SURESH ANGADI EDUCATION FOUNDATION'S
ANGADI INSTITUTE OF TECHNOLOGY AND MANAGEMENT
Savagaon Road, BELAGAVI – 590 009.
(Approved by AICTE, New Delhi &Affiliated to Visvesvaraya Technological University, Belagavi, Accredited by NAAC)
Department of Artificial Intelligence and Data Science

LABORATORY MANUAL

Data Structures Lab (2022 Scheme)

Subject Code : BCSL305

Prepared By:

Prof.D.M.Choudhari
Assistant Professor Dept.AI & DS

Campus :Savagaon Road, Belagavi – 590 009. ☎: 0831 – 2438100, 2438123, Fax: 0831-2438197
Web Site: www.aitmbgm.ac.in, E-mail: director.aitm@gmail.com

VISION

To deliver a quality and responsive education in the field of artificial intelligence and data science emphasizing professional skills to face global challenges in the evolving IT paradigm.

MISSION

- Leverage multiple pedagogical approaches to impart knowledge on the current and emerging AI technologies.
- Develop an inclusive and holistic ambiance that bolsters problem solving, cognitive abilities and critical thinking.
- Enable students to develop trust worthiness, team spirit, understanding law-of-the-land, social behaviour to be a global stake holder

PROGRAM OUTCOMES

PO 1: Engineering Knowledge: Apply the Knowledge of Mathematics, Science, Engineering Fundamentals, and an Engineering specialization to the solution of complex Engineering problems.

PO 2: Problem Analysis: Identify, Formulate, Review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of Mathematics, natural sciences and engineering sciences.

PO 3: Design/Development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental conditions.

PO 4: Conduct investigations on complex problems: Use research based knowledge and research methods including design of Experiments, analysis and interpretation of data, and synthesis of Information to provide valid conclusions.

PO 5: Modern tool usage: Create, select, and apply appropriate technique, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6: The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess society, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO 11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12: Lifelong learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Learning objectives:

CLO 1. To explain fundamentals of data structures and their applications.

CLO 2. To illustrate representation of Different data structures such as Stack, Queues, Linked Lists, Trees and Graphs.

CLO 3. To Design and Develop Solutions to problems using Linear Data Structures

CLO 4. To discuss applications of Nonlinear Data Structures in problem solving.

CLO 5. To introduce advanced Data structure concepts such as Hashing and Optimal Binary Search Trees

Course Outcomes:

CO 1. Explain different data structures and their applications.

CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.

CO 3. Use the concept of linked list in problem solving.

CO 4. Develop solutions using trees and graphs to model the real-world problem.

CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees.

EXPERIMENT NO-1

AIM:

To Design, Develop and Implement a menu driven Program in C for the following Array operations.

a. Creating an Array of N Integer Elements

b. Display of Array Elements with Suitable Headings

c. Inserting an Element (ELEM) at a given valid Position (POS)

d. Deleting an Element at a given valid Position (POS)

e. Exit

THEORY:

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. To create an array, define the data type (like int) and specify the name of the array followed by square brackets [].

Design, Develop and Implement a menu driven Program in C for the following array operations.

- Creating an array of N Integer Elements
- Display of array Elements with Suitable Headings
- Inserting an Element (ELEM) at a given valid Position (POS)
- Deleting an Element at a given valid Position (POS)
- Exit.

DISPLAY():

1. Step 1 → Take the array arr[] and define its values, that is define the value for each element of the array arr[].
2. Step 2 → Start a loop for each value of the array arr[].
3. Step 3 → Print the value associated with arr[i] where i is the current iteration.

INSERT():

Inserting elements in an array using C Language

1. Enter the size of the array.
2. Enter the position where you want to insert the element.
3. Next enter the number that you want to insert in that position.

DELETE():

1. Move to the specified location which you want to remove in given array.

2. Copy the next element to the current element of array. Which is you need to perform array[i] = array[i + 1] .
3. Repeat above steps till last element of array.
4. Finally decrement the size of array by one.

EXIT():

The exit() function is used to terminate a process or function calling immediately in the program. It means any open file or function belonging to the process is closed immediately as the exit() function occurred in the program. The exit() function is the standard library function of the C, which is defined in the stdlib.h header file. So, we can say it is the function that forcefully terminates the current program and transfers the control to the operating system to exit the program. The exit(0) function determines the program terminates without any error message, and then the exit(1) function determines the program forcefully terminates the execution process.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int a[4],n,elem,i,pos;
void create()
{
   printf("\nEnter the size of array elements:");
   scanf("%d",&n);
   printf("Enter the elements of the array:\n");
   for(i=0;i<n;i++)
      scanf("%d",&a[i]);
}
void display()
{
   int i;
   printf("\nThe array elements are:");
   for(i=0;i<n;i++)
```

```
    {
        printf("%d\t",a[i]);
    }
}
void insert()
```

```
{
    int temp;
    printf("Enter the position of the new element:");
    scanf("%d",&pos);
    printf("\nEnter the element to be inserted:");
    scanf("%d",&elem);
    for(i=n-1;i>=pos;i--)
    {
        a[i+1]=a[i];
    }
    a[pos]=elem;
    n=n+1;
}
void del()
{
    printf("\nEnter the position of the element to be deleted");
    scanf("%d",&pos);
    elem=a[pos];
    for(i=pos;i<n-1;i++)
    {
        a[i]=a[i+1];
    }
    n=n-1;
    printf("\nThe deleted element is=%d",elem);
}
void main()
{
```

```
int ch;
do{
    printf("\n\n...Menu...\n");
    printf("1.create\n2.display3.insert\n4.delete\n5.exit\n");
    printf("--------------");
```

```
        printf("\nenter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
          case 1:create();
          break;
          case 2:display();
          break;
          case 3:insert();
          break;
          case 4:del();
          break;
          case 5:exit(0);
          break;
          default:printf("invalid choice:\n");
          break;
        }
    }while(ch!=5);
}
```

OUTPUT:

```
_____MENU________
1.    CREATE
2.    DISPLAY
3.    INSERT
4.    DELETE
5.    EXIT
----------------------------

ENTER YOUR CHOICE: 1
Enter the size of the array elements: 3 Enter the elements for
the array:10 25 30
```

ENTER YOUR CHOICE: 2
The array elements are:
10 25 30

ENTER YOUR CHOICE: 3
Enter the position for the new element: 1 Enter the element to be inserted : 20

ENTER YOUR CHOICE: 2

The array elements are:
10 20 25 30

ENTER YOUR CHOICE: 4
Enter the position of the element to be deleted: 3 The deleted element is =30

enter your
choice: 5Exit

EXPERIMENT NO-02

AIM:

Design, Develop and implement a menu driven program in c for the following operations on STACK of integers

a. Push an element on stack.

b. Pop an element on stack.

c. Demonstrate how stack can be used to check palindrome.

d. Demonstrate overflow and underflow situations on stack.

e. Display the status of stack.

f. Exit.

THEORY:

A stack is a logical concept that consists of a set of similar elements. The term is often used in programming and memory organization in computers. Programming stacks are based on the principle of last in first out (LIFO), a commonly used type of data abstract that consists of two major operations, push and pop.

A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

Push an element

1. STEP 1: START.
2. STEP 2: Store the element to push into array.
3. STEP 3: Check if top== (MAXSIZE-1) then stack is full else goto step
4. STEP 4: Increment top as top = top+1.
5. STEP 5: Add element to the position stk[top]=num.
6. STEP 6:STOP.

pop: Removes the topmost element from the stack. isEmpty : Checks whether the stack is empty. isFull : Checks whether the stack is full. top: Displays the topmost element of the stack.

Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

Step 3 − If the stack is not empty, accesses the data element at which top is pointing.

Step 4 − Decreases the value of top by 1.

Step 5 − Returns success.

Demonstrate how stack can be used to check palindrome:

If the length of the string is odd then neglect the middle character. Till the end of the string, keep popping elements from the stack and compare them with the current character i.e. string[i]. If there is a mismatch then the string is not a palindrome. If all the elements match then the string is a palindrome.

Demonstrate overflow and underflow situations on stack:

Integer Overflow occurs when we attempt to store a value greater than the data type's largest value. Similarly, Integer Underflow occurs when we attempt to store a value that is less than the least value of the data type. We can detect these overflows and underflows either mathematically (or) programmatically.

DISPLAY:

The display() function displays all the elements in the stack. It uses a for loop to do so. If there are no elements in the stack, then Stack is empty is printed.

EXIT():

exit() exits from the program, not just from the current function so other instructions will not be executed. Save this answer.

PROGRAM:

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

#define STACK_SIZE 5

int top=-1,stack[STACK_SIZE];
void push();
void pop();
void display();

void main()
{
    int ch;
```

```
while(1)
{
    printf("\n***Stack menu***");
    printf("\n\n1.Push\n2.Pop\n3.Display\n4.Exit");
    printf("\n\nEnter your choice(1-4):");
    scanf("%d",&ch);
```

```
        switch(ch)
        {
            case 1:push();
            break;
            case 2:pop();
            break;
            case 3:display();
            break;
            case 4:exit(0);
            break;
            default:printf("\nWrong Choice!!");
        }
    }
}
void push()
{
    int val;
    if(top==STACK_SIZE-1)
    {
        printf("\nStack is full!!");
    }
    else
    {
        printf("\nEnter element to push:");
        scanf("%d",&val);
        top=top+1;
        stack[top]=val;
```

```
    }
}
void pop()
{
    if(top==-1)
    {
```

```
        printf("\nStack is Empty!!");
    }
    else
    {
        printf("\nDeleted element is %d",stack[top]);
        top=top-1;
    }
}
void Display()
{
    int i;
    if(top==-1)
    {
        printf("\nStack is empty!!");
    }
    else
    {
        printf("\nStack is...\n");
        for(i=top;i>=0;--i)
            printf("%d\n",stack[i]);
    }
}
```

