Experiment No: 01

Experiment Name: Write a prolog program that contains a number of predicates that describe a family's genealogical relationships.

Objectives: In this program we will learn how to find out the proper answer for a query using Prolog programming.

Theory: At first, we will define some facts regarding family members. Then we will form some rules using these facts. Then we will enter query to check if some goals are satisfied or not.

Algorithm:

Step 1: Start.

Step 2: Write down some facts regarding family members.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax: The following facts and rules is defined in the program.

Facts:

1. Amy is a female.
   Prolog fact representation: female(amy).
2. Johnette is a female.
   Prolog fact representation: female(johnette).
3. Anthony is a male.
   Prolog fact representation: male(anthony).
4. Bruce is a male.
   Prolog fact representation: male(bruce).
5. Ogden is a male.
   Prolog fact representation: male(ogden).
6. Amy is a parent of Johnette.
   Prolog fact representation: parentof(amy,johnette).
7. Amy is a parent of Anthony.
   Prolog fact representation: parentof(amy,anthony).
8. Amy is a parent of Bruce.
   Prolog fact representation: parentof(amy,anthony).
9. Ogden is a parent of Johnette.
   Prolog fact representation: parentof(ogden,johnette).
10. Ogden is a parent of Anthony.
    Prolog fact representation: parentof(ogden,anthony).

11. Ogden is a parent of Bruce.
Prolog fact representation: parentof(ogden,bruce).

Rules:

1. X and Y are sibling if Z is parent of X and Y.
   Prolog rules representation: siblingof(X,Y) :- parentof(Z,X), parentof(Z,Y).
2. X is brother of Y if Z is parent of X and Y and X is male.
   Prolog rules representation: brotherof(X,Y) :-
   parentof(Z,X), male(X),
   parentof(Z,Y).

Code:

```
% fact part is written here
    female(amy).
    female(johnette).
    male(anthony).
    male(bruce).
    male(ogden).
    parentof(amy,johnette).
    parentof(amy,anthony).
    parentof(amy,bruce).
    parentof(ogden,johnette).
    parentof(ogden,anthony).
    parentof(ogden,bruce).
%Rules part are written here
    siblingof(X,Y) :-
        parentof(Z,X), parentof(Z,Y).
    brotherof(X,Y) :-
        parentof(Z,X), male(X), parentof(Z,Y).
```

Query & Output:

```
% c:/users/user/documents/prolog/zahir compiled 0.00 sec, -2 clauses
?- siblingof(amy,Y).
false.

?- siblingof(anthony,Y).
Y = johnette ;
Y = anthony ;
Y = bruce ;
Y = johnette ;
Y = anthony ;
Y = bruce.
```

Conclusion: We have successfully described a family's gene logical relationship in this experiment.

Experiment No:02

Experiment Name: Write a prolog program to print a list of items.

Objectives: In this program we will learn how to print a list of items using prolog programming.

Theory: Here, we will write down some items as a fact and after that we will print these items by executing the query.

Algorithm:

Step 1: Start.

Step 2: Write down a list of items using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts:

1. Define item.
   Prolog fact representation: item(x).

Rules:

1. write the item names. Print them one after another in a new line also it will immediately fail when prolog encounters it as a goal.
   Prolog rules representation: products: - item(X), write(X), nl, fail.

Code:

```
% Print a list of items

item(pen).

item(laptop).

item(bag).

products :-

item(X), write(X), nl, fail.
```

Query & Output:

```
% c:/users/user/documents/prolog/t compiled 0.00 sec, 0 clauses
?- products.
pen
laptop
bag
false.
```

Conclusion: We have successfully printed a list of item in this experiment.

Experiment No:03

Experiment Name: Write a prolog program to Print a list of numbers.

Objectives: In this program we will learn how to print a list of numbers using prolog programming.

Theory: Here, we will write down some numbers as a fact and after that we will print these numbers by executing the query.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts:

1. Define number.
   Prolog fact representation: number(x).

Rules:

1. write the numbers. Print them one after another in a new line also it will immediately fail when prolog encounters it as a goal.
   Prolog rules representation: array :- num(X), write(X), nl, fail.

Code:

```
% Print a list of numbers

num(100).

num(200).

num(300).

num(400).

num(500).
```

array :-

num(X), write(X), nl, fail.

Query & Output:

```
% c:/users/user/documents/prolog/t compiled 0.00 sec, 0 clauses
?- array.
100
200
300
400
500
false.

?-
```

Conclusion: We have successfully printed a list of numbers in this experiment.

Experiment No:04

Experiment Name: Write a prolog program to Print the GPA of a given student.

Objectives: In this program we will learn how to print GPA of a given student using prolog programming.

Theory: At first, we will define GPA of some students with their name and GPA. Then we will find GPA of a student by entering his/her name and print the GPA.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts: X = name of the student, Y= GPA of the student.

Prolog fact representation: gpa(X,Y).

Rules: Read student name from user and write the GPA.

Prolog rules representation: read(X),gpa(X,Y),nl, write(Y).

Code:

```
% Print the GPA of a given student.

gpa(zahir, 3.5).

gpa(arfan, 3.2).

gpa(eti, 3.7).

gpa(ikiryo, 3.3).

gpa(nobita, 3.4).

result :-

write("Enter student name:"),

read(X), gpa(X,Y),nl,
```

write("GPA is "),

write(Y).

Query & Output:

```
?-
% c:/users/user/documents/prolog/z4 compiled 0.00 sec, 0 clauses
?- result.
Enter student name:zahir.

GPA is 3.5
true.

?-
```

Conclusion: We have successfully printed a students name with his/her GPA in this experiment.

Experiment No:05

Experiment Name: Write a prolog program to Print the student-professor relation.

Objectives: In this program we will learn how to print student-professor relation using prolog programming.

Theory: At first, we will define some students with their name & the course that he/she studies. we will define some professor with their name & the course that he/she teaches. Then we will find out, professor & student have a relation if they belong to the same course.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts: 1. X = professor name, Y = course teaches.

Prolog fact representation: teaches(X,Y).

2. Y = student name, C = course studies.

Prolog fact representation: studies(Y,C).

Rules: 1. course studies by the student.

Prolog rule representation: studies(Y,C).

2. course teaches by the professor.

Prolog rule representation: studies(X,Y).

Code:

```
%student-professor relation

studies(arfan, csc135).

studies(zahir, csc135).

studies(eti, csc131).

studies(sokal, csc131).
```

teaches(ikiryo, csc135).

teaches(nobita, csc131).

teaches(peter, csc171).

teaches(oporagita, csc134).

professor(X, Y) :-

teaches(X, C), studies(Y, C).

