 0;  
    
  while(front\_pos <= rear\_pos)  
    
  {  
    
  printf("%d ",cqueue\_arr[front\_pos]);  
    
  front\_pos++;  
    
  }  
    
  }  
    
  printf("\n");  
    
  }  
    
  int main()  
    
  {  
    
  int choice,item;  
    
  do  
    
  {  
    
  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 :  
    
  printf("Input the element for insertion in queue : ");  
    
  scanf("%d", &item);  
    
  insert(item);  
    
  break;  
    
  case 2 :  
    
  del();  
    
  break;  
    
  case 3:  
    
  display();  
    
  break;  
    
  case 4:  
    
  break;  
    
  default:  
    
  printf("Wrong choice\n");  
    
  }  
    
  }while(choice!=4);  
    
  return 0;  
    
  }   
  Results
* result.JPG
* Analysis  
  Time complexity: O(1) for enqueue and dequeue operations in queue.

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

Singly Linked List Implementation

# Manual (Algorithm, Code and Analysis)

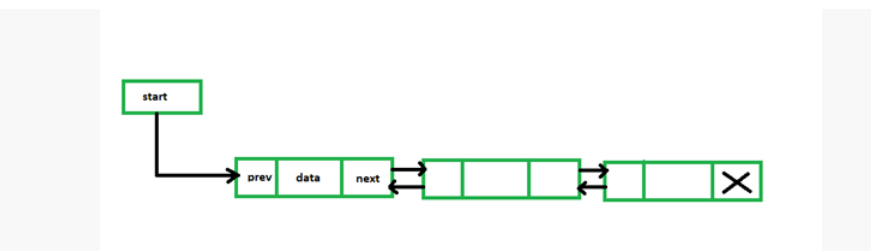
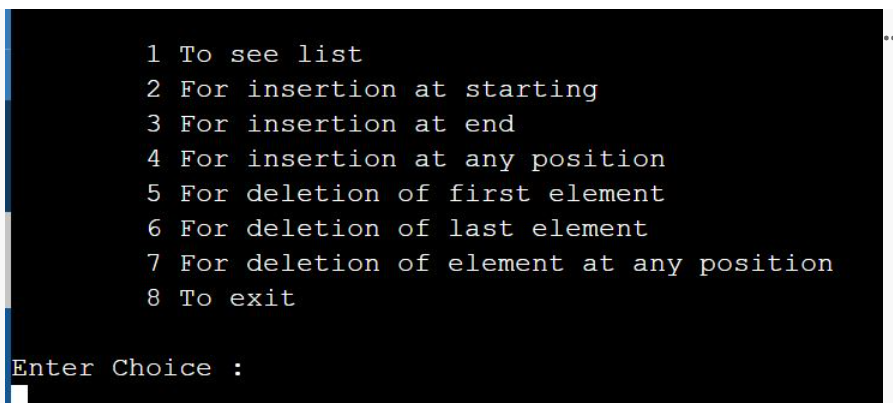
* **Experiment Title**A menu driven Program for operations on Singly Linked List (SLL)  
  **Introduction**A Linked List is a linear data structure that consists of two parts: one is the data part and the other is the address part.  
    
  **Operations to be performed:**traverse(): To see the contents of the linked list, it is necessary to traverse the given linked list. The given traverse() function traverses and prints the content of the linked list.  
  insertAtFront(): This function simply inserts an element at the front/beginning of the linked list.  
  insertAtEnd(): This function inserts an element at the end of the linked list.  
  insertAtPosition(): This function inserts an element at a specified position in the linked list.  
  deleteFirst(): This function simply deletes an element from the front/beginning of the linked list.  
  deleteEnd(): This function simply deletes an element from the end of the linked list.  
  deletePosition(): This function deletes an element from a specified position in the linked list.  
  maximum(): This function finds the maximum element in a linked list.  
  mean(): This function finds the mean of the elements in a linked list.  
  sort(): This function sort the given linked list in ascending order.  
  reverseLL(): This function reverses the given linked list.  
  **Algorithm**https://www.programiz.com/dsa/linked-list-operations (Read This)  
  **Code**// C program for the all operations in   
  // the Singly Linked List   
  #include <stdio.h>   
    
     
  // Linked List Node   
    
  struct node {   
    
   int info;   
    
   struct node\* link;   
  };   
    
  struct node\* start = NULL;   
    
     
  // Function to traverse the linked list   
    
  void traverse()   
  {   
    
   struct node\* temp;   
    
     
    
   // List is empty   
    
   if (start == NULL)   
    
   printf("\nList is empty\n");   
    
     
    
   // Else print the LL   
    
   else {   
    
   temp = start;   
    
   while (temp != NULL) {   
    
   //printf("Data = %d\n",   
    
   // temp->info);   
   printf("%d ->",temp->info);  
    
   temp = temp->link;   
    
   }   
    
   }   
  }   
    
     
  // Function to insert at the front   
  // of the linked list   
    
  void insertAtFront()   
  {   
    
   int data;   
    
   struct node\* temp;   
    
   temp = malloc(sizeof(struct node));   
    
   printf("\nEnter number to"  
    