OUTPUT:

```
--------STACK OPERATIONS-----------
1.Push 2.Pop
3.Palindrome 4.Display 5.Exit
----------------------------
```

Enter your choice: 1
Enter the element to be inserted: 1

Enter your choice: 1
Enter the element to be inserted: 2

Enter your choice: 1
Enter the element to be inserted: 1

Enter your choice: 1
Enter the element to be inserted: 5

Enter your choice: 2 The poped element: 5

Enter your choice: 4 The stack
elements are: 12
1
Enter your choice: 3 Numbers= 1
Numbers= 2
Numbers= 1 reverse operation :
reverse array : 12
1
check for palindrome : It is palindrome number

Enter your choice: 5
Exit

EXPERIMENT NO-3

AIM:

To Design, Develop and implement a program in c for converting an Infix Expression to Postfix Expression Program should support for both parenthesized and free parenthesized expressions with the operators +,-,*,/,% (Remainder),^(Power) and alphanumeric operands.

THEORY:

1. Infix Expression
2. Prefix Expression
3. Postfix Expression

1. Infix:

The expression of the form a operator b (a + b). When an operator is in-between every pair of operands.

2. Prefix:

An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2). Example : *+AB-CD (Infix : (A+B) * (C-D))

3. Postfix:

In a postfix expression, • an operator is written after its operands.

• The infix expression 2+3 is 23+ in postfix notation.

• For postfix expressions, operations are performed in the order in which they are written (left to right).

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

1. Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.
2. Step 2: Obtain the "nearly" postfix expression of the modified expression i.e CB*A+.

PROGRAM:

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

```
int F(char symbol)
{
    switch(symbol)
    {
    case '+':
    case '-':
        return 2;
    case '*':
    case '/':
        return 4;
    case '^':
    case '$':return 5;
    case '(':return 0;
    case '#':return -1;
    default: return 8;
    }
}
int G(char symbol)
{
    switch(symbol)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 3;
```

```
case '^':
case '$':return 6;
case '(':return 9;
case '#':return 0;
default:return 7;
}
```

```
}
void infix_postfix(char infix[],char postfix[])
{
    int top,j,i;
    char s[30],symbol;
    top=-1;
    s[++top]='#';
    j=0;
    for(i=0;i<strlen(infix);i++)
    {
        symbol=infix[i];
        while(F(s[top])>G(symbol))
        {
            postfix[j]=s[top--];
            j++;
        }
        if(F(s[top])!=G(symbol))
            s[++top]=symbol;
        else
            top--;
    }
    while(s[top]!='#')
    {
        postfix[j++]=s[top--];
    }
    postfix[j]='\0';
    strrev(postfix);
```

```
}
void main()
{
    char infix[20];
    char postfix[20];
    printf("Enter a valid infix expression\n");
```

```
    scanf("%s",infix);
    infix_postfix(infix,postfix);
    printf("\nthe infix expression is:\n");
    printf("%s",infix);
    printf("\nThe postfix expression is:\n");
    printf("%s",postfix);
}
```

OUTPUT:

```
Enter a valid infix expression
(A+(B-P)+D)

the infix expression is:
(A+(B-P)+D)
The postfix expression is:
ABP-D*
Process returned 11 (0xB)   execution time : 18.258 s
Press any key to continue.
```

EXPERIMENT NO-4

AIM:

To Design, Develop and implement a program in c for converting an Infix Expression to Prefix Expression Program should support for both parenthesized and free parenthesized expressions with the operators +,-,*,/,% (Remainder),^(Power) and alphanumeric operands.

THEORY:

Infix Expression:

Infix notation is the notation commonly used in arithmetical and logical formulae and statements. It is characterized by the placement of operators between operands—"infixed operators"—such as the plus sign in 2 + 2.

Prefix Expression:

An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Conversion from Infix to Prefix Expression:

1. Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.
2. Step 2: Obtain the "nearly" postfix expression of the modified expression i.e CB*A+.
3. Step 3: Reverse the postfix expression.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
int F(char symbol)
{
   switch(symbol)
   {
   case '+':
```

```
case '-':
    return 2;
case '*':
case '/':
```

```
        return 4;
    case '^':
    case '$':return 5;
    case '(':return 0;
    case '#':return -1;
    default: return 8;
    }
}
int G(char symbol)
{
    switch(symbol)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 3;
    case '^':
    case '$':return 6;
    case '(':return 9;
    case '#':return 0;
    default:return 7;
    }
}
void infix_prefix(char infix[],char prefix[])
{
```

```
int top;
char s[30];
int j;
int i;
char symbol;
top=-1;
```

```
    s[++top]='#';
    j=0;
    strrev(infix);
    for(i=0;i<strlen(infix);i++)
    {
        symbol=infix[i];
        while(F(s[top])>G(symbol))
        {
            prefix[j]=s[top--];
            j++;
        }
        if(F(s[top])!=G(symbol))
            s[++top]=symbol;
        else
            top--;
    }
    while(s[top]!='#')
    {
        prefix[j++]=s[top--];
    }
    prefix[j]='\0';
    strrev(prefix);
}
void main()
{
    char infix[20];
    char prefix[20];
```

```
    printf("Enter a valid infix expression\n");
    scanf("%s",infix);
    infix_prefix(infix,prefix);
    printf("the prefix expression\n");
    printf("%s\n",prefix);
}
```

OUTPUT:

Enter a valid infix expression

(a+b)*(c+d)

the prefix expression

ab+c*

Process returned 0 (0x0)   execution time : 84.895 s

Press any key to continue.

EXPERIMENT NO:05

AIM:

Evaluation of postfix Expression

THEORY:

If it is a digit then, push it on to the stack. If it is an operator then, pop out the top most two contents from the stack and apply the operator on them. Later on, push the result on to stack. If the input symbol is '\0', empty the stack

Postfix notation is one of the few ways to represent algebraic expressions. It is used in computers because it is faster than other types of notations (such as infix notation) as parentheses are not required to represent them.

As the name suggests, in the postfix expression operators follow their operands. Therefore, the process of postfix evaluation is quite different than solving the infix notation (normal notation used in daily use).

The algorithm for evaluation of postfix expression is as follows -

- Create a stack that holds integer type data to store the operands of the given postfix expression. Let it be st.
- Iterate over the string from left to right and do the following -
  - If the current element is an operand, push it into the stack.
  - Otherwise, if the current element is an operator (say /)do the following -
    - Pop an element from st, let it be op1.
    - Pop another element from st, let it be op2.
    - Computer the result of op2 / op1, and push it into the stack. Note the order �.�.*i.e*. op2 / op1 should not be changed otherwise it will affect the final result in some cases.
- At last, st will consist of a single element �.�.*i.e*. the result after evaluating the postfix expression.

Let the the postfix expression be "10 22 + 8 / 6 * 5 +". Now, proceeding as per the algorithm -

1. Create a stack (say st).
2. Now since the first element is an operand, so we will push it in the stack �.�.*i.e*. st.push(10). After which stack looks like -
3. Now we traverse further in the postfix expression and found that the next element is again an operand. So, we will push it into the stack, after which the stack will look like -

st
22
10

4. The next element of the expression is an operator (+ operator) so we will do the following -
   1. Pop an element (say op1) from the stack. Here op1 = 22.
   2. Pop another element (say op2) from the stack. Here op2 = 10.
   3. Now, we will compute op2 + op1 which is 32, and push it into the stack. After this step stack will look like -

st
32

5. On traversing further in the expression string, we found next element is an operand. So we will push it into the stack. Upon doing this, the stack will look like -

st
8
32

6. The next element is the / operator, so we will do the following -

1. Pop an element (say op1) from the stack. Here op1 = 8.
2. Pop another element (say op2) from the stack. Here op2 = 32.
3. Now, we will compute op2 / op1 which is 4, and push it into the stack. After this step stack will look like -

st
4

7. The next element is an operand, hence pushing it in the stack. After this step stack will look like -

st
6
4

8. The next element is the * operator, hence we need to perform the following steps -

1. Pop an element (say op1) from the stack. Here op1 = 6.
2. Pop another element (say op2) from the stack. Here op2 = 4.
3. Now, we will compute op2 * op1 which is 24, and push it into the stack. After this step stack will look like -

st
24

9. The next operator is an operand, hence pushing it into the stack. Upon doing which stack will look like -

st
5
24

10. The last element is the + operator. Hence it is required to do the following steps -
    1. Pop an element (say op1) from the stack. Here op1 = 5.
    2. Pop another element (say op2) from the stack. Here op2 = 24.
    3. Now, we will compute op2 + op1 which is 29, and push it into the stack. After this step stack will look like -

st
29

11. Now, we have traversed the given postfix expression, and hence as

PROGRAM:

```
#include  <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>


// Stack type
```

```
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    if (!stack->array) return NULL;

    return stack;
```

```
}

int isEmpty(struct Stack* stack)
{
```

```
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}
 int evaluatePostfix(char* exp)
```

```
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;
```

```
// See if stack was created successfully
if (!stack) return -1;

// Scan all characters one by one
for (i = 0; exp[i]; ++i)
{
    // If the scanned character is an operand (number here),
    // push it to the stack.
    if (isdigit(exp[i]))
        push(stack, exp[i] - '0');

    // If the scanned character is an operator, pop two
    // elements from stack apply the operator
    else
    {
        int val1 = pop(stack);
        int val2 = pop(stack);
        switch (exp[i])
        {
```

```
case '+': push(stack, val2 + val1); break;

case '-': push(stack, val2 - val1); break;

case '*': push(stack, val2 * val1); break;

case '/': push(stack, val2/val1); break;
```

```
            }
        }
    }
    return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "231*+9-";
    printf ("postfix evaluation: %d", evaluatePostfix(exp));
    return 0;
}
```

OUTPUT:

```
postfix evaluation: -4
Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```

EXPERIMENT NO-6

AIM:

To Design, Develop and implement a meant menu driven program in c for the following operations

a. Insert an element on the Queue.

b. Delete an element on the Queue.

c. Display an element on the Queue.

d. Exit

THEORY:

Queue: A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue. See the below figure.

Three types of Queues:

1. Linear Queue

2. Circular Queue

3. Double ended Queue

1. Linear Queue:

A Linear Queue is generally referred to as Queue. It is a linear data structure that follows the FIFO (First In First Out) order. A real-life example of a queue is any queue of customers waiting to buy a product from a shop where the customer that came first is served first.

Arranges the data in a linear pattern. The insertion and deletion operations are fixed i.e, done at the rear and front end respectively. Linear queue requires more memory space. In the case of a linear queue, the element added in the first position is going to be deleted in the first position. The order of operations performed on any element is fixed i.e, FIFO. In a linear queue, we can easily fetch out the peek value.

2. Circular Queue:

A Circular Queue is a special version of queue where the last element of the queue is connected to the first element of the queue forming a circle. The operations are performed based on FIFO (First In First Out) principle. It is also called 'Ring Buffer'.

Arranges the data in a circular order where the rear end is connected with the front end. Insertion and deletion are not fixed and it can be done in any position. It requires less memory space. In the case of circular queue, the order of operations performed on an element may change. In a circular queue, we cannot fetch out the peek value easily.

3. Double ended Queue:

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define MAX 4
int ch,front=0,rear=-1,count=0;
char q[MAX],item;
void insert()
{
    if(count==MAX)
        printf("\nQueue is full");
    else
    {
        rear=(rear+1)%MAX;
        q[rear]=item;
        count++;
    }
}
```

```
void del()
{
    if(count==0)
        printf("\nQueue is Empty");
    else
    {
```

```
        if(front>rear&&rear==MAX-1)
        {
            front=0;   rear=-1;   count=0;
        }
        else
        {
            item=q[front];
            printf("\nDeleted item is:%c",item);
            front=(front+1)%MAX;
            count--;
        }
    }
}
void display()
{
    int i,f=front,r=rear;
    if(count==0)
        printf("\nQueue is empty");
    else
    {
        printf("\nContents of Queue is:\n");
        for(i=f;i<=r;i++)
        {
            printf("%c\t",q[i]);
            f=(f+1)%MAX;
        }
    }
```

```
}
void main()
{
    do
    {
        printf("\n1.insert\n2.Delete\n3.Display\n4.Exit");

        printf("\nEnter the choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:printf("\nEnter the character/item to be inserted:");
            scanf("%s",&item);
            insert();
            break;
            case 2:del();
            break;
            case 3:display();
            break;
            case 4:exit(0);
            break;
        }
    }while(ch!=4);
}
```

OUTPUT:

```
1.insert
2.Delete
3.Display
```

4.Exit

Enter the choice:1

Enter the character/item to be inserted:1

1.insert

2.Delete

3.Display

4.Exit

Enter the choice:1

Enter the character/item to be inserted:2

1.insert

2.Delete

3.Display

4.Exit

Enter the choice:1

Enter the character/item to be inserted:3

1.insert

2.Delete

3.Display

4.Exit

Enter the choice:1

Enter the character/item to be inserted:4

1.insert

2.Delete

3.Display

4.Exit

Enter the choice:1

Enter the character/item to be inserted:5

Queue is full
1.insert
2.Delete

3.Display
4.Exit
Enter the choice:3

Contents of Queue is:

```
1       2       3       4
1.insert
2.Delete
3.Display
4.Exit
Enter the choice:2

Deleted item is:1
1.insert
2.Delete
3.Display
4.Exit
Enter the choice:3

Contents of Queue is:
2       3       4
1.insert
2.Delete
3.Display
4.Exit
Enter the choice:4