Query & Output:

```
% c:/users/user/documents/prolog/l6 compiled 0.00 sec, 0 clauses
?- studies(arfan,W).
W = csc135.

?- professor(nobita,S).
S = eti ;
S = sokal.

?-
```

Conclusion: We have successfully executed student-professor relation in this experiment.

Experiment No:06

Experiment Name: Write a prolog program to Print the Nth Fibonacci number.

Objectives: Objective of this program is to learn how to find Nth number of Fibonacci series in prolog programming.

Theory: The Fibonacci numbers, commonly denoted Fn, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is, $F0 = 0$, $F1 = 1$, And $Fn = Fn\text{-}1 + Fn\text{-}2$ $For\ n > 1$.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts:1. 1st number of fibonacci series is defined as 1.

Prolog fact representation: fib(1,1).

2. 2nd number of fibonacci series is defined as 1.

Prolog fact representation: fib(2,1).

3. Nth number (where, N>=3) of fibonacci series is defined as

N1 is N-1,N2 is N-2

Prolog fact representation: fib(N,R):-N>=3 N1 is N-1,N2 is N-2.

Rules: Nth number of Fibonacci series.

Prolog rule representation: fib(N,R):- N1=N-1, N2=N-2, fib(N1,R1), fib(N2,R2), R=R1+R2.

Code:

```
% Factorial of a number.

fact(0, 1).

fact(N, F) :-
```

N > 0,

N1 is N-1,

fact(N1, F1),

F is N*F1.

Query & Output:

```
% c:/users/user/documents/prolog/ba compiled 0.00 sec, 0 clauses
?- fact(5,F).
F = 120 ;
false.

?-
```

Conclusion: We have successfully printed Nth number of the fibonacci series in this experiment.

Experiment No:07

Experiment Name: Write a prolog program to Print the Factorial of a number.

Objectives: To know to calculate factorial of a number using prolog programming.

Theory: The factorial, symbolized by an exclamation mark (!), is a quantity defined for all integer s greater than or equal to 0. For an integer n greater than or equal to 1, the factorial is the product of all integers less than or equal to n but greater than or equal to 1. The factorial value of 0 is defined as equal to 1. The factorial values for negative integers are not defined. Mathematically, the formula for the factorial is as follows. If n is an integer greater than or equal to 1, then n ! = n (n - 1)(n - 2)(n - 3) ... (3)(2)(1) If p = 0, then p ! = 1 by convention. The factorial is of interest to number theorists. It often arises in probability calculations, particularly those involving combinations and permutations. We have to write a prolog program that will find the factorial of a number.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts: 1. If N=0, then R=1. i.e., 0! = 1.

Prolog fact representation: fact(0,1).

Rules: If N≠ 0, $N! = R$:- call fact(N1,R1), N=N1+1, R=R1*N.

Prolog rule representation: fact(N,R):-fact(N1,R1),N is N1+1,R is R1*N.

Code:

```
fact(0,1).

fact(N,R):-fact(N1,R1),N is N1+1,R is R1*N.
```

Query & Output:

```
% c:/users/user/documents/prolog/a compiled 0.00 sec, 0 clauses
% c:/users/user/documents/prolog/a compiled 0.00 sec, 0 clauses
?- fact(6,R).
R = 720 ;
```

Conclusion: We have successfully printed the factorial of a number in this experiment.

Experiment No:08

Experiment Name: Write a prolog program to Read a number and write the cube of the number.

Objectives: In this program is to calculate the cube of a given number in prolog programming.

Theory: We will read input from keyboard and write the output. We will demonstrate the cube of a number, $n3 = n * n * n$. Here, read(X): is used for reading terms from current input stream. Write(X): is used for outputting term X on the current output file.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Rules: Read X from keyboard and calculate cube and write the output:

cube:-read(X), process (X).

process(N):-C=N*N*N, write (C),cube.

Code:

```
cube:- write('Enter a number: '),

read(X), process(X).

process(stop):-!.

process(N):- C is N*N*N,

write('Cube of '),write(N),

write(' is '),write(C),nl,cube.
```

Query & Output:

```
?-
% c:/users/user/documents/prolog/qq compiled 0.00 sec, -2 clauses
?- cube.
Enter a number: 2.
Cube of 2 is 8
Enter a number: |:
```

Conclusion: We have successfully executed the cube program**.**

Experiment No:09

Experiment Name: Write a prolog program for Computing Greatest Common Divisor (GCD).

Objectives: To learn to find the GCD of two numbers in prolog programming.

Theory: GCD stands for Greatest Common Divisor. GCD is largest number that divides the given numbers.

GCD Example: Find the GCD of 45 and 54.

Step 1: Find the divisors of given numbers:

The divisors of 45 are: 1, 3, 5, 9, 15, 45

The divisors of 54 are: 1, 2, 3, 6, 9, 18, 27, 54

Step 2: Find the greatest number that these two lists share in common. In this example the GCD is 9.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

gcd(X,Y,D):- Used to find gcd of X,Y & keep it in D,

Code:

```
gcd(X,0,X).

gcd(X,Y,D):- R is X mod Y, gcd(Y,R,D).
```

Query & Output:

```
?-
% c:/users/user/documents/prolog/cod8 compiled 0.00 sec, -1 clauses
?- gcd(5,6,K).
K = 1
```

Conclusion: We have successfully completed the GCD program using Prolog programming.

Experiment No:10

Experiment Name: Write a prolog program to satisfy a list of goals.

Objectives: To learn how facts and rules are works.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts:

1. big bear.
   Prolog fact representation: big(bear).
2. gig elephant.
   Prolog fact representation: big(elephant).
3. small cat.
   Prolog fact representation: small(cat).
4. grown bear.
   Prolog fact representation: brown(bear).
5. black cat.
   Prolog fact representation: black(cat).
6. gray elephant.
   Prolog fact representation: gray(elephant).

Rules: Anything black is dark.

Prolog fact representation: dark(Z) :- black(Z).

Code:

```
big(bear). %Clause 1

big(elephant). %Clause 2

small(cat). %Clause 3
```

brown(bear). %Clause 4

black(cat). %Clause 5

gray(elephant). %Clause 6

dark(Z) :- black(Z). %Clause 7: Anything black is dark

dark(Z) :- brown(Z). %Clause 8: Anything brown is dark

Query & Output:

```
?- big(X).
X = bear ;
X = elephant.

?-
```

Conclusion: Successfully executed the program.

Experiment No:11

Experiment Name: Write a prolog program to find the length of list.

Objectives: To calculate the length of list in prolog programming.

Theory: This is used to find the length of list L. We will define one predicate to do this task. Suppose the predicate name is list length(L,N). This takes L and N as input argument. This will count the elements in a list L and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases –

- If list is empty, then length is 0.
- If the list is not empty, then L = [Head|Tail], then its length is 1 + length of Tail.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

list_length(): Used to count the number of elements in a list & shows as output.