   " be inserted : ");   
    
   scanf("%d", &data);   
    
   temp->info = data;   
    
     
    
   // Pointer of temp will be   
    
   // assigned to start   
    
   temp->link = start;   
    
   start = temp;   
  }   
    
     
  // Function to insert at the end of   
  // the linked list   
    
  void insertAtEnd()   
  {   
    
   int data;   
    
   struct node \*temp, \*head;   
    
   temp = malloc(sizeof(struct node));   
    
     
    
   // Enter the number   
    
   printf("\nEnter number to"  
    
   " be inserted : ");   
    
   scanf("%d", &data);   
    
     
    
   // Changes links   
    
   temp->link = 0;   
    
   temp->info = data;   
    
   head = start;   
    
   while (head->link != NULL) {   
    
   head = head->link;   
    
   }   
    
   head->link = temp;   
  }   
    
     
  // Function to insert at any specified   
  // position in the linked list   
    
  void insertAtPosition()   
  {   
    
   struct node \*temp, \*newnode;   
    
   int pos, data, i = 1;   
    
   newnode = malloc(sizeof(struct node));   
    
     
    
   // Enter the position and data   
    
   printf("\nEnter position and data :");   
    
   scanf("%d %d", &pos, &data);   
    
     
    
   // Change Links   
    
   temp = start;   
    
   newnode->info = data;   
    
   newnode->link = 0;   
    
   while (i < pos - 1) {   
    
   temp = temp->link;   
    
   i++;   
    
   }   
    
   newnode->link = temp->link;   
    
   temp->link = newnode;   
  }   
    
     
  // Function to delete from the front   
  // of the linked list   
    
  void deleteFirst()   
  {   
    
   struct node\* temp;   
    
   if (start == NULL)   
    
   printf("\nList is empty\n");   
    
   else {   
    
   temp = start;   
    
   start = start->link;   
    
   free(temp);   
    
   }   
  }   
    
     
  // Function to delete from the end   
  // of the linked list   
    
  void deleteEnd()   
  {   
    
   struct node \*temp, \*prevnode;   
    
   if (start == NULL)   
    
   printf("\nList is Empty\n");   
    
   else {   
    
   temp = start;   
    
   while (temp->link != 0) {   
    
   prevnode = temp;   
    
   temp = temp->link;   
    
   }   
    
   free(temp);   
    
   prevnode->link = 0;   
    
   }   
  }   
    
     
  // Function to delete from any specified   
  // position from the linked list   
    
  void deletePosition()   
  {   
    
   struct node \*temp, \*position;   
    
   int i = 1, pos;   
    
     
    
   // If LL is empty   
    
   if (start == NULL)   
    
   printf("\nList is empty\n");   
    
     
    
   // Otherwise   
    
   else {   
    
   printf("\nEnter index : ");   
    
     
    
   // Position to be deleted   
    
   scanf("%d", &pos);   
    
   position = malloc(sizeof(struct node));   
    
   temp = start;   
    
     
    
   // Traverse till position   
    
   while (i < pos - 1) {   
    
   temp = temp->link;   
    
   i++;   
    
   }   
    
     
    
   // Change Links   
    
   position = temp->link;   
    
   temp->link = position->link;   
    
     
    
   // Free memory   
    
   free(position);   
    
   }   
  }   
    
     
  // Function to find the maximum element   
  // in the linked list   
    
  void maximum()   
  {   
    
   int a[10];   
    
   int i;   
    
   struct node\* temp;   
    
     
    
   // If LL is empty   
    
   if (start == NULL)   
    
   printf("\nList is empty\n");   
    
     
    
   // Otherwise   
    
   else {   
    
   temp = start;   
    
   int max = temp->info;   
    
     
    
   // Traverse LL and update the   
    
   // maximum element   
    
   while (temp != NULL) {   
    
     
    
   // Update the maximum   
    
   // element   
    
   if (max < temp->info)   
    
   max = temp->info;   
    
   temp = temp->link;   
    
   }   
    
   printf("\nMaximum number "  
    
   "is : %d ",   
    
   max);   
    
   }   
  }   
    
     
  // Function to find the mean of the   
  // elements in the linked list   
    
  void mean()   
  {   
    
   int a[10];   
    
   int i;   
    
   struct node\* temp;   
    
     
    
   // If LL is empty   
    
   if (start == NULL)   
    
   printf("\nList is empty\n");   
    
     
    
   // Otherwise   
    
   else {   
    
   temp = start;   
    
     
    
   // Stores the sum and count of   
    
   // element in the LL   
    
   int sum = 0, count = 0;   
    
   float m;   
    
     
    
   // Traverse the LL   
    
   while (temp != NULL) {   
    
     
    
   // Update the sum   
    
   sum = sum + temp->info;   
    
   temp = temp->link;   
    
   count++;   
    
   }   
    
     
    
   // Find the mean   
    
   m = sum / count;   
    
     
    
   // Print the mean value   
    
   printf("\nMean is %f ", m);   
    