Process returned 0 (0x0)   execution time : 70.200 s
Press any key to continue.
```

EXPERIMENT NO-7

AIM:

To Design and Develop and implement a menu driven program in c for the following expressions on Singly linked list (SLL) of student data with the fields: USN, Name, Branch, Sem, Phone No.

THEORY:

A Linked List is a linear data structure. Every linked list has two parts, the data section and the address section that holds the address of the next element in the list, which is called a node.

The size of the linked list is not fixed, and data items can be added at any locations in the list. The disadvantage is that to get to a node, we must traverse to all the way from the first node to the node that we require. The Linked List is like an array but unlike an array, it is not stored sequentially in the memory.

The most popular types of a linked list are:

1. Singly link list
2. Doubly link list

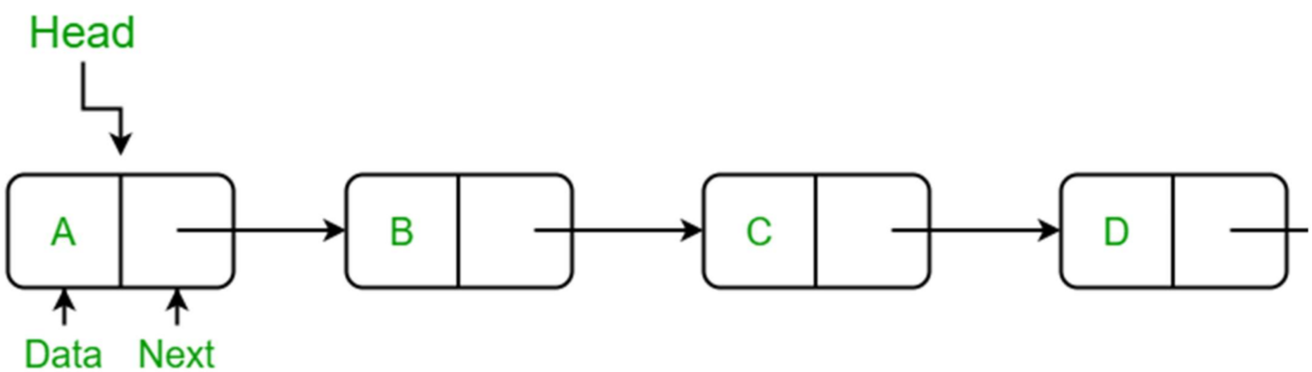
Example of Linked List

Format:[data,address]

Head->[3,1000]->[43,1001]->[21,1002]

In the example, the number 43 is present at location 1000 and the address is present at in the previous node. This is how a linked list is represented.

Like arrays, a Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes. Here, each node stores the data and the address of the next node.



Advantages of Linked Lists over arrays:

- Dynamic Array.
- Ease of Insertion/Deletion.

Types of Linked Lists:

- Simple Linked List – In this type of linked list, one can move or traverse the linked list in only one direction. where the next pointer of each node points to other nodes but the next pointer of the last node points to NULL. It is also called "Singly Linked List".
- Doubly Linked List – In this type of linked list, one can move or traverse the linked list in both directions (Forward and Backward)
- Circular Linked List
- Doubly Circular Linked List
- Header Linked List

Basic operations on Linked Lists:

- Deletion
- Insertion
- Search
- Display

Representation of Singly Linked Lists:

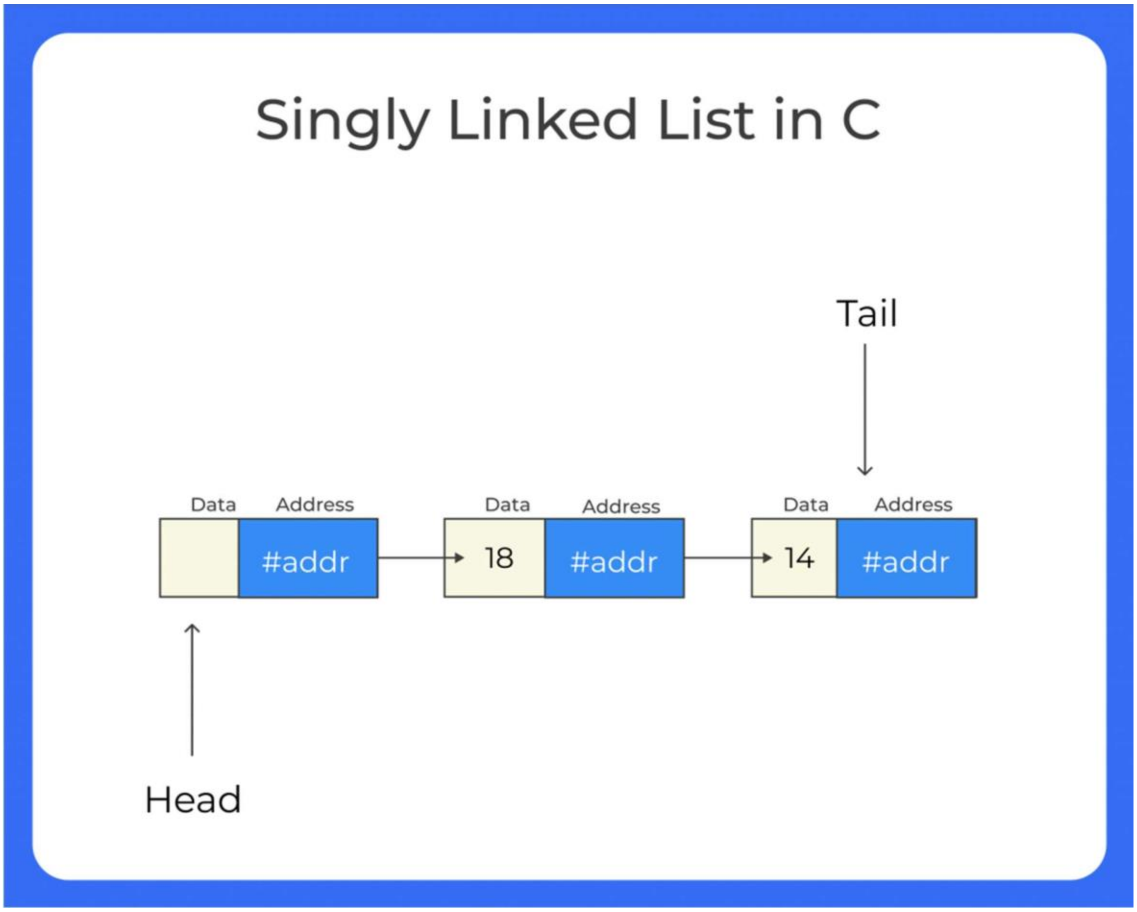
A linked list is represented by a pointer to the first node of the linked list. The first node is called the head of the linked list. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

- A Data Item (we can store integers, strings, or any type of data).
- Pointer (Or Reference) to the next node (connects one node to another) or an address of another node.

SINGLY LINKED LIST:

For constructing a singly linked list in C we make use of the structure keyword (struct), for creating user-defined data types, which can store various different types of data in the nodes of the singly linked list. The other is a pointer type variable, which stores the address of the next node.


Singly Linked List in C
Tail
Data
Address
#addr
Data
Address
18
#addr
Data
Address
14
#addr
Head

PROGRAM:

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
struct Student
{
   int rollnumber;
   char name[100];
   char phone[100];
   float percentage;
   struct Student *next;

}* head;
void insert(int rollnumber, char* name, char* phone, float percentage)
{

   struct Student * student = (struct Student *) malloc(sizeof(struct Student));
   student->rollnumber = rollnumber;
   strcpy(student->name, name);
   strcpy(student->phone, phone);
   student->percentage = percentage;
   student->next = NULL;

   if(head==NULL){
      // if head is NULL
      // set student as the new head
      head = student;
   }
   else{
      // if list is not empty
      // insert student in beginning of head
      student->next = head;
      head = student;
   }

}
void search(int rollnumber)
{
   struct Student * temp = head;
   while(temp!=NULL){
      if(temp->rollnumber==rollnumber){
         printf("Roll Number: %d\n", temp->rollnumber);
         printf("Name: %s\n", temp->name);
         printf("Phone: %s\n", temp->phone);
         printf("Percentage: %0.4f\n", temp->percentage);
         return;
      }
```

```
        temp = temp->next;
    }
    printf("Student with roll number %d is not found !!!\n", rollnumber);
}
void update(int rollnumber)
{

    struct Student * temp = head;
    while(temp!=NULL){

        if(temp->rollnumber==rollnumber){
            printf("Record with roll number %d Found !!!\n", rollnumber);
            printf("Enter new name: ");
            scanf("%s", temp->name);
            printf("Enter new phone number: ");
            scanf("%s", temp->phone);
            printf("Enter new percentage: ");
            scanf("%f",&temp->percentage);
            printf("Updation Successful!!!\n");
            return;
        }
        temp = temp->next;

    }
    printf("Student with roll number %d is not found !!!\n", rollnumber);

}
void Delete(int rollnumber)
{
    struct Student * temp1 = head;
    struct Student * temp2 = head;
    while(temp1!=NULL){

        if(temp1->rollnumber==rollnumber){

            printf("Record with roll number %d Found !!!\n", rollnumber);

            if(temp1==temp2){
                // this condition will run if
                // the record that we need to delete is the first node
                // of the linked list
                head = head->next;
                free(temp1);
            }
            else{
                // temp1 is the node we need to delete
```

```
            // temp2 is the node previous to temp1
            temp2->next = temp1->next;
            free(temp1);
          }

          printf("Record Successfully Deleted !!!\n");
          return;

        }
        temp2 = temp1;
        temp1 = temp1->next;

    }
    printf("Student with roll number %d is not found !!!\n", rollnumber);

}
void display()
{
    struct Student * temp = head;
    while(temp!=NULL){

        printf("Roll Number: %d\n", temp->rollnumber);
        printf("Name: %s\n", temp->name);
        printf("Phone: %s\n", temp->phone);
        printf("Percentage: %0.4f\n\n", temp->percentage);
        temp = temp->next;

    }
}
int main()
{
    head = NULL;
    int choice;
    char name[100];
    char phone[100];
    int rollnumber;
    float percentage;
    printf("1 to insert student details\n2 to search for student details\n3 to delete student
details\n4 to update student details\n5 to display all student details");
    do
    {
        printf("\nEnter Choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
          case 1:
```

```
            printf("Enter roll number: ");
            scanf("%d", &rollnumber);
            printf("Enter name: ");
            scanf("%s", name);
            printf("Enter phone number: ");
            scanf("%s", phone);
            printf("Enter percentage: ");
            scanf("%f", &percentage);
            insert(rollnumber, name, phone, percentage);
            break;
        case 2:
            printf("Enter roll number to search: ");
            scanf("%d", &rollnumber);
            search(rollnumber);
            break;
        case 3:
            printf("Enter roll number to delete: ");
            scanf("%d", &rollnumber);
            Delete(rollnumber);
            break;
        case 4:
            printf("Enter roll number to update: ");
            scanf("%d", &rollnumber);
            update(rollnumber);
            break;
        case 5:
            display();
            break;
      }

  } while (choice != 0);
}
```