Code:

```
list_length([],0).

list_length([_|TAIL],N) :- list_length(TAIL,N1), N is N1 + 1.
```

Query & Output:

```
?-
% c:/users/user/documents/prolog/vk compiled 0.02 sec, -2 clauses
?- list_length([],0).
true.

?- list_length([a,b,c,d],R).
R = 4.

?-
```

Conclusion: we have successfully calculated the length of the list.

Experiment No:12

Experiment Name: Write a prolog program to concatenation the two lists.

Objectives: To know concatenation of list in prolog programming.

Theory: This is used to find the length of list L. We will define one predicate to do this task. Suppose the predicate name is list_length(L,N). This takes L and N as input argument. This will count the elements in a list L and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases –

1. If list is empty, then length is 0.
2. If the list is not empty, then L = [Head|Tail], then its length is 1 + length of Tail.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Rules:

1) Append operation between an empty list and a list L will return the L.

append([],L,L).

2) Append operation of two non-empty list:

append([H|T],L,[H|R]):-append(T,L,R).

Code:

```
append([],L,L).

append([H|T],L,[H|R]):-append(T,L,R).
```

Query & Output:

```
% c:/users/user/documents/prolog/vk compiled 0.00 sec, -1 clauses
?- append([1],[2,3],R).
R = [1, 2, 3].

?-
```

Conclusion: we have successfully concatenation the two lists

Experiment No:13

Experiment Name: Write a prolog program to delete an element from the list.

Objectives: To learn how to perform delete operations of a list in prolog programming.

Theory: The list is a simple data structure that is widely used in non-numeric programming. List consists of any number of items, for example, red, green, blue, white, dark. It will be represented as, [red, green, blue, white, dark]. The list of elements will be enclosed with square brackets.

A list can be either empty or non-empty. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list consists of two things as given below-

The first item, called the head of the list;

The remaining part of the list, called the tail.

Concatenation of two lists means adding the list items of the second list after the first one. So if two lists are [a,b,c] and [1,2], then the final list will be [a,b,c,1,2]. So to do this task we will create one predicate called list_concat(), that will take first list L1, second list L2, and the L3 as resultant list.

Suppose we have a list L and an element X, we have to delete X from L. So there are three cases –

- If X is the only element, then after deleting it, it will return empty list.
- If X is head of L, the resultant list will be the Tail part.
- If X is present in the Tail part, then delete from there recursively.

Appending two lists means adding two lists together, or adding one list as an item. Now if the item is present in the list, then the append function will not work. So we will create one predicate namely, list_append(L1, L2, L3).

We have to write a prolog program that will perform lists concatenation, append & deletion operation.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

1. list_concat([],[],Variable): Used to concatenate first list with second list & keep it in the variable,

2. list_append(var,[],Variable): Used to append var in the list & keep it in the variable,

3. list_delete(var,[],variable): Used to delete var from the list & keep it in the variable.

Code:

```
list_concat([],L,L).

list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).

list_delete(X, [X], []).

list_delete(X,[X|L1], L1).

list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X,L2,L1).

list_member(X,[X|_]).

list_member(X,[_|TAIL]) :- list_member(X,TAIL).

list_append(A,T,T) :- list_member(A,T),!.

list_append(A,T,[A|T]).
```

Query & Output:

```
1 ?- list_concat([a,b,c,d],[1,2,3],R).
R = [a, b, c, d, 1, 2, 3].

2 ?- list_append(e,[a,b,c,d],R).
R = [e, a, b, c, d].

3 ?- list_delete(e,[a,b,c,d,e],R).
R = [a, b, c, d] ,
```

Conclusion: We have successfully executed the program.

Experiment No:14

Experiment Name: Write a prolog program to insert an item into list.

Objectives: Objective of this program is to demonstrate add operation of a list in prolog programming.

Theory: Add-We will give an element in a list. The element will be added in the list.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts: Add X to the list L: add(X,L,[X|L]).

Code:

```
list_insert(X,L,R):- list_delete(X,R,L).
```

Query & Output:

```
?-
% c:/users/rafs/documents/prolog/16 compiled 0.00 sec, -2 clauses
% c:/users/rafs/documents/prolog/16 compiled 0.00 sec, 0 clauses
?- list_insert(e,[a,i,o,u],NewList).
NewList = [e, a, i, o, u]
```

Conclusion: We have successfully executed the add operation in prolog programming.

Experiment No:15

Experiment Name: Write a prolog program to union of two sets.

Objectives: Learn to implement union operation in prolog.

Theory: In set theory, the union (denoted by ∪) of a collection of sets is the set of all elements in the collection.

$A \cup B = \{x: x \in A \text{ or } x \in B\}$.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

1. The result of union between an empty list and X is X: union([],X,X):-!.
2. Union between a list and Y(which is a member of the list): union([X|R],Y,Z):-member(X,Y),union(R,Y,Z),!.
3. Union between a list and Y(which is not a member of the list): union([X|R],Y,[X|Z]):-union(R,Y,Z).

Code:

```
union([],X,X):-!.

union([X|R],Y,Z):- member(X,Y),union(R,Y,Z),!.

union([X|R],Y,[X|Z]):- union(R,Y,Z).
```

Query & Output:

```
% c:/users/rafs/documents/prolog/17 compiled 0.00 sec, 0 clauses
?- union([a,b,c],[c,d,e],R).
R = [a, b, c, d, e].
```

Conclusion: We have successfully executed the program.

Experiment No:16

Experiment Name: Write a prolog program to intersection of two sets.

Objectives: Learn to implement intersection operation in prolog.

Theory: The intersection operation is denoted by the symbol ∩. The set A ∩ B—read "A intersection B" or "the intersection of A and B"—is defined as the set composed of all elements that belong to both A and B.
$A \cap B = \{x: x \in A \; and \; x \in B\}$.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

1. Intersection between a list and X returns an empty list. intersect([],X,[]):-!.
2. Intersection between a list and Y(which is not a member of the list). intersect([X|R],Y,[X|T]):- member(X,Y),intersect(R,Y,T),!. Intersection between a list and Y(which is a member of the list).
3. intersect([X|R],Y,L):- intersect(R,Y,L).

Code:

```
intersect([],
X, []) :- !.

intersect([X|R], Y, [X|T]) :-
member(X, Y), intersect (R, Y,
T).

intersect([X|R], Y, Z) :-
intersect (R, Y, Z).
```

Query & Output:

```
% c:/users/rafs/documents/prolog/18 compiled 0.00 sec, 0 clauses
?- intersect([a,b,c],[a,d,e],R).
R = [a]
```

Conclusion: We have successfully implemented intersection operation in this experiment.

Experiment No:17

Experiment Name: Write a prolog program to do bubble sort program.

Objectives: Learn to implement bubble sort in prolog.