   }   
  }   
    
     
  // Function to sort the linked list   
  // in ascending order   
    
  void sort()   
  {   
    
   struct node\* current = start;   
    
   struct node\* index = NULL;   
    
   int temp;   
    
     
    
   // If LL is empty   
    
   if (start == NULL) {   
    
   return;   
    
   }   
    
     
    
   // Else   
    
   else {   
    
     
    
   // Traverse the LL   
    
   while (current != NULL) {   
    
   index = current->link;   
    
     
    
   // Traverse the LL nestedly   
    
   // and find the minimum   
    
   // element   
    
   while (index != NULL) {   
    
     
    
   // Swap with it the value   
    
   // at current   
    
   if (current->info > index->info) {   
    
   temp = current->info;   
    
   current->info = index->info;   
    
   index->info = temp;   
    
   }   
    
   index = index->link;   
    
   }   
    
     
    
   // Update the current   
    
   current = current->link;   
    
   }   
    
   }   
  }   
    
     
  // Function to reverse the linked list   
    
  void reverseLL()   
  {   
    
   struct node \*t1, \*t2, \*temp;   
    
   t1 = t2 = NULL;   
    
     
    
   // If LL is empty   
    
   if (start == NULL)   
    
   printf("List is empty\n");   
    
     
    
   // Else   
    
   else {   
    
     
    
   // Traverse the LL   
    
   while (start != NULL) {   
    
     
    
   // reversing of points   
    
   t2 = start->link;   
    
   start->link = t1;   
    
   t1 = start;   
    
   start = t2;   
    
   }   
    
   start = t1;   
    
     
    
   // New head Node   
    
   temp = start;   
    
     
    
   printf("Reversed linked "  
    
   "list is : ");   
    
     
    
   // Print the LL   
    
   while (temp != NULL) {   
    
   printf("%d ", temp->info);   
    
   temp = temp->link;   
    
   }   
    
   }   
  }   
    
     
  // Driver Code   
    
  int main()   
  {   
    
   int choice;   
    
   while (1) {   
    
     
    
   printf("\n\t1 To see list\n");   
    
   printf("\t2 For insertion at"  
    
   " starting\n");   
    
   printf("\t3 For insertion at"  
    
   " end\n");   
    
   printf("\t4 For insertion at "  
    
   "any position\n");   
    
   printf("\t5 For deletion of "  
    
   "first element\n");   
    
   printf("\t6 For deletion of "  
    
   "last element\n");   
    
   printf("\t7 For deletion of "  
    
   "element at any position\n");   
    
   printf("\t8 To find maximum among"  
    
   " the elements\n");   
    
   printf("\t9 To find mean of "  
    
   "the elements\n");   
    
   printf("\t10 To sort element\n");   
    
   printf("\t11 To reverse the "  
    
   "linked list\n");   
    
   printf("\t12 To exit\n");   
    
   printf("\nEnter Choice :\n");   
    
   scanf("%d", &choice);   
    
     
    
   switch (choice) {   
    
   case 1:   
    
   traverse();   
    
   break;   
    
   case 2:   
    
   insertAtFront();   
    
   break;   
    
   case 3:   
    
   insertAtEnd();   
    
   break;   
    
   case 4:   
    
   insertAtPosition();   
    
   break;   
    
   case 5:   
    
   deleteFirst();   
    
   break;   
    
   case 6:   
    
   deleteEnd();   
    
   break;   
    
   case 7:   
    
   deletePosition();   
    
   break;   
    
   case 8:   
    
   maximum();   
    
   break;   
    
   case 9:   
    
   mean();   
    
   break;   
    
   case 10:   
    
   sort();   
    
   break;   
    
   case 11:   
    
   reverseLL();   
    
   break;   
    
   case 12:   
    
   exit(1);   
    
   break;   
    
   default:   
    
   printf("Incorrect Choice\n");   
    
   }   
    
   }   
    
   return 0;   
  }  
  **Result**
* ****
* **result 3.JPG**
* **Analysis**Traverse /Access- O(N)  
  Search- O(N)  
  Insertion - O(1)  
  Deletion - O(1)

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

Doubly Linked List

# Manual (Algorithm, Code and Analysis)

* Experiment Title  
  A menu driven Program for the operations on Doubly Linked List (DLL)  
    
  **Introduction**A Linked List is a linear data structure that consists of two parts: one is the data part and the other is the address part. A Doubly Linked List in contains three parts: one is the data part and the other two are the address of the next and previous node in the list.
* 
* DLL.png
* **Algorithm / Code**// C program for the all operations in  
  // the Doubly Linked List  
  #include <stdio.h>  
  #include <stdlib.h>  
    
  // Linked List Node  
  struct node {  
  int info;  
  struct node \*prev, \*next;  
  };  
  struct node\* start = NULL;  
    
  // Function to traverse the linked list  
  void traverse()  
  {  
  // List is empty  
  if (start == NULL) {  
  printf("\nList is empty\n");  
  return;  
  }  
  // Else print the Data  
  struct node\* temp;  
  temp = start;  
  while (temp != NULL) {  
  printf("Data = %d\n", temp->info);  
  temp = temp->next;  
  }  
  }  
    