OUTPUT:

–________MENU____________
1 – create a SLL of n emp
2 - Display from beginning
3 - Insert at end
4 - delete at end
5 - Insert at beg
6 - delete at beg
7 - exit

Enter choice : 1
Enter no of students : 2
Enter usn,name, branch, sem, phno of student : 007 vijay CSE 3 121 Enter usn,name, branch, sem, phno of student : 100 yashas CSE 3 911

Enter choice : 2
Linked list elements from begining : 100 yashas CSE 3911 007 vijay CSE 3 121
No of students = 2

Enter choice : 3
Enter usn,name, branch, sem, phno of student : 001 raj CSE 3 111

Enter choice : 2
Linked list elements from begining :100 yashas CSE 3 911
007 vijay CSE 3 121
001 raj CSE 3 111
No of students = 3

Enter choice : 4
001 raj CSE 3
111 Enter choice :
2
Linked list elements from begining :
100 yashas CSE 3 911
007 vijay CSE 3 121 No of students = 2

Enter choice : 5
Enter usn,name, branch, sem, phno of student : 003 harsh cse 3 111

Enter choice : 2
Linked list elements from begining
: 003 harsh cse 3 111100 yashas
CSE 3 911
007 vijay CSE 3 121 No of students = 3
Enter choice : 6 003 harsh cse 3 111

Enter choice : 2
Linked list elements from
begining : 100 yashas CSE 3911
007 vijay CSE 3 121 No of
students = 2

Enter choice : 7
Exit

EXPERIMENT NO-08

AIM:

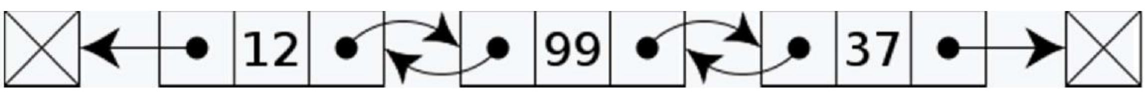
To Design, Develop and implement a menu driven program in C for the following operations on doubly linked list (DLL) of employee data with the fields: SSN, Name, Department, Designation, Sal, Ph.no

THEORY:

In computer science, a doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

In a doubly linked circular linked list, each node has pointers that point to both the next node and the previous node.

The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Traversing the list:

Traversal of a doubly linked list can be in either direction. In fact, the direction of traversal can change many times, if desired. Traversal is often called iteration, but that choice of terminology is unfortunate, for iteration has well-defined semantics (e.g., in mathematics) which are not analogous to traversal.

Following are the basic operations supported by Doubly Linked Lists:

1. Insertion: This operation is used for adding new elements to the list. A new element can be added at the front, at the end of a linked list, or after a given node. The various insertion techniques are discussed below.

2. Deletion: This operation is used to remove elements from the list. The node to be deleted may be the first node, last node, or any node in between the first and last node

3. Traversal: This operation is used to access each element of the list. The Linked List is traversed when printing the contents of the linked list.

Insertion in Doubly Linked Lists:
A Node in a doubly-linked list has a pointer to both the previous and the next node of the linked list. Insertion of a new node in an existing doubly linked list can be done in the following places:

1. Insertion at the Beginning
2. Insertion at the End
3. Insertion after a given Node

Advantages of Doubly Linked Lists:
Following are the main advantages of Doubly Linked Lists:-

1. A DLL can be traversed in both forward and reverse directions
2. A new node can be inserted before a given node quite easily by just changing the pointers.
3. The complete Linked List need not be traversed for deletion operation as in Singly Linked List.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int count=0;
struct node{
struct node* prev;
int Empno;
long int phno;
 float sal;
char name[20],dept[10],desg[20];
struct node* next;
} *h,*temp,*temp1,*temp2,*temp4; void create()
{
 int Empno;
long int phno;
float sal;
char name[20],dept[10],desg[20];
temp=(struct node*)malloc(sizeof(struct node));
temp->prev=NULL;
temp->next=NULL;
printf(" \n enter Empno ,name,department,designation,salary and phno of employee:\n");
scanf("%d%s%s%s%f%ld",&Empno,&name,&dept,&desg,&sal,&phno);

temp->Empno=Empno;
strcpy(temp->name,name);
strcpy(temp->dept,dept);
strcpy(temp->desg,desg);
temp->sal=sal;
temp->phno=phno;
count++;
}
void insertbeg()
{
if(h==NULL)
{
create();
h=temp;
```

```
temp1=h;
}
else {
create();
temp->next=h; h->prev=temp; h=temp;
}
```

```
return;
}
printf("\n linked list elements from beginning :\n");
while (temp2!=NULL)
{
printf("\n %d%s%s%s%f%ld\n", temp2->Empno,temp2->name,
temp2->dept,temp2->desg, temp2->sal,temp2->phno);
temp2=temp2->next;
}
printf("\n number of employees = %d",count);
}
int deleteend()
{
struct node *temp;
temp=h;
if(temp==NULL)
{
printf("\n list is empty \n"); return 0;
}
if(temp->next==NULL)
{
printf("%d%s%s%s%f%ld\n",temp->Empno,temp->name,temp->dept,temp->desg,
temp->sal,temp->phno); free(temp);
h=NULL;
}
else
{
temp=temp1;
temp2=temp1->prev;
temp2->next=NULL;
printf("%d%s%s%s%f%ld\n",temp->Empno,temp->name,temp->dept,temp->desg,
temp->sal,temp->phno); free(temp); temp1=temp2;
}
count--;
return 0;
}
int deletebeg()
{
struct node *temp;
temp=h;
```

```
if(temp==NULL)
{
printf("\n list is empty \n");

}
void insertend()
{
if(h==NULL)
{
create();
h=temp;
temp1=h;
}
Else
{
create();
temp1->next=temp;
temp->prev=temp1;
temp1=temp;
}
}
void displaybeg()
{
temp2=h;
if(temp2==NULL)
{
printf("\n list is empty");

return 0;
}
if(temp->next==NULL)
{
printf("%d%s%s%s%f%ld\n",temp->Empno,temp->name,temp->dept,temp->desg,
temp->sal,temp->phno); free(temp);
h=NULL;
}
```

```
else
{
h=h->next;
h->prev=NULL;
printf("%d%s%s%s%f%ld\n",temp->Empno,temp->name,temp->dept,temp->desg,
temp->sal,temp->phno); free(temp);
}
count--;
return 0;
}
void employerDetails()
{
int ch,n,i;
h=NULL;
temp=temp1=NULL;
printf("_____menu     ");
printf("\n 1.create a DLL of n emp");
printf("\n 2.display from beginning ");
printf("\n 3. insert  at end");
printf("\n 4. delete at end ");
printf("\n 5. insert at beginning ");
printf("\n 6.delete at beginning");
printf("\n 7. to show DLL as queue");
printf("\n 8. exit\n");
printf("          \n");
while(1){
printf("\n enter choice");
scanf("%d",&ch);
switch(ch)
{
case 1: printf(" \n enter no of employees");
scanf("%d",&n);
for(i=0;i<n;i++)
insertend();
break;
case 2: displaybeg(); break;
case 3: insertend(); break;
case 4: deleteend(); break;
```

```
case 5: insertbeg(); break;
case 6: deletebeg(); break;
case 7: exit(0);
default: printf(" wrong choice \n ");
}
}
}
int main()
{
    employerDetails();
    return 0;
}
```

OUTPUT:

```
____menu
 1.create a DLL of n emp
 2.display from beginning
 3. insert  at end
 4. delete at end
 5. insert at beginning
 6.delete at beginning
 7. to show DLL as queue
 8. exit
 enter choice1
 enter no of employees2
 enter Empno ,name,department,designation,salary and phno of employee:
1.Surya is student 150 456
 enter Empno ,name,department,designation,salary and phno of employee:
2.Ram is student 189 762

 enter choice2

 linked list elements from beginning :