Theory: Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

We have to write a prolog program that will perform Bubble Sorting.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

1.gt(X,Y): Used to define X is greater than Y,

2. swap(): Used to swap values,

3. bubblesort(List, result): Used to sort the list & keep it in result.

Code:

```
gt(X,Y) :- X > Y.

bubblesort(List, Sorted) :-

swap(List, List1), !, % A useful swap in List?

bubblesort(List1, Sorted).

bubblesort(Sorted, Sorted). % list is already sorted

swap([X,Y|Rest], [Y,X|Rest]) :- % Swap first two

gt(X,Y).

swap([Z|Rest], [Z|Rest1]) :- % Swap elements in tail

swap(Rest, Rest1).
```

Query & Output:

```
1 ?- bubblesort([3,8,1,2,4],R).
R = [1, 2, 3, 4, 8].

2 ?- bubblesort([9,3,1,7,5],R).
R = [1, 3, 5, 7, 9].
```

Conclusion: We have successfully run this program.

Experiment No:18

Experiment Name: Write a prolog program to demonstrate looping until a condition is satisfied.

Objectives: To learn about looping until a condition in prolog programming.

Theory: This experiment shows the use of recursion to read terms entered by the user from the keyboard and output them to the screen, until a word end is encountered, using 'disjunctive goal'(word=end).

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts:

Rules: Process until "end" is entered:

test:-write('Output stream'), read(word), write(word), (word=end; test).

Code:

test :- write('Type the word : '), read(word), write('Input was: '),

write(word), nl, (word=end; test).

Query & Output:

```
?-
% c:/users/rafs/documents/prolog/20 compiled 0.00 sec, 0 clauses
% c:/users/rafs/documents/prolog/20 compiled 0.00 sec, 0 clauses
?- test.
Type the word : good.
Input was: good
Type the word : |:
```

Conclusion: We have successfully implemented looping until a condition is satisfied in this experiment.

Experiment No:19

Experiment Name: Write a prolog program to demonstrate looping of a fixed number of times.

Objectives: Learn about looping a fixed number of times in prolog programming.

Theory: The testloop predicate is defined as 'loop from N, write the value of N, then subtract one to give to M, then loop from M'. 'And by the first clause, is defined as 'if the argument is zero, do nothing(stop!) '.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

If the argument is zero, do nothing: stop! testloop(0).

Loop from N, write the value of N, then subtract one to give to M, then loop from M: testloop(N):-N>0, write(N), M is N-1, testloop(M).

Code: testloop(0). testloop(N):-N>0,write('Number:'), write(N),
nl,M is N-1,testloop(M).

Query & Output:

```
?-
% c:/users/rafs/documents/prolog/21 compiled 0.00 sec, -2 clauses
?- testloop(4).
Number : 4
Number : 3
Number : 2
Number : 1
true
```

Conclusion: We have successfully implemented looping of a fixed number of time in this experiment.

Experiment No:20

Experiment Name: Write a prolog program to print 1-n by using looping.

Objectives: Learn about to print 1-n of using looping in prolog programming.

Theory: The testloop predicate is defined as 'loop from N, write the value of N, then subtract one to give to M, then loop from M'. 'And by the first clause, is defined as 'if the argument is zero, do nothing(stop!) '.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

If the argument is zero, do nothing: stop! testloop(0).

Loop from N, write the value of N, then subtract one to give to M, then loop from M:

testloop(N):-N>0, write(N), M is N-1, testloop(M).

Code:

start:- write('N= ?'), read(N), loop(0,N). loop(N0, N):- N1
is N0+1, N1=< 10, write(N1), loop(N1, N).

Query & Output:

```
?-
% c:/users/rafs/documents/prolog/22 compiled 0.00 sec, -2 clauses
?- start.
N= ?8.
12345678
false.

?-
```

Conclusion: We have successfully implemented to print 1-n by using looping.

Experiment No:21

Experiment Name: Write a prolog program to print n-1 of using looping.

Objectives: Learn about to print n-1 of using looping in prolog programming.

Theory: It is the reverse of print n-1 using looping. The testloop predicate is defined as 'loop from N, write the value of N, then subtract one to give to M, then loop from M'. 'And by the first clause, is defined as 'if the argument is zero, do nothing(stop!)'.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Rules:

If the argument is zero, do nothing: stop! testloop(0).

Loop from N, write the value of N, then subtract one to give to M, then loop from M: testloop(N):-N>0, write(N), M is N-1, testloop(M).

Code:

```
start:- write('N= ?'), read(N), loop(0,N). loop(N0,
N):- N1 is N0+1, N1=< 10, write(N1), loop(N1, N).
```

Query & Output:

```
% c:/users/rafs/documents/prolog/23 compiled 0.00 sec, -1 clauses
?- start.
N= ?8.
87654321
false.
```

Conclusion: We have successfully implemented to print n-1 using looping.

Experiment No:22

Experiment Name: Write a menu-based prolog program.

Objectives: Learn about menu-base prolog program in prolog programming.

Theory: When I started learning Prolog, I thought that the interactive console was great for trying things out and running queries real time, but I noticed an overall lack of instruction out there on writing a menu driven program that will take input. Because of that, I created a quick little program that will demonstrate a simple menu and some I/O. In menu-base program there exist different types of looping system.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

If the argument is zero, do nothing: stop! testloop(0).

Loop from N, write the value of N, then subtract one to give to M, then loop from M:

testloop(N):- N>0, write(N), M is N-1, testloop(M).

Code:

```
start:- nl,

write('**MENU**'),nl,
write('1. Personal Info'),nl,
write('2. Family Info'),nl,
write('3. Academic Info'),nl,
write('0. Exit'),nl,nl,
write('Choice = ? '),read(X),

( X = 1, write('Personal
Info:'),nl, write('Name:
'),read(Name), write('Mobile:
'),read(Mob), write('Email:
'),read(Em), write('Name:
'),write(Name),nl,
```

```
write('Mobile:
'),write(Mob),nl, write('Email:
'),write(Em),nl,

start;

X = 2,


write('Family Info:'),nl, write('Father
name: '),read(Fnam), write('Mother
name: '),read(Mnam), write('Home
Address: '),read(Home), write('Father
name: '),write(Fnam),nl, write('Mother
name: '),write(Mnam),nl,

write('Home Address:
'),write(Home),nl, start; X = 3,
write('Academic Info:'),nl, write('Dept.:
'),read(Dept), write('Roll No.:
'),read(Roll), write('CGPA:
'),read(Cgpa), write('Dept.:
'),write(Dept),nl, write('Roll No.:
'),write(Roll),nl,

write('CGPA:
'),write(Cgpa),nl, start; X = 0,
write('Thanks')).
```

Query & Output:

```
% c:/users/rafs/documents/prolog/new folder/example_24 compiled 0.
00 sec, 0 clauses
?- start.

**MENU**
1. Personal Info
2. Family Info
3. Academic Info
0. Exit

Choice = ? 2.
Family Info:
Father name: |: milon.
Mother name: |: jui.
Home Address: |: dinajpur.
Father name: milon
Mother name: jui
Home Address: dinajpur
```

Conclusion: We have successfully implemented menu-based prolog program in this experiment.