  // Function to insert at the front  
  // of the linked list  
  void insertAtFront()  
  {  
  int data;  
  struct node\* temp;  
  temp = (struct node\*)malloc(sizeof(struct node));  
  printf("\nEnter number to be inserted: ");  
  scanf("%d", &data);  
  temp->info = data;  
  temp->prev = NULL;  
    
  // Pointer of temp will be  
  // assigned to start  
  temp->next = start;  
  start = temp;  
  }  
    
  // Function to insert at the end of  
  // the linked list  
  void insertAtEnd()  
  {  
  int data;  
  struct node \*temp, \*trav;  
  temp = (struct node\*)malloc(sizeof(struct node));  
  temp->prev = NULL;  
  temp->next = NULL;  
  printf("\nEnter number to be inserted: ");  
  scanf("%d", &data);  
  temp->info = data;  
  temp->next = NULL;  
  trav = start;  
    
  // If start is NULL  
  if (start == NULL) {  
    
  start = temp;  
  }  
    
  // Changes Links  
  else {  
  while (trav->next != NULL)  
  trav = trav->next;  
  temp->prev = trav;  
  trav->next = temp;  
  }  
  }  
    
  // Function to insert at any specified  
  // position in the linked list  
  void insertAtPosition()  
  {  
  int data, pos, i = 1;  
  struct node \*temp, \*newnode;  
  newnode = malloc(sizeof(struct node));  
  newnode->next = NULL;  
  newnode->prev = NULL;  
    
  // Enter the position and data  
  printf("\nEnter position : ");  
  scanf("%d", &pos);  
  printf("\nEnter number to be inserted: ");  
  scanf("%d", &data);  
  newnode->info = data;  
  temp = start;  
    
  // If start==NULL,  
  if (start == NULL) {  
  start = newnode;  
  newnode->prev = NULL;  
  newnode->next = NULL;  
  }  
    
  // If position==1,  
  else if (pos == 1) {  
  newnode->next = start;  
  newnode->next->prev = newnode;  
  newnode->prev = NULL;  
  start = newnode;  
  }  
    
  // Change links  
  else {  
  while (i < pos - 1) {  
  temp = temp->next;  
  i++;  
  }  
  newnode->next = temp->next;  
  newnode->prev = temp;  
  temp->next = newnode;  
  temp->next->prev = newnode;  
  }  
  }  
    
  // Function to delete from the front  
  // of the linked list  
  void deleteFirst()  
  {  
  struct node\* temp;  
  if (start == NULL)  
  printf("\nList is empty\n");  
  else {  
  temp = start;  
  start = start->next;  
  if (start != NULL)  
  start->prev = NULL;  
  free(temp);  
  }  
  }  
    
  // Function to delete from the end  
  // of the linked list  
  void deleteEnd()  
  {  
  struct node\* temp;  
  if (start == NULL)  
  printf("\nList is empty\n");  
  temp = start;  
  while (temp->next != NULL)  
  temp = temp->next;  
  if (start->next == NULL)  
  start = NULL;  
  else {  
  temp->prev->next = NULL;  
  free(temp);  
  }  
  }  
    
  // Function to delete from any specified  
  // position from the linked list  
  void deletePosition()  
  {  
  int pos, i = 1;  
  struct node \*temp, \*position;  
  temp = start;  
    
  // If DLL is empty  
  if (start == NULL)  
  printf("\nList is empty\n");  
    
  // Otherwise  
  else {  
  // Position to be deleted  
  printf("\nEnter position : ");  
  scanf("%d", &pos);  
    
  // If the position is the first node  
  if (pos == 1) {  
  position = start;  
  start = start->next;  
  if (start != NULL) {  
  start->prev = NULL;  
  }  
  free(position);  
  return;  
  }  
    
  // Traverse till position  
  while (i < pos - 1) {  
  temp = temp->next;  
  i++;  
  }  
  // Change Links  
  position = temp->next;  
  if (position->next != NULL)  
  position->next->prev = temp;  
  temp->next = position->next;  
    
  // Free memory  
  free(position);  
  }  
  }  
    
  // Driver Code  
  int main()  
  {  
  int choice;  
  while (1) {  
    
  printf("\n\t1 To see list\n");  
  printf("\t2 For insertion at"  
  " starting\n");  
  printf("\t3 For insertion at"  
  " end\n");  
  printf("\t4 For insertion at "  
  "any position\n");  
  printf("\t5 For deletion of "  
  "first element\n");  
  printf("\t6 For deletion of "  
  "last element\n");  
  printf("\t7 For deletion of "  
  "element at any position\n");  
  printf("\t8 To exit\n");  
  printf("\nEnter Choice :\n");  
  scanf("%d", &choice);  
    
  switch (choice) {  
  case 1:  
  traverse();  
  break;  
  case 2:  
  insertAtFront();  
  break;  
  case 3:  
  insertAtEnd();  
  break;  
  case 4:  
  insertAtPosition();  
  break;  
  case 5:  
  deleteFirst();  
  break;  
  case 6:  
  deleteEnd();  
  break;  
  case 7:  
  deletePosition();  
  break;  
    
  case 8:  
  exit(1);  
  break;  
  default:  
  printf("Incorrect Choice. Try Again \n");  
  continue;  
  }  
  }  
  return 0;  
  }  
  **Result**
* ****
* **result.JPG**
* **Analysis**
* **complexity.JPG**



DATA STRUCTURES LAB

# Manual

This experiment focus on learning BST concept, operations and traversals.