 1.Surya is student 150.000000456


 2.Ramisstudent189.000000762

 number of employees = 2
 enter choice
```

EXPERIMENT NO-09

AIM:

To Design, Develop and Implement a menu driven program in c for the following operations on binary search tree (BST) of integer.
a. Create a BST of N integer
b. Traverse the BST in Inorder, Preorder and Postorder
c. Search the BST for the given element (KEY) and report the appropriate message
d. Delete an element from BST
e. Exit

THEORY:

A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

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.

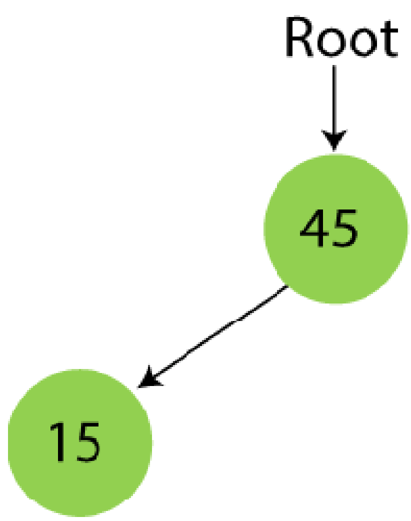
Basic Operations:

1. Insertion in Binary Search Tree
2. Searching in Binary Search Tree
3. Deletion in Binary Search Tree
4. Binary Search Tree (BST) Traversals – Inorder, Preorder, Post Order
5. Convert a normal BST to Balanced BST

Step 1 - Insert 45.

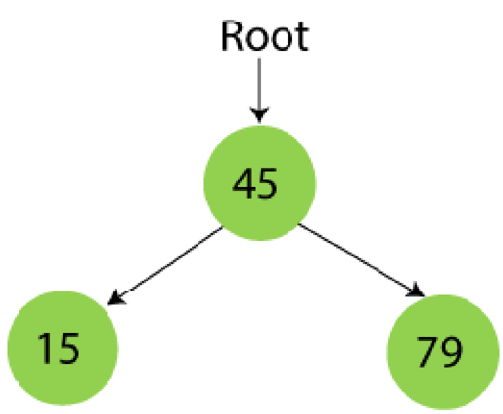
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



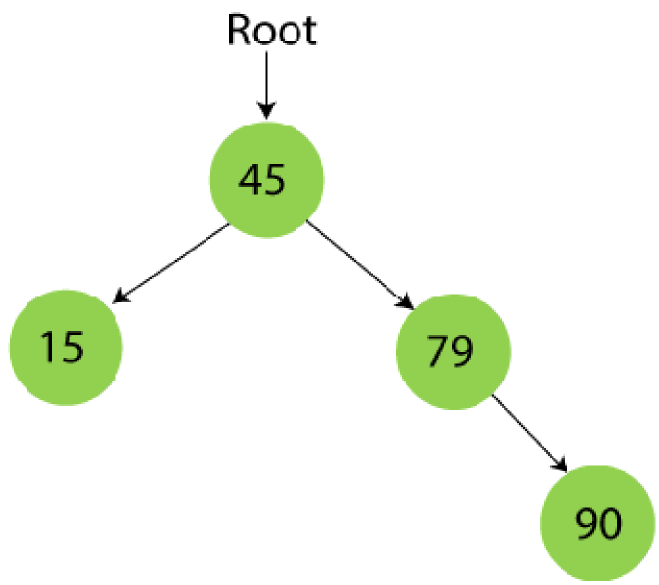
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



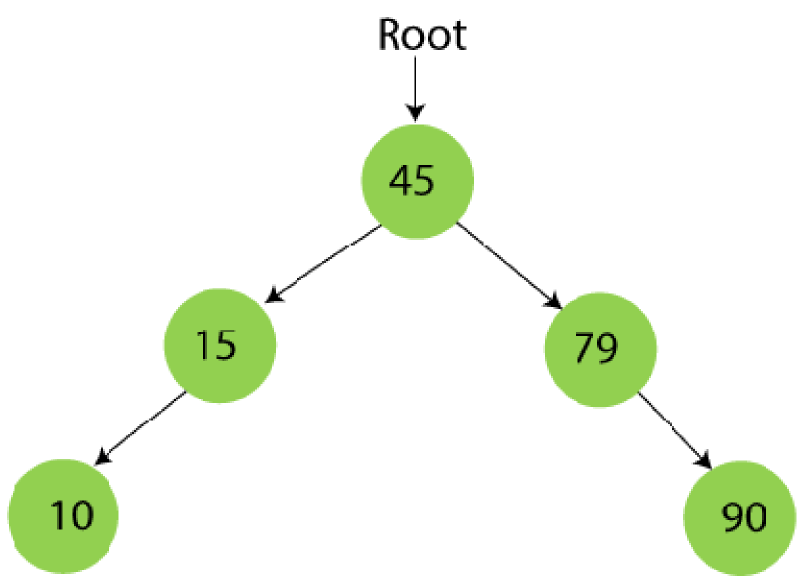
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



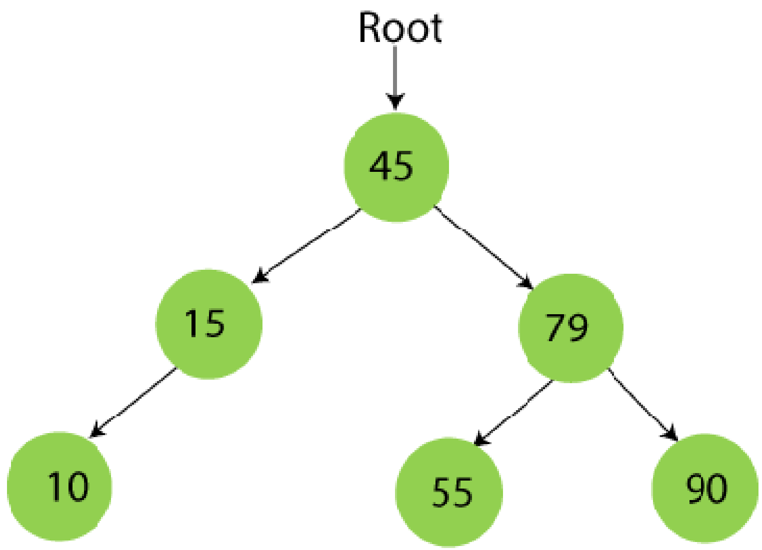
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



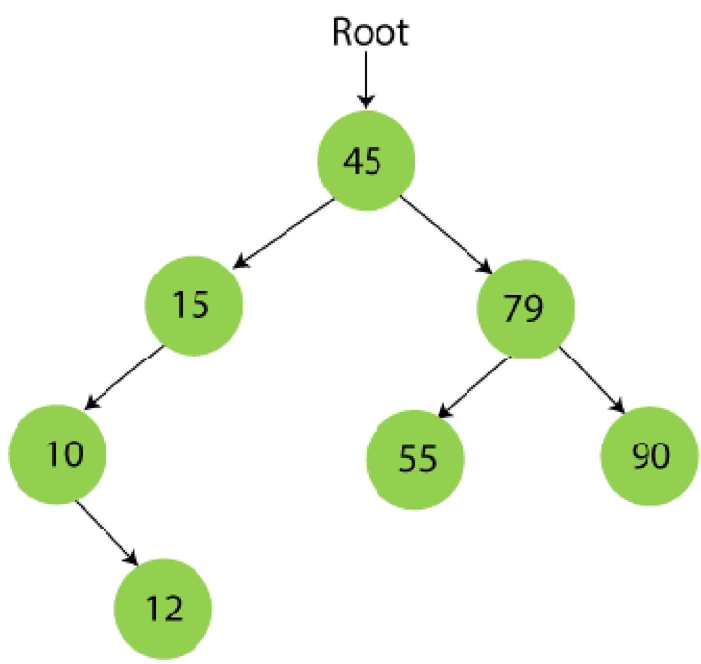
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



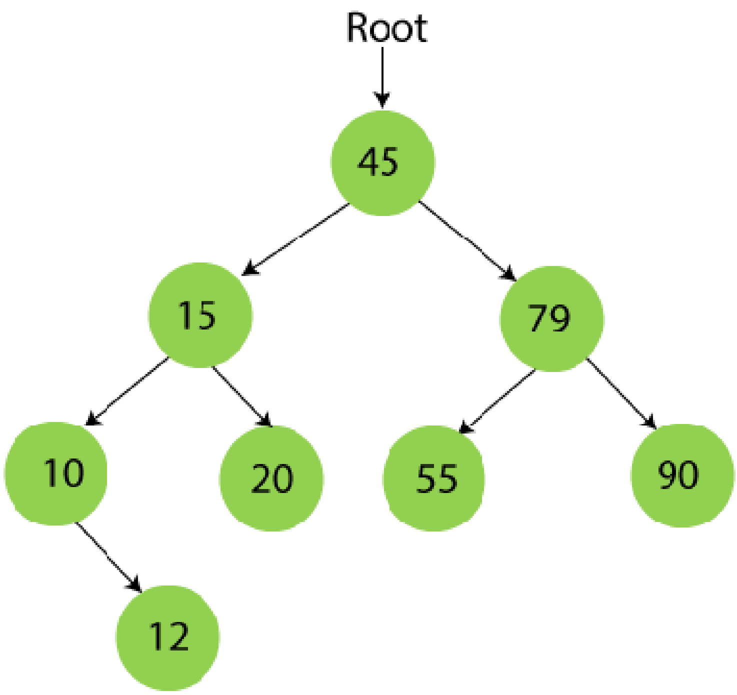
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



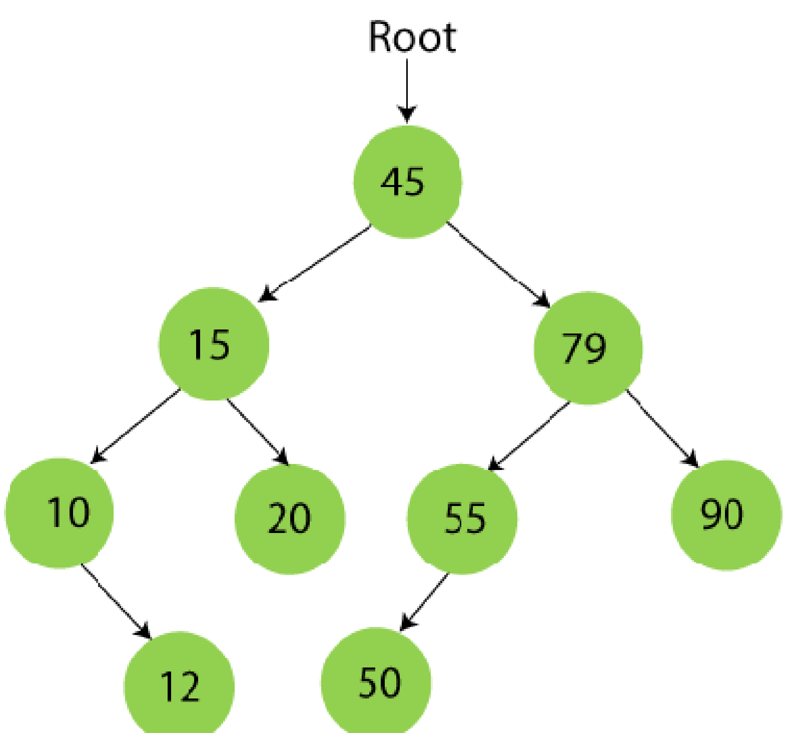
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

    int item;

    struct node*left;

    struct node*right;

};

void inorderTraversal(struct node*root)

{

    if(root==NULL)

        return;

    inorderTraversal(root->left);

    printf("%d->",root->item);

    inorderTraversal(root->right);

}

void preorderTraversal(struct node*root)

{

    if(root==NULL)
```

```
        return;
    printf("%d->",root->item);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
void postorderTraversal(struct node*root)
{
    if(root==NULL)
        return;
    printf("%d->",root->item);
    postorderTraversal(root->left);
    postorderTraversal(root->right);
}
struct node*createNode(value)
{
    struct node*newNode=malloc(sizeof(struct node));
    newNode->item=value;
    newNode->left=NULL;
    newNode->right=NULL;
    return newNode;
};
struct node*insertLeft(struct node*root,int value)
```

```
    {
    root->left=createNode(value);
    return root->left;
};
struct node*insertRight(struct node*root,int value)
{
    root->right=createNode(value);
    return root->right;
};
int main()
{
    struct node*root=createNode(1);
    insertLeft(root,2);
    insertRight(root,3);
    insertRight(root->right,7);
    insertLeft(root->left,4);
    insertLeft(root->right,6);
    insertRight(root->left,5);
    insertLeft(root->left->left,8);
    insertRight(root->left->left,9);
    insertLeft(root->left->right,10);
    insertRight(root->left->right,11);
```

```
    insertLeft(root->right->left,12);
    insertRight(root->right->left,13);


    insertLeft(root->right->right,14);
    insertRight(root->right->right,15);
    printf("Inorder Traversal\n");
    inorderTraversal(root);
    printf("preorder Traversal\n");
    preorderTraversal(root);
    printf("\npostorder Traversal\n");
    postorderTraversal(root);
}
```