Experiment No:23

Experiment Name: Write a prolog program that checks whether there exists a route between two nodes.

Objectives: To learn how to check whether there exists a route between two nodes using prolog.

Theory: To traverse any path and node tree system can maintain the looping system. The testloop predicate is defined as 'loop from N, write the value of N, then subtract one to give to M, then loop from M'. 'And by the first clause, is defined as 'if the argument is zero, do nothing(stop!) '. When this loop does not contain any node then the tree traverse is end.

Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

If we route any type of tree then we should follow this:

Define Route:

• Route from node A to B is defined as follows:

– Route from A to B, if there is an edge from A some node C and a route from C to B.
– Route from A to B, if there is a direct edge from A to B.

Code:

```
edge(p, q). edge(q, r). edge(q, s).
edge(s, t). route(A, B) :- edge (A, C),
route(C, B). route(A, B) :- edge(A,
B).
```

Query & Output:

```
?-
% c:/users/rafs/documents/prolog/25 compiled 0.00 sec, 0 clauses
?- route(p,s).
true
```

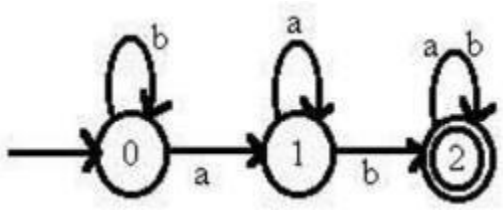
Conclusion: We have successfully implemented a tree in prolog programming in this experiment.

Experiment No:24

Experiment Name: Write a prolog program for simulating Graph & DFA.

Objectives: Learn how to simulate Graph & DFA in prolog programming.

Theory: DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time. In DFA, there is only one path for specific input from the current state to the next state. DFA does not accept the null move, i.e., the DFA cannot change state without any input character. DFA can contain multiple final states. It is used in Lexical Analysis in Compiler. In this experiment, we will specify a state table for a DFA that accepts the language (a,b)*ab(a,b)* . For example, the state diagram for this machine is:



Algorithm:

Step 1: Start.

Step 2: Write down a list of numbers using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts: delta(0,a,1). -> move 0 to 1 on input a. delta(0,b,0). -> move 0 to 0 on input b. delta(1,a,1). -> move 1 to 1 on input a. delta(1,b,2). -> move 1 to 1 on input b. delta(2,b,2). -> move 2 to 2 on input b.

delta(2,a,2). -> move 2 to 2 on input a. start(0). -> Start state 0.
final(2). -> Final state 2.

Rules: parse(L):- start(S),trans(S,L). trans(X,[A|B]):-
delta(X,A,Y), trans(Y,B).

trans(X,[]):-final(X).

Code:

```
parse(L):- start(S),trans(S,L). trans(X,[A|B]):-delta(X,A,Y),write(X),write(' '),
write([A|B]),nl,trans(Y,B). trans(X,[]):-final(X),write(X),write(' '),write([]),nl. delta(0,a,1).
delta(0,b,0). delta(1,a,1). delta(1,b,2). delta(2,b,2). delta(2,a,2). start(0). final(2).
```

Query & Output:

```
?-
% c:/users/rafs/documents/prolog/26 compiled 0.00 sec, 3 clauses
% c:/users/rafs/documents/prolog/26 compiled 0.00 sec, 0 clauses
?- parse([b,b,a,a,b,a,b]).
0 [b,b,a,a,b,a,b]
0 [b,a,a,b,a,b]
0 [a,a,b,a,b]
1 [a,b,a,b]
1 [b,a,b]
2 [a,b]
2 [b]
2 []
true
```

Conclusion: We have successfully implemented this experiment for simulating Graph & DFA.

Experiment No: 25

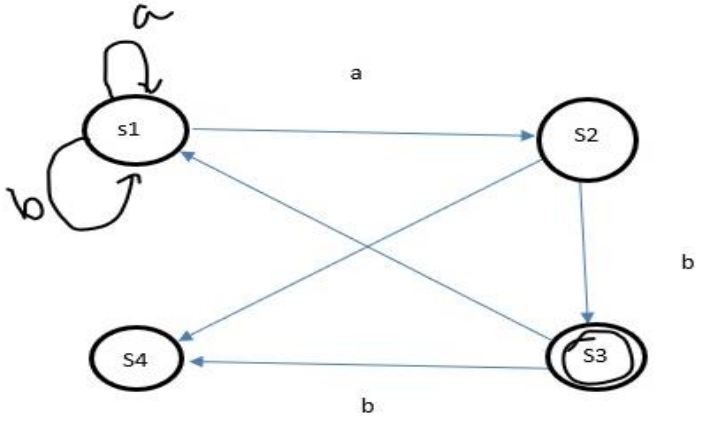
Experiment Name: Write a prolog program for simulating NFA.

Objectives: Here we will learn how to simulate NFA for prolog program.

Theory: NFA stands for non-deterministic finite automata. Simulating a NFA in Prolog, an automaton can be specified by three relations:

1. A unary relation 'final' which defines the final states of the automation.
2. A three-argument relation 'trans' which defines the state transitions so that: trans(S1, X, S2) This means that a transition from S1 to S2 is possible when the current input symbol X is read.

3. A binary relation 'silent' Silent(S1, S2) meaning that a silent move is possible from state S1 to S2.



Algorithm:

Step 1: Start.

Step 2: Write down the states using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step and finally goes to the final state. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts:

```
final(s3). # final state.
```

```
tran(s1,a,s1). # state s1 moves to s1 on input a.
tran(s1,a,s2).  # state s1 moves to s2 on input a.
tran(s1,b,s1). # state s1 moves to s1 on input s1.
tran(s2,b,s3). # state s2 moves to s3 on input s3.
tran(s3,b,s4). # state s3 moves to s4 on input s4.
silent(s2,s4). # null movement from state s2 to s4.
silent(s3,s1). # null movement from state s3 to s1.
```

Rules:

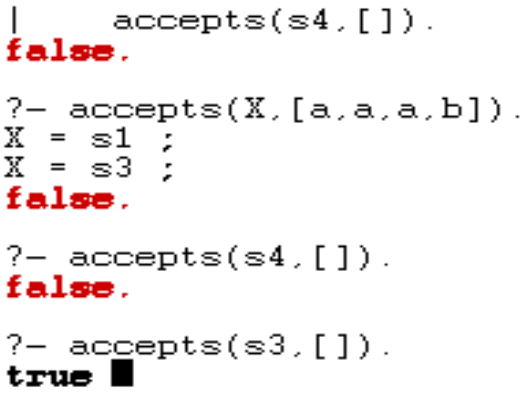
```
accepts(S,[]):-
   final(S).
accepts(S,[H|T]):-
   tran(S,H,S1),
   accepts(S1,T).
accepts(S,String):-
   silent(S,S1),
   accepts(S1,String).
```

Code:

```
final(s3).
tran(s1,a,s1).
tran(s1,a,s2).
tran(s1,b,s1).
tran(s2,b,s3).
tran(s3,b,s4).
silent(s2,s4).
silent(s3,s1).
accepts(S,[]):-
```

```
    final(S).
accepts(S,[H|T]):-
    tran(S,H,S1),
    accepts(S1,T).
accepts(S,String):-
    silent(S,S1),
    accepts(S1,String).
```

Query & Output:



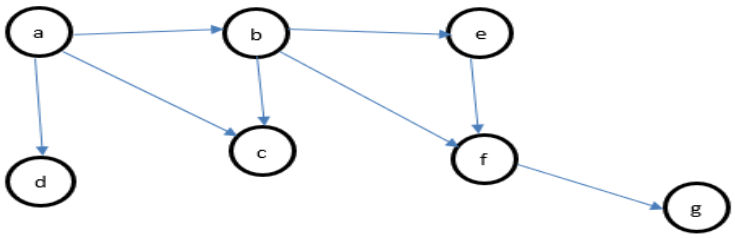
Conclusion: We have successfully executed the prolog code for NFA.

Experiment No:26

Experiment Name: Write a prolog program to find path in a graph.

Objectives: We will learn how to find a path in a graph using prolog.

Theory: To find a path in a graph, at first, we will define a graph. Then if there a path between two nodes, then the program will return true; otherwise return false.



Algorithm:

Step 1: Start.

Step 2: Write down the paths between two nodes using facts.

Step 3: Write down some rules for this facts.

Step 4: Run the query. If the query matches the rules go to the next step & finally we will go. Otherwise send error.

Step 5: The End.

Program Syntax:

Facts: Path from x to y exists.

Prolog fact representation: arc(x,y).

Rules: 1) If there is a path from X to Y:

Prolog rule representation: path(X,Y):-arc(X,Y). path(X,Y):-arc(X,Z),path(Z,Y).

2) All the path from node X to Y:

Prolog rule representation: pathall(X,X,[]). pathall(X,Y,[X,Z|L]):-arc(X,Z),pathall(Z,Y,L).

Code:

```
path(X,Y):-arc(X,Y).
```

```
path(X,Y):-arc(X,Z),path(Z,Y).
pathall(X,X,[]).
pathall(X,Y,[X,Z|L]):-arc(X,Z),pathall(Z,Y,L).
arc(a,b).
arc(a,c).
arc(b,e).
arc(b,f).
arc(b,c).
arc(a,d).
arc(e,f).
arc(f,g).
```

Query & Output:

```
?- consult('path.pl').
true.

?- path(a,b).
true .

?- path(c,g).
false.

?- pathall(a,g,R).
R = [a, b, b, e, e, f, f, g] ;
R = [a, b, b, f, f, g] ;
false.

?-
```

Conclusion: We have successfully found a path from node X to Y and all the paths from X to Y in a graph in this experiment.

Experiment No:27

Experiment Name: Reading/Writing Files in prolog.

Objectives: Here, we will learn how to read/write files in prolog programming.

Theory: There are some built-in predicates, that can be used to read from file and write into it.

The tell and told: If we want to write into a file, except the console, we can write the tell () predicate. This tell () predicate takes filename as argument. If that file is not present, then create a new file, and write into it. That file will be opened until we write the told command. We can open more than one file using tell (). When told is called, all files will be closed.

The see and seen: When we want to read from file, not from the keyboard, we have to change current input stream. So, we can use see () predicate. This will take filename as input. When the read operation is completed, then we will use seen command.

Algorithm:

Step 1: Start.

Step 2: Create a file.

Step 3: Write down some facts..

Step 4: Run the query. If the query matches the rules go to the next step. Otherwise send error.

Step 5: The End.

Code:

```
| ?- told('myFile.txt').

uncaught exception: error(existence_error(procedure,told/1),top_level/0)

| ?- told("myFile.txt").

uncaught exception: error(existence_error(procedure,told/1),top_level/0)

| ?- tell('myFile.txt').

yes

| ?- tell('myFile.txt').

yes

| ?- write('Hello World').
```

yes

| ?- write(' Writing into a file'),tab(5),write('myFile.txt'),nl.

Yes

Query & Output: Hello World Writing into a file myFile.txt

Conclusion: Successfully executed the prolog program.

Experiment No. 28

Experiment Name: Write a program for Depth first search (DFS).

Objective: Depth First Search (DFS) is to visit every node of a graph and collect some sort of information about how that node was discovered.

Theory: Depth First Search is a graph traversal technique for finding a vertex in a graph. In the search process, depth first search is also used to determine the order in which it visits the vertices. Without producing loops, depth first search finds the edges to be employed in the search process.

Code:

```
#include<bits/stdc++.h>

using namespace std;


vector<int>graph[27];

vector<int>SearchList;

bool visited[30];

bool found = false;


void DFS(int node){
   cout<<(char)(node + 'A')<<" ";
   for(int i=0; i<SearchList.size(); i++) {
      if(node == SearchList[i]) {
         found = true;
         cout<<endl;
         return;
      }
   }
   visited[node] = true;
   for(int i=0; i<graph[node].size(); i++){
      int adjacent = graph[node][i];
      if(!visited[adjacent] && !found) DFS(adjacent);
```

```
    }
}

int main(){
    freopen("input.txt", "r", stdin);
    int How_Many_edge, how_many_search;
    char root, from, to, temp;
    int from_int, to_int;
    cin>>root;
    cin>>How_Many_edge;
    for(int i=0; i<How_Many_edge; i++){
        cin>>from>>to;
        from_int = from - 'A';
        to_int = to - 'A';
        graph[from_int].push_back(to_int);
        graph[to_int].push_back(from_int);
    }
    cin>>how_many_search;
    for(int i=0; i<how_many_search; i++){
        cin>>temp;
        SearchList.push_back(temp - 'A');
    }
    DFS(root - 'A');
}
```

Input:

```
A
16
A B
A C
B D
B E
C F
C G
E H
E I
G J
G K
H L
H M
H N
I O
I P
K Q
2
N J
```

Output:

```
A B D E H L M N

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for depth first search.

Experiment No. 29

Experiment Name: Write a program for Breadth first search (BFS).

Objective: Breadth First Search (BFS) is to visit every node of a graph and collect some sort of information about how that node was discovered.

Theory: The breadth-first search or BFS algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It begins at the root of the tree or graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.

Code:

```
#include<bits/stdc++.h>

using namespace std;


vector<int>graph[27];

vector<int>SearchList;


bool visited[30];

queue<int>Q;


void BFS(int node){
   Q.push(node);
   while(!Q.empty()){
      int node = Q.front();
      Q.pop();
      cout<<(char)(node + 'A')<<" ";
      for(int i=0; i<SearchList.size(); i++)  if(node == SearchList[i]) {cout<<endl; return;}


      visited[node] = true;
      for(int i=0; i<graph[node].size(); i++){
         int adjacent = graph[node][i];
         if(!visited[adjacent]) Q.push(adjacent);
```

```
        }
    }
}

int main(){
    freopen("input.txt", "r", stdin);
    int How_Many_edge, how_many_search;
    char root, from, to, temp;
    int from_int, to_int;
    cin>>root;
    cin>>How_Many_edge;
    for(int i=0; i<How_Many_edge; i++){
        cin>>from>>to;
        from_int = from - 'A';
        to_int = to - 'A';
        graph[from_int].push_back(to_int);
        graph[to_int].push_back(from_int);
    }
    cin>>how_many_search;
    for(int i=0; i<how_many_search; i++){
        cin>>temp;
        SearchList.push_back(temp - 'A');
    }
    BFS(root - 'A');
}
```

Input:

```
A
16
A B
A C
B D
B E
C F
C G
E H
E I
G J
G K
H L
H M
H N
I O
I P
K Q
2
N J
```

Output:

```
A B C D E F G H I J

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for breadth first search.

Experiment No. 30

Experiment Name: Write a program for Depth limited search(DLS).

Objective: Solve the drawback of the infinite path in the Depth-first search.

Theory: Depth limited search is the new search algorithm for uninformed search. The unbounded tree problem happens to appear in the depth-first search algorithm, and it can be fixed by imposing a boundary or a limit to the depth of the search domain.

Code:

```
#include<bits/stdc++.h>
using namespace std;

vector<int>graph[27];
vector<int>SearchList;
bool visited[30];
bool found = false;
int limit;

void DLS(int node, int level){
    if(level>limit) return;
    cout<<(char)(node + 'A')<<" ";
    for(int i=0; i<SearchList.size(); i++) {
        if(node == SearchList[i]) {
            found = true;
            cout<<endl;
            return;
        }
    }
    visited[node] = true;
    for(int i=0; i<graph[node].size(); i++){
        int adjacent = graph[node][i];
        if(!visited[adjacent] && !found) DLS(adjacent, level + 1);
    }
}

int main(){
    freopen("input_DLS.txt", "r", stdin);
    int How_Many_edge, how_many_search;
    char root, from, to, temp;
    int from_int, to_int;
    cin>>root;
    cin>>How_Many_edge;
    for(int i=0; i<How_Many_edge; i++){
```

```
        cin>>from>>to;
        from_int = from - 'A';
        to_int = to - 'A';
        graph[from_int].push_back(to_int);
        graph[to_int].push_back(from_int);
    }
    cin>>how_many_search;
    for(int i=0; i<how_many_search; i++){
        cin>>temp;
        SearchList.push_back(temp - 'A');
    }
    cin>>limit;
    DLS(root - 'A', 1);
    if(found) cout<<"YES, We reach the goal, with this limit"<<endl;
    else cout<<"\nNo, We cann't reach the goal, with this limit"<<endl;
}
```

Input:

```
S
10
S A
S B
A C
A D
B I
B J
C E
C F
D G
I H
1
J
3
```

Output:

```
S A C D B I J
YES, We reach the goal, with this limit

Process returned 0 (0x0)   execution time : 0.193 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for Depth limited search.

Experiment No. 31

Experiment Name: Write a program for Iterative Deepening Search (IDS).

Objective: Solve the drawback of the infinite path in the Depth-first search.

Theory: Depth limited search is the new search algorithm for uninformed search. The unbounded tree problem happens to appear in the depth-first search algorithm, and it can be fixed by imposing a boundary or a limit to the depth of the search domain.

Code:

```
#include<bits/stdc++.h>
using namespace std;

vector<int>graph[27];
vector<int>SearchList;
bool visited[30];
bool found = false;
int limit;

void DLS(int node, int level = 1){
   if(level>limit) return;
   cout<<(char)(node + 'A')<<" ";
   for(int i=0; i<SearchList.size(); i++) {
      if(node == SearchList[i]) {
         found = true;
         cout<<endl;
         return;
      }
   }
   visited[node] = true;
   for(int i=0; i<graph[node].size(); i++){
```

```
        int adjacent = graph[node][i];
        if(!visited[adjacent] && !found) DLS(adjacent, level + 1);
    }
}

int main(){
    freopen("input_IDS.txt", "r", stdin);
    int How_Many_edge, how_many_search;
    char root, from, to, temp;
    int from_int, to_int;
    cin>>root;
    cin>>How_Many_edge;
    for(int i=0; i<How_Many_edge; i++){
        cin>>from>>to;
        from_int = from - 'A';
        to_int = to - 'A';
        graph[from_int].push_back(to_int);
        graph[to_int].push_back(from_int);
    }
    cin>>how_many_search;
    for(int i=0; i<how_many_search; i++){
        cin>>temp;
        SearchList.push_back(temp - 'A');
    }
    for(limit = 1; limit <= 5; limit++){
        DLS(root - 'A', 1);
        if(found){
            cout<<"YES, We reach the goal, within limit: "<<limit<<endl;
```

```
            break;
        }
        cout<<endl;
        memset(visited, false, sizeof(visited));
    }
    if(!found) cout<<"\nNo, We cann't reach the goal, with this limit"<<endl;
}
```

Input:

```
A
17
A B
A C
A D
B E
B F
C G
D H
E I
F J
F K
G L
H M
H N
K O
K P
L R
N S
1
R
```

Output:

```
A
A B C D
A B E F C G D H
A B E I F J K C G L D H M N
A B E I F J K O P C G L R
YES, We reach the goal, within limit: 5

Process returned 0 (0x0)   execution time : 0.211 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for Iterative Deeping Search.

Experiment No. 32

Experiment Name: Write a program for Bidirectional Search.

Objective: To find smallest path form source to goal vertex.

Theory: Bidirectional Search is Graph Search Algorithm where two graph traversals (BFS) take place at the same time and is used to find the shortest distance between a fixed start vertex and end vertex. It is a faster approach, reduces the time required for traversing the graph. It can be used for other applications as well.