# Manual

* /\*  
  9 Design, Develop and Implement a Program in C for the following operations  
  on Singly Circular Linked List (SCLL) with header nodes  
  a. Represent and Evaluate a Polynomial P(x,y,z) = 6x2y2z-  
  4yz5+3x3yz+2xy5z-2xyz3  
  b. Find the sum of two polynomials POLY1(x,y,z) and POLY2(x,y,z) and  
  store the result in POLYSUM(x,y,z)  
  Support the program with appropriate functions for each of the above  
  operations.  
  \*/  
    
  #include<stdio.h>  
  #include<stdlib.h>  
  #include<math.h>  
    
  typedef struct poly\_node  
  {  
  float coef;  
  int expx;  
  int expy;  
  int expz;  
  struct poly\_node \*link;  
  } POLY;  
    
  POLY \*getNode();  
  void read\_poly(POLY \*head, int n);  
  void print\_poly(POLY \*head);  
  POLY \*add\_poly(POLY \*h1, POLY \*h2);  
  int compare(POLY \*temp1, POLY \*temp2);  
  void attach(float cf, POLY \*exptemp, POLY \*\*tempres);  
  POLY\* delete(POLY \*head, POLY \*temp);  
  void evaluate(POLY \*head);  
    
  void main()  
  {  
  int n1, n2;  
  POLY \*POLY1 = getNode();  
  POLY \*POLY2 = getNode();  
  POLY \*POLYSUM = getNode();  
    
  POLY1->expx = -1;  
  POLY1->link = POLY1;  
  POLY2->link = POLY2;  
  POLYSUM->link = POLYSUM;  
  printf("\nEnter the number of terms for both polynomials\n");  
  scanf("%d%d",&n1, &n2);  
  printf("\nEnter 1st Polynomial\n");  
  read\_poly(POLY1, n1);  
  printf("\n1st Polynomial is\n");  
  print\_poly(POLY1);  
  printf("\nEnter 2nd Polynomial\n");  
  read\_poly(POLY2, n2);  
  printf("\n2nd Polynomial is\n");  
  print\_poly(POLY2);  
  POLYSUM = add\_poly(POLY1, POLY2);  
  printf("\nThe Resultant polynomial is\n");  
  print\_poly(POLYSUM);  
  evaluate(POLYSUM);  
  }  
    
  POLY \*getNode()  
  {  
  POLY \*temp = (POLY \*) malloc(sizeof(POLY));  
  if(temp == NULL)  
  {  
  printf("No Memory\n");  
  exit(0);  
  }  
  return temp;  
  }  
    
  void read\_poly(POLY \*head, int n)  
  {  
  int i;  
  POLY \*new = NULL;  
  POLY \*temp = head;  
  for(i=0; i<n; i++)  
  {  
  new = getNode();  
  printf("Enter Coef and Exps\n");  
  scanf("%f%d%d%d", &(new->coef), &(new->expx), &(new->expy), &(new->expz));  
  (temp->link) = new;  
  temp = temp->link;  
  }  
  temp->link = head;  
  return;  
  }  
    
  void print\_poly(POLY \*head)  
  {  
  POLY \*temp = head->link;  
  while(temp != head)  
  {  
  printf("%f\*X^%d\*Y^%d\*Z^%d\t", temp->coef, temp->expx, temp->expy, temp->expz);  
  temp = temp->link;  
  }  
  printf("\n");  
  return;  
  }  
    
  POLY \*add\_poly(POLY \*h1, POLY \*h2)  
  {  
  float cf;  
  POLY \*temp1 = h1->link, \*temp2 = NULL;  
  POLY \*result = getNode();  
  POLY \*tempres = result;  
  while(temp1 != h1)  
  {  
  temp2 = h2->link;  
  while(temp2 != h2)  
  {  
  switch(compare(temp1, temp2))  
  {  
  case 1:   
  cf = temp1->coef + temp2->coef;  
  if(cf)  
  {  
  attach(cf, temp1, &tempres);  
  }  
  temp1 = temp1->link;  
  h2 = delete(h2, temp2);  
  temp2 = h2->link;  
  break;  
    
  case 2:   
  temp2 = temp2->link;  
  break;  
  }  
  }  
  if(temp1 != h1)  
  {  
  attach(temp1->coef, temp1, &tempres);  
  temp1 = temp1->link;  
  }  
  }  
  temp2 = h2->link;  
  while(temp2 != h2)  
  {  
  attach(temp2->coef, temp2, &tempres);  
  temp2 = temp2->link;  
  }  
  tempres->link = result;  
  return result;  
  }  
    
  int compare(POLY \*temp1, POLY \*temp2)  
  {  
  if((temp1->expx == temp2->expx) && (temp1->expy == temp2->expy) && (temp1->expz == temp2->expz))  
  {  
  return 1;  
  }  
  return 2;  
  }  
    
  void attach(float cf, POLY \*exptemp, POLY \*\*tempres)  
  {  
  POLY \*new = getNode();  
  new->coef = cf;  
  new->expx = exptemp->expx;  
  new->expy = exptemp->expy;  
  new->expz = exptemp->expz;  
  (\*tempres)->link = new;  
  \*tempres = new;  
  return;  
  }  
    