OUTPUT:

```
Inorder Traversal
8->4->9->2->10->5->11->1->12->6->13->3->14->7->15->preorder  Traversal
1->2->4->8->9->5->10->11->3->6->12->13->7->14->15->
postorder Traversal
1->2->4->8->9->5->10->11->3->6->12->13->7->14->15->
Process returned 0 (0x0)   execution time : 0.050 s
Press any key to continue.
```

EXPERIMENT NO-10

AIM:

THEORY:
In a Graph, we have a set of nodes and these nodes are connected with each other with the help of some edges. The nodes or vertices are used to store data and this data can be used further.

A graph is a type of non-linear data structure that is used to store data in the form of nodes and edges.

The following is the pictorial representation of a Graph having 5 nodes or vertices:

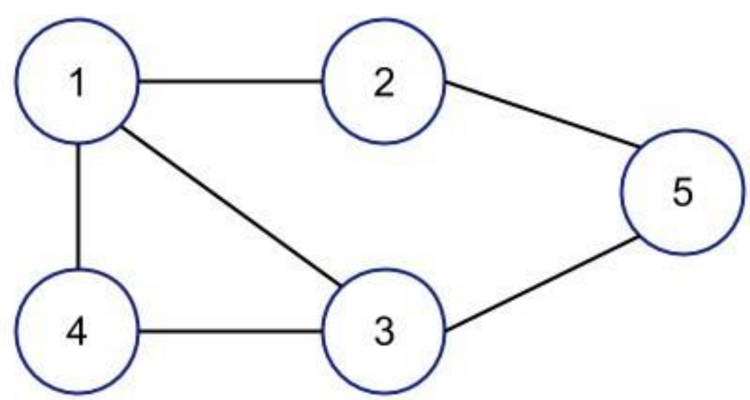


Figure: Graph example

The above is an example of Graph having a set of vertices as V = {1, 2, 3, 4, 5} and the set of edges as E = {1-2, 1-3, 1-4, 2-1, 2-5, 3-1, 3-4, 3-5, 4-1, 4-3, 5-2, 5-3}.

Properties of a Graph:

- Distance between vertices: It is the minimum number of edges present between two nodes. If there are more than one possible paths from one node to another, then the distance between those two vertices is the shortest path between those vertices.

d(A, B); Here, A and B are the nodes and d is the distance between these two nodes

Types of a Graph:

1. Directed and Undirected Graph: It is a very basic type of graph. A graph can be directed or undirected i.e. the edges of a graph can be undirected or directed. In an undirected graph, for every edge, you can traverse in both the direction i.e. if there is an edge between node A and node B , then you can traverse from node A to node B or from node B to node A . But in a directed graph, the direction is given to you and you can traverse in the given direction only.

For example,

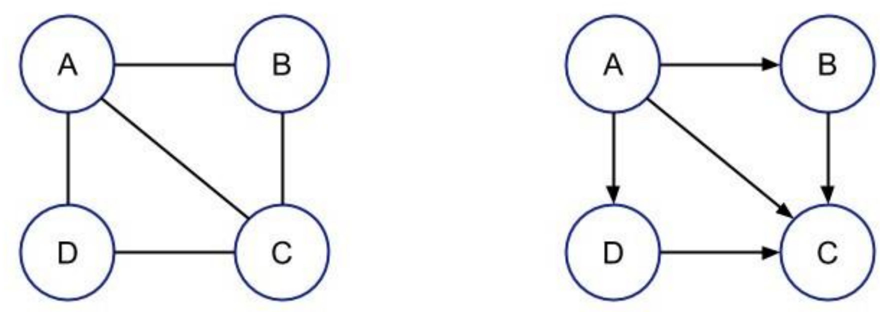


Figure: Undirected Graph **Figure:** Directed Graph

2. Weighted and Unweighted Graph

3. Cyclic and Acyclic Graph

4. Connected and Disconnected Graph

5. Sparse and Dense Graph

Graph Representation:

Adjacency Matrix

Let us assume that the graph is G(n, m) . Here G is the graph, n is the total number of nodes and m is the total number of edges present in the graph G .
We know that the total number of possible edges of an undirected graph is n(n-1)/2 and that of a directed graph is n(n-1). So, the value of " m " should lie between:

1 <= m <= n(n-1)/2 ---- > for undirected graph
1 <= m <= n(n-1) -----> for directed graph

we use an Adjacency matrix which is a 2D matrix of size n*n , where n is the number of nodes present in the graph.

matrix[i][j] = 1; if there is an edge between node i and node j

matrix[i][j] = 0; if there is no edge between node i and node j

PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>

int a[50][50],n,visited[50];

int q[20],front=-1,rear=-1;

int s[20],top=-1,count=0;

void bfs(int v)

{

   int i,cur;

   visited[v]=1;

   q[++rear]=v;

   while(front!=rear)

   {

      cur=q[++front];
```

```
        for(i=1;i<=n;i++)
        {
            if((a[cur][i]==1&&(visited[i]==0)))
{
                q[++rear]=i;
                visited[i]=1;
                printf("%d",i);
            }
        }
    }
}
void dfs(int v)
{
    int i;
    visited[v]=1;
    s[++top]=v;
    for(i=1;i<=n;i++)
    {
        if(a[v][i]==1&&visited[i]==0)
        {
            printf("%d",i);
            dfs(i);
```

```
        }
    }
}


int main()
{
    int ch,start,i,j;
    printf("Enter the number of vertices in graph");
    scanf("%d",&n);
    printf("\nEnter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);
    }
    for(i=1;i<=n;i++)
        visited[i]=0;
    printf("\nEnter the starting vertex:");
    scanf("%d",&start);
    printf("\n==>1.BFS:Print all nodes reachable from a given starting node");
    printf("\n==>2.DFS:Print all nodes reachable from a given starting node");
    printf("\n==3:Exit");
```

```
 printf("\nEnter your choice:");
 scanf("%d",&ch);
 switch(ch)
 {

case 1:
     printf("\nNodes reachable from starting vertex %d are:",start);
     bfs(start);
     for(i=1;i<=n;i++)
     {
         if(visited[i]==0)
             printf("\nThe vertex that is not reachable is %d",i);
     }
     break;
  case 2:printf("\nNodes reachable from starting vertex %d are:\n",start);
 dfs(start);
 break;
 case 3:exit(0);
  default:printf("\nPlease enter valid choice:");
 }
```

OUTPUT:

```
Enter the number of vertices in graph 6
Enter the adjacency matrix:

0 1 0 1 0 0

0 0 1 0 1 0

0 0 0 1 0 0

0 0 0 0 1 0

0 0 0 0 0 1

1 0 1 0 0 0
Enter the starting vertex:7
==>1.BFS:Print all nodes reachable from a given starting node

==>2.DFS:Print all nodes reachable from a given starting node

==3:Exit

Enter your choice:1
Nodes reachable from starting vertex 7 are:

The vertex that is not reachable is 1

The vertex that is not reachable is 2

The vertex that is not reachable is 3

The vertex that is not reachable is 4

The vertex that is not reachable is 5

The vertex that is not reachable is 6

Process returned 0 (0x0)   execution time : 75.618 s

Press any key to continue.
```

EXPERIMENT NO-11

AIM:

To convert the item into a small integer or hash value. This integer is used as an index to store the original data. It stores the data in a hash table. You can use a hash key to locate data quickly.

THEORY:

Hashing is a function used to map data to a fixed-length value. Businesses use hashing in authentication systems and to validate different types of data, such as files and documents. Understanding what hashing is and how it's used is important because it can help to prevent data breaches and protect stored information.

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string.

The most popular use for hashing is the implementation of hash tables. A hash table stores key and value pairs in a list that is accessible through its index. Because key and value pairs are unlimited, the hash function will map the keys to the table size. A hash value then becomes the index for a specific element.

A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. To prevent the conversion of hash back into the original key, a good hash always uses a one-way hashing algorithm.
Hash tables support functions that include the following:

- insert (key, value)
- get (key)
- delete (key)

For example, if you have a list of 20000 numbers, and you have given a number to search in that list- you will scan each number in the list until you find a match.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct set
{
    int key;
    int data;
};
struct set*array;
int capacity=10;
int size=0;
int hashFunction(int key)
{
    return(key%capacity);
}
int checkPrime(int n)
{
    int i;
    if(n==1||n==0)
    {
        return 0;
    }
    for(i=2;i<n/2;i++)
    {
        if(n%i==0)
        {
            return 0;
        }
    }
```

```
    return 1;
}
int getPrime(int n)
{
    if(n%2==0)
    {
        n++;
    }
    while(!checkPrime(n))
    {
        n+=2;
    }
    return n;
}
void init_array()
{
    capacity=getPrime(capacity);
    array=(struct set*)malloc(capacity*sizeof(struct set));
    for(int i=0;i<capacity;i++)
    {
        array[i].key=0;
        array[i].data=0;
    }
}
void insert(int key,int data)
{
    int index=hashFunction(key);
    if(array[index].data==0)
    {
```

```
        array[index].key=key;
        array[index].data=data;
        size++;
        printf("\nKey(%d)has been inserted\n",key);
    }
    else if(array[index].key==key)
    {
        array[index].data=data;
    }
    else
    {
        printf("|nCollision occured \n");
    }
}
void remove_element(int key)
{
    int index=hashFunction(key);
    if(array[index].data==0)
    {
        printf("\nThis key does not exist\n");
    }
    else
    {
        array[index].key=0;
        array[index].key=0;
        size--;
        printf("\nKey(%d)has been removed\n",key);
    }
}
```

```
void display()
{
    int i;
    for(i=0;i<capacity;i++)
    {
        if(array[i].data==0)
        {
            printf("\narray[%d]:/",i);
        }
        else
        {
            printf("\nkey:%d array[%d]:%d\t",array[i].key,i,array[i].data);
        }
    }
}
int size_of_hashtable()
{
    return size;
}
int main()
{
    int choice,key,data,n;
    int c=0;
    init_array();
    do
    {
        printf("1.insert item in the hash table\n2.Remove item from the Hash Table\n3.check
the size of hash table\n4.Display the Hash table\n\nPlease enter your choice:");
```

```
        scanf("%d",&choice);
        switch(choice)
        {
        case 1:
            printf("Enter key -:\t");
            scanf("%d",&key);
            printf("Enter data -:\t");
            scanf("%d",&data);
            insert(key,data);
            break;
        case 2:
            printf("Enter the key to delete -:");
            scanf("%d",&key);
            remove_element(key);
            break;
        case 3:
            n=size_of_hashtable();
            printf("Size of Hash Table is -:%d\n",n);
            break;
        case 4:
            display();
            break;
            default:printf("Invalid input\n");
        }
        printf("\nDo you want to continue(press 1 for yes):");
        scanf("%d",&c);
    }while(c==1);
}
```

OUTPUT:

1. insert item in the hash table
2. Remove item from the Hash Table
3. check the size of hash table
4. Display the Hash table

Please enter your choice:1

Enter key -: 1

Enter data -: 2

Key(1)has been inserted

Do you want to continue(press 1 for yes):1

1. insert item in the hash table
2. Remove item from the Hash Table
3. check the size of hash table
4. Display the Hash table

Please enter your choice:1

Enter key -: 2

Enter data -: 4

Key(2)has been inserted

Do you want to continue(press 1 for yes):1

1. insert item in the hash table
2. Remove item from the Hash Table
3. check the size of hash table
4. Display the Hash table

Please enter your choice:3

Size of Hash Table is -:2

Do you want to continue(press 1 for yes):1

1. insert item in the hash table
2. Remove item from the Hash Table
3. check the size of hash table
4. Display the Hash table

Please enter your choice:2

Enter the key to delete -:2

Key(2)has been removed

Do you want to continue(press 1 for yes):1

1. insert item in the hash table
2. Remove item from the Hash Table
3. check the size of hash table
4. Display the Hash table

Please enter your choice:4

array[0]:/

key:1 array[1]:2
key:0 array[2]:4
array[3]:/
array[4]:/
array[5]:/
array[6]:/
array[7]:/
array[8]:/

array[9]:/

array[10]:/

Do you want to continue(press 1 for yes):1

1. insert item in the hash table
2. Remove item from the Hash Table
3. check the size of hash table
4. Display the Hash table

Please enter your choice:4

array[0]:/

key:1 array[1]:2

key:0 array[2]:4

array[3]:/

array[4]:/

array[5]:/

array[6]:/

array[7]:/

array[8]:/

array[9]:/

array[10]:/

Do you want to continue(press 1 for yes):