Code:

```
#include<bits/stdc++.h>
using namespace std;

vector<int>graph[100];
queue<int> Forward, Backword;
bool ForwardVisited[100], BackwordVisited[100];

void BiDirectional_Search(int root, int goal){
   Forward.push(root);
   Backword.push(goal);
   while(!Forward.empty() && !Backword.empty()){
      if(!Forward.empty()){
         int node = Forward.front();
         Forward.pop();
         cout<<node<<" ";
         ForwardVisited[node] = true;
         if(BackwordVisited[node]) return;
         for(int i=0; i<graph[node].size(); i++){
            int adjacent = graph[node][i];
            if(!ForwardVisited[adjacent]) Forward.push(adjacent);
         }
```

```
        }
        if(!Backword.empty()){
            int node = Backword.front();
            Backword.pop();
            cout<<node<<" ";
            BackwordVisited[node] = true;
            if(ForwardVisited[node]) return;
            for(int i=0; i<graph[node].size(); i++){
                int adjacent = graph[node][i];
                if(!BackwordVisited[adjacent]) Backword.push(adjacent);
            }
        }
    }
}
int main(){
    freopen("input_BiDirectional.txt", "r", stdin);
    int How_Many_edge, how_many_search;
    int from, to, root, goal;
    cin>>How_Many_edge;
    for(int i=0; i<How_Many_edge; i++){
        cin>>from>>to;
        graph[from].push_back(to);
        graph[to].push_back(from);
    }
    cin>>root;
    cin>>goal;
    BiDirectional_Search(root, goal);
    return 0;
```

}

Input:

```
14
1 4
2 4
3 6
5 6
4 8
6 8
8 9
9 10
10 11
10 12
11 13
11 14
12 15
12 16
1
16
```

Output:

```
1 16 4 12 2 10 8 15 6 9 9
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for Bidirectional Search.

Experiment No. 33

Experiment Name: Write a program for Greedy Best First Search.

Objective: To find the most promising path from a given starting point to a goal.

Theory: Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search.

Code:

```
#include<bits/stdc++.h>
using namespace std;

int Heuristic[26] = {0};
vector<int> graph[26];
char Goal;
bool found, visited[26];
vector<int>Closed;
void Best_First_Search(int root){
   priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,int>> >Open;
   Open.push({Heuristic[root], root});
   while(!Open.empty()){
      pair<int, int> top = Open.top();
      Open.pop();
      Closed.push_back(top.second);
      visited[top.second] = true;
      if(top.second == (Goal - 'A')) {
         found = true;
         return;
      }
      for(int i=0; i<graph[top.second].size(); i++){
```

```
            int adjacent = graph[top.second][i];
            if(!visited[adjacent]) Open.push({Heuristic[adjacent], adjacent});
        }
    }
}
int main(){
    freopen("GreedyBestFirstSearch.txt", "r", stdin);
    int How_many_node, How_many_edge, heuristic, cost;
    char ch, node1, node2, Source;

    cin>>How_many_node;
    for(int i = 0; i<How_many_node; i++){
        cin>>ch>>heuristic;
        Heuristic[ch - 'A'] = heuristic;
    }
    cin>>How_many_edge;
    for(int i = 0; i<How_many_edge; i++){
        cin>>node1>>node2>>cost;
        graph[node1 - 'A'].push_back(node2 - 'A');
        graph[node2 - 'A'].push_back(node1 - 'A');
    }
    cin>>Source;
    cin>>Goal;
    Best_First_Search(Source - 'A');
    if(found){
        for(auto node : Closed) cout<<(char)(node + 'A')<<" ";
        cout<<endl;
    }
```

```
    else cout<<"Goal not Found"<<endl;
    return 0;
}
```

Input:

```
10
A 12
B 4
C 7
D 3
E 8
F 2
H 4
I 9
S 13
G 0
9
S A 3
S B 2
A C 4
A D 1
B E 3
B F 1
E H 5
F I 2
F G 3
S
G
```

Output:

```
S B F G

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for greedy best first search.

Experiment No. 34

Experiment Name: Write a program for A* Best First Search.

Objective: To find the shortest path which uses distance as a heuristic.

Theory: It is a searching algorithm that is used to find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken.

Code:

```
#include<bits/stdc++.h>

using namespace std;


int Heuristic[26] = {0};

vector<int> graph[26];

char Goal;

bool found, visited[26];

vector<int>Closed;

int COST[26][26];


struct Node{
   int nodeNo;
   int Commulative_cost;
};
struct Comp{
   bool operator()(Node const& n1, Node const& n2){
      return (n1.Commulative_cost+Heuristic[n1.nodeNo] >
n2.Commulative_cost+Heuristic[n2.nodeNo]);
   }
};


void A_Star_Search(int root){
```

```
    priority_queue<Node, vector<Node>, Comp> Open;
    Node node;
    node.nodeNo = root;
    node.Commulative_cost = 0;
    Open.push(node);
    while(!Open.empty()){
        Node top = Open.top();
        Open.pop();
        visited[top.nodeNo] = true;
        Closed.push_back(top.nodeNo);
        if(top.nodeNo == (Goal - 'A')) {
            found = true;
            return;
        }
        for(int i=0; i<graph[top.nodeNo].size(); i++){
            int adjacent = graph[top.nodeNo][i];
            if(!visited[adjacent]){
                Node adjacentNode;
                adjacentNode.nodeNo = adjacent;
                adjacentNode.Commulative_cost = top.Commulative_cost +
COST[top.nodeNo][adjacent];
                Open.push(adjacentNode);
            }
        }
    }
}

int main(){
    freopen("A_Star_Search.txt", "r", stdin);
```

```
    int How_many_node, How_many_edge, heuristic, cost;
    char ch, node1, node2, Source;
    cin>>How_many_node;
    for(int i = 0; i<How_many_node; i++){
        cin>>ch>>heuristic;
        Heuristic[ch - 'A'] = heuristic;
    }
    cin>>How_many_edge;
    for(int i = 0; i<How_many_edge; i++){
        cin>>node1>>node2>>cost;
        graph[node1 - 'A'].push_back(node2 - 'A');
        graph[node2 - 'A'].push_back(node1 - 'A');
        COST[node1 - 'A'][node2 - 'A'] = cost;
        COST[node2 - 'A'][node1 - 'A'] = cost;
    }
    cin>>Source;
    cin>>Goal;

    A_Star_Search(Source - 'A');

    if(found){
        for(auto node : Closed) cout<<(char)(node + 'A')<<" ";
        cout<<endl;
    }
    else cout<<"Goal not Found"<<endl;
    return 0;
}
```

Input:

```
6
S 5
A 3
B 4
C 2
D 6
G 0
8
S A 1
S G 10
A C 1
A B 2
B D 5
C D 3
C G 4
D G 2
S
G
```

Output:

```
S A C G

Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```

Conclusion: We have successfully implemented this experiment for A* best first search.