  POLY\* delete(POLY \*head, POLY \*temp)  
  {  
  POLY \*previous = head, \*present = head->link;  
  while(present != temp)  
  {  
  previous = present;  
  present = present->link;  
  }  
  previous->link = present->link;  
  free(present);  
  return head;  
  }  
    
  void evaluate(POLY \*head)  
  {  
  float result = 0.0;  
  int x,y,z;  
  POLY \*temp = head->link;  
  printf("\nEnter exponents\n");  
  scanf("%d%d%d", &x, &y, &z);  
  while(temp != head)  
  {  
  result += (temp->coef)\*pow(x, temp->expx)\*pow(y, temp->expy)\*pow(z, temp->expz);  
  temp = temp->link;  
  }  
  printf("\nResult after evaluation is %f\n", result);  
  return;  
  }

Unit 3

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

This experiment is targeted to implement Graph data structures to solve a given problem.

# Manual (Algorithm, Code and Analysis)

* **Problem Statement**Design, Develop and Implement a menu driven Program in C for the following operations on Binary Search Tree (BST) of Integers .
  + Create a BST of N Integers: 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2
  + Traverse the BST in Inorder, Preorder and Post Order
  + Search the BST for a given element (KEY) and report the appropriate message
  + Exit
* **Introduction**Binary Search Tree is a node-based binary tree data structure which has the following properties:
  + The left subtree of a node contains only nodes with keys lesser than the node’s key.
  + The right subtree of a node contains only nodes with keys greater than the node’s key.
  + The left and right subtree each must also be a binary search tree.
* Thus, a binary search tree (BST) divides all its sub-trees into two segments; left sub-tree and right sub-tree and can be defined as  
  left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)  
  Primary operations of a binary search tree are following.
  + **Search** − search an element in a tree.
  + **Insert** − insert an element in a tree.
  + **Preorder Traversal** − traverse a tree in a preorder manner.
  + **Inorder Traversal** − traverse a tree in an inorder manner.
  + **Postorder Traversal** − traverse a tree in a postorder manner
* **Define a node having some data, references to its left and right child nodes.**struct node  
  {  
  int data;  
  struct node \*leftChild;  
  struct node \*rightChild;  
  };  
  **ALGORITHM:**Step 1: Start.  
  Step 2: Create a Binary Search Tree for N elements.  
  Step 3: Traverse the tree in inorder.  
  Step 4: Traverse the tree in preorder  
  Step 6: Traverse the tree in postorder.  
  Step 7: Search the given key element in the BST.  
  Step 8: Delete an element from BST.  
  Step 9: Stop  
  **Program**#include <stdio.h>  
    
  #include <stdlib.h>  
    
  struct BST  
    
  {  
    
  int data;  
    
  struct BST \*left;  
    
  struct BST \*right;  
    
  };  
    
  typedef struct BST NODE;  
    
  NODE \*node;  
    
  NODE\* createtree(NODE \*node, int data)  
    
  {  
    
  if (node == NULL)  
    
  {  
    
  NODE \*temp;  
    
  temp= (NODE\*)malloc(sizeof(NODE));  
    
  temp->data = data;  
    
  temp->left = temp->right = NULL;  
    
  return temp;  
    
  }  
    
  if (data < (node->data))  
    
  {  
    
  node->left = createtree(node->left, data);  
    
  }  
    
  else if (data > node->data)  
    
  {  
    
  node -> right = createtree(node->right, data);  
    
  }  
    
  return node;  
    
  }  
    
  NODE\* search(NODE \*node, int data)  
    
  {  
    
  if(node == NULL)  
    
  printf("\nElement not found");  
    
  else if(data < node->data)  
    
  {  
    
  node->left=search(node->left, data);  
    
  }  
    
  else if(data > node->data)  
    
  {  
    
  node->right=search(node->right, data);  
    
  }  
    
  else  
    
  printf("\nElement found is: %d", node->data);  
    
  return node;  
    
  }  
    
  void inorder(NODE \*node)  
    
  {  
    
  if(node != NULL)  
    
  {  
    
  inorder(node->left);  
    
  printf("%d\t", node->data);  
    
  inorder(node->right);  
    
  }  
    
  }  
    
  void preorder(NODE \*node)  
    
  {  
    
  if(node != NULL)  
    
  {  
    
  printf("%d\t", node->data);  
    
  preorder(node->left);  
    
  preorder(node->right);  
    
  }  
    
  }  
    
  void postorder(NODE \*node)  
    
  {  
    
  if(node != NULL)  
    
  {  
    
  postorder(node->left);  
    
  postorder(node->right);  
    
  printf("%d\t", node->data);  
    
  }  
    
  }  
    
  NODE\* findMin(NODE \*node)  
    
  {  
    
  if(node==NULL)  
    
  {  
    
  return NULL;  
    
  }  
    
  if(node->left)  
    
  return findMin(node->left);  
    
  else  
    
  return node;  
    
  }  
    
  NODE\* del(NODE \*node, int data)  
    
  {  
    
  NODE \*temp;  
    
  if(node == NULL)  
    
  {  
    
  printf("\nElement not found");  
    
  }  
    
  else if(data < node->data)  
    
  {  
    
  node->left = del(node->left, data);  
    
  }  
    
  else if(data > node->data)  
    
  {  
    
  node->right = del(node->right, data);  
    
  }  
    
  else  
    
  { /\* Now We can delete this node and replace with either minimum element in the right sub tree or maximum element in the left subtree \*/  
    
  if(node->right && node->left)  
    
  { /\* Here we will replace with minimum element in the right sub tree \*/  
    
  temp = findMin(node->right);  
    
  node -> data = temp->data;  
    
  /\* As we replaced it with some other node, we have to delete that node \*/  
    
  node -> right = del(node->right,temp->data);  
    
  }  
    
  else  
    
  {  
    
  /\* If there is only one or zero children then we can directly remove it from the tree and connect its parent to its child \*/  
    
  temp = node;  
    
  if(node->left == NULL)  
    
  node = node->right;  
    
  else if(node->right == NULL)  
    
  node = node->left;  
    
  free(temp); /\* temp is longer required \*/  
    
  }  
    
  }  
    
  return node;  
    
  }  
    
  int main()  
    
  {  
    
  int data, ch, i, n;  
    
  NODE \*root=NULL;  
    
  while (1)  
    
  {  
    
  printf("\n1.Insertion in Binary Search Tree");  
    
  printf("\n2.Search Element in Binary Search Tree");  
    
  printf("\n3.Delete Element in Binary Search Tree");  
    
  printf("\n4.Inorder\n5.Preorder\n6.Postorder\n7.Exit");  
    
  printf("\nEnter your choice: ");  
    
  scanf("%d", &ch);  
    
  switch (ch)  
    
  {  
    
  case 1: printf("\nEnter N value: " );  
    
  scanf("%d", &n);  
    
  printf("\nEnter the values to create BST like(6,9,5,2,8,15,24,14,7,8,5,2)\n");  
    
  for(i=0; i<n; i++)  
    
  {  
    
  scanf("%d", &data);  
    
  root=createtree(root, data);  
    
  }  
    
  break;  
    
  case 2: printf("\nEnter the element to search: ");  
    
  scanf("%d", &data);  
    
  root=search(root, data);  
    
  break;  
    
  case 3: printf("\nEnter the element to delete: ");  
    
  scanf("%d", &data);  
    
  root=del(root, data);  
    
  break;  
    
  case 4: printf("\nInorder Traversal: \n");  
    
  inorder(root);  
    
  break;  
    
  case 5: printf("\nPreorder Traversal: \n");  
    
  preorder(root);  
    
  break;  
    
  case 6: printf("\nPostorder Traversal: \n");  
    
  postorder(root);  
    
  break;  
    
  case 7: exit(0);  
    
  default:printf("\nWrong option");  
    
  break;  
    
  }  
    
  }  
    
  return 0 ;  
  }

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

This experiment is based on understanding on Hashing Concept and use it to provide effective search.

# Manual (Algorithm, Code and Analysis)

* **Design, Develop and Implement a Program in C for the following operations on Graph(G) of Cities:  
  a. Create a Graph of N cities using Adjacency Matrix.  
   b. Print all the nodes reachable from a given starting node in a digraph using DFS/BFS method  
    
  Code**#include <stdio.h>  
  #include <stdlib.h>  
  int a[20][20],q[20],visited[20],reach[20],n,f=0,r=-1,count=0;  
  void bfs(int v)  
  {  
   int i;  
   for(i=1;i<=n;i++)  
   if(a[v][i]&&!visited[i])  
   {  
   visited[i]=1;  
   q[++r]=i;  
   }  
   if(f<=r)  
   bfs(q[f++]);  
  }  
  void dfs(int v)  
  {  
   int i;  
   reach[v]=1;  
   for(i=1;i<=n;i++)  
   if(a[v][i]&&!reach[i])  
   {  
   printf("%d->%d\n",v,i);  
   count++;  
   dfs(i);  
   }  
  }  
  int main()  
  {  
   int v,ch,i,j;  
   printf("\nenter no. of vertices:");  
   scanf("%d",&n);  
   for(i=1;i<=n;i++)  
   reach[i]=visited[i]=q[i]=0;  
   printf("\nEnter graph data in matrix form:\n");  
   for(i=1;i<=n;i++)  
   for(j=1;j<=n;j++)  
   scanf("%d",&a[i][j]);  
   printf("\n1.BFS\n2.DFS\n3.Exit\nEnter choice:");  
   scanf("%d",&ch);  
   switch(ch)  
   {  
   case 1:printf("\nEnter vertex:");  
   scanf("%d",&v);  
   bfs(v);  
   printf("\nThe nodes that are reacheble from %d are:\n",v);  
   for(i=1;i<=n;i++)  
   if(visited[i])  
   printf("%d ",i);  
   break;  
   case 2:dfs(1);  
   if(count==n-1)  
   printf("\ngraph is connected");  
   else  
   printf("\ngraph is not connected");  
   break;  
   case 3:exit(0);  
   default:printf("\nInvalid choice");  
   }  
   return 0;  
    
  }

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

This experiment will apply tree data structure for Heap Sort implementation

# Manual (Algorithm, Code and Analysis)

* **Problem Statement:  
    
  Code**#include<stdio.h>  
  #include<stdlib.h>  
    
  int key[20],n,m;  
  int \*ht,index;  
  int count = 0;  
    
  void **insert(int key)**{  
   index = key % m;  
   while(ht[index] != -1)  
   {  
   index = (index+1)%m;  
   }  
   ht[index] = key;  
   count++;  
   }  
    
  void **display()**{  
   int i;  
   if(count == 0)  
   {  
   printf("\nHash Table is empty");  
   return;  
   }  
    
   printf("\nHash Table contents are:\n ");  
   for(i=0; i<m; i++)  
   printf("\n T[%d] --> %d ", i, ht[i]);  
  }  
    
    
  void main()  
  {  
   int i;  
   printf("\nEnter the number of employee records (N) : ");  
   scanf("%d", &n);  
    
   printf("\nEnter the two digit memory locations (m) for hash table: ");  
   scanf("%d", &m);  
    
   ht = (int \*)malloc(m\*sizeof(int));  
   for(i=0; i<m; i++)  
   ht[i] = -1;  
    
   printf("\nEnter the four digit key values (K) for N Employee Records:\n ");  
   for(i=0; i<n; i++)  
   scanf("%d", &key[i]);  
    
   for(i=0;i<n;i++)  
   {  
   if(count == m)  
   {  
   printf("\n~~~Hash table is full. Cannot insert the record %d key~~~",i+1);  
   break;  
   }  
   insert(key[i]);  
   }  
    
   //Displaying Keys inserted into hash table  
   **display();**}  
    
  **Output  
  *Output:***Enter the number of employee records (N) : **12**Enter the two digit memory locations (m) for hash table: **15**Enter the four digit key values (K) of 'N' Employee Records:  
  **1234  
  5678  
  3456  
  2345  
  6799  
  1235  
  7890  
  3214  
  3456  
  1235  
  5679  
  2346**Hash Table contents are:  
    
   **T[0] --> 7890  
   T[1] --> -1  
   T[2] --> -1  
   T[3] --> -1  
   T[4] --> 1234  
   T[5] --> 2345  
   T[6] --> 3456  
   T[7] --> 6799  
   T[8] --> 5678  
   T[9] --> 1235  
   T[10] --> 3214  
   T[11] --> 3456  
   T[12] --> 1235  
   T[13] --> 5679  
   T[14] --> 2346**

DATA STRUCTURES LAB

# Manual (Algorithm, Code and Analysis)

# Manual (Algorithm, Code and Analysis)

* **Algorithm  
  Heap Sort Algorithm for sorting in increasing order:**   
  **1.** Build a max heap from the input data.   
  **2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.   
  **3.** Repeat step 2 while the size of the heap is greater than 1.  
    
  **Program**#include <iostream>  
    
  using namespace std;  
    
  void heapify(int arr[], int n, int i)  
  {  
  int largest = i; // Initialize largest as root  
  int l = 2 \* i + 1; // left = 2\*i + 1  
  int r = 2 \* i + 2; // right = 2\*i + 2  
    
  if (l < n && arr[l] > arr[largest])  
  largest = l;  
    
    
  if (r < n && arr[r] > arr[largest])  
  largest = r;  
    
    
  if (largest != i) {  
  swap(arr[i], arr[largest]);  
    
    
  heapify(arr, n, largest);  
  }  
  }  
    
    
  void heapSort(int arr[], int n)  
  {  
    
  for (int i = n / 2 - 1; i >= 0; i--)  
  heapify(arr, n, i);  
    
    
  for (int i = n - 1; i > 0; i--) {  
  // Move current root to end  
  swap(arr[0], arr[i]);  
    
  heapify(arr, i, 0);  
  }  
  }  
    
  void printArray(int arr[], int n)  
  {  
  for (int i = 0; i < n; ++i)  
  cout << arr[i] << " ";  
  cout << "\n";  
  }  
    
  int main()  
  {  
  int arr[] = { 12, 11, 13, 5, 6, 7 };  
  int n = sizeof(arr) / sizeof(arr[0]);  
    
  heapSort(arr, n);  
    
  cout << "Sorted array is \n";  
  printArray(arr, n);  
  }  
  **Output**Sorted array is   
  5 6 7 11 12 13

Mst 1st

