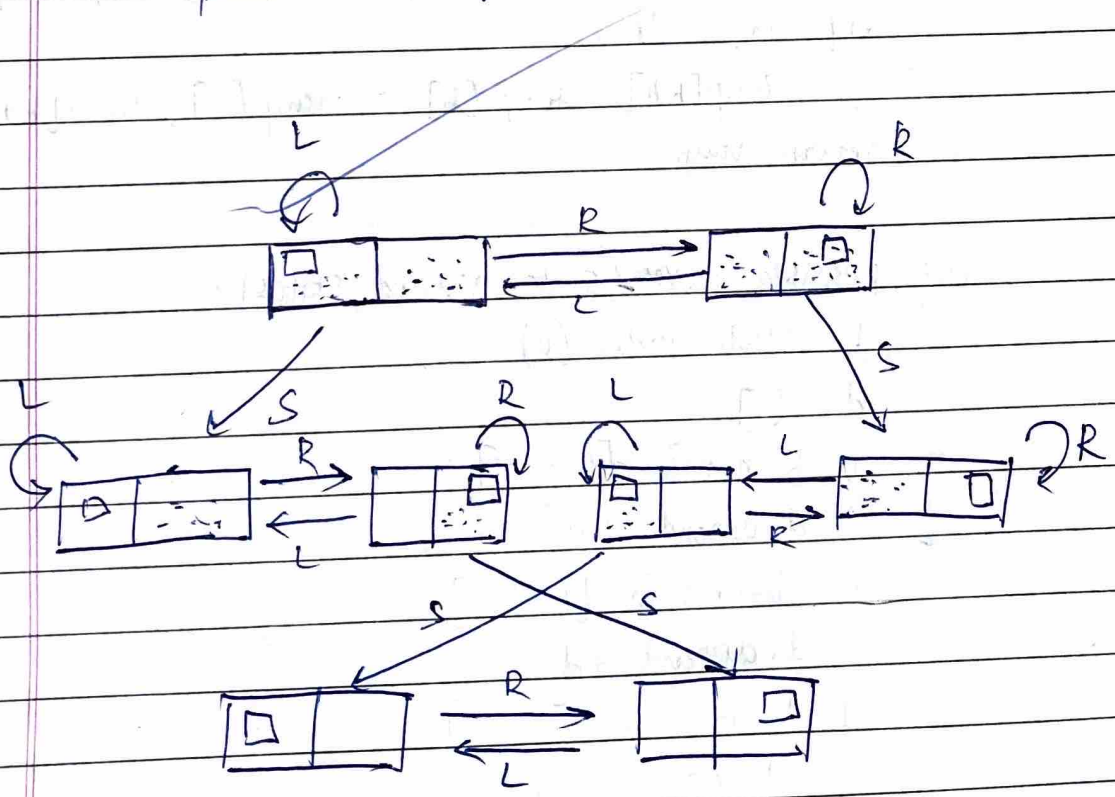
Vacuum Cleaner

```
def vaccuum-world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location = input("Enter location of vacuum")
    Status = input("Enter status of location :")
    status_complement = input("Enter Status of other room")
    print("Initial location condition" + str(goal-state))
    if location == '1':
        print("Location A is dirty")
        goal-state['A'] = '0'
        cost += 1
        print("Cost for cleaning A" + str(cost))
        print("Location A has been cleaned")
        if status_complement == '1':
            print("Location B is dirty")
            print("Moving right to location B")
            cost += 1
            print("Cost for moving right" + str(cost))
            goal-state['B'] = '0'
            cost += 1
            print("Cost for suck" + str(cost))
            print("Location B has been cleaned")
        else:
            print("No action" + str(cost))
            print("Location B is cleaned")

    if Status == '0'
        print("Location A is already clean")
```

```
if Status - Complement == '1':
    print("Location B is dirty")
    print("Moving RIGHT to Location B")
    cost += 1
    print("Cost for moving right " + Str(cost))
    goal_state['B'] = '0'
    cost += 1
    print("Cost for Suck" + cost)
    print("Location B has been cleaned")
else:
    print("NO action" + str(cost))
    print(cost)
    print("Location B is already clean")
else:
    print("Vacuum is placed in location B")
    if Status == '1':
        print("Location B is dirty")
        goal_state['B'] = '0'
        cost += 1
        print("Cost for cleaning" + Str(cost))
        print("Location B has been cleaned")
        if Status_complement == '1':
            print("Location A is dirty")
            print("Moving left to location A")
            cost += 1
            print("Cost for moving left" + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("Cost for Suck" + cost)
            print("Location A has been cleaned")
        else
            print(cost)
```

print ("Location B is already clean")
if Status_complement == '1':
print ("Location A is dirty")
print ("Moving left to location A")
cost += 1
print ("Cost for moving left" + cost)
goal_state['A'] = '0'
cost += 1
print ("Cost for suck" + cost)
print ("Location A has been cleaned")
else
print ("No action" + cost)
print ("Location A is already clean")
print ("Goal state")
print (goal_state)
print ("Performance measurement" + cost)



Program-2

8 puzzle using BFS

```
import numpy as np
import pandas as pd
import os

def gen(state, m, b):
    temp = state.copy()

    if m == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    elif m == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    elif m == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    elif m == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    return temp

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []
    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
```

```
    pos_moves = []
    for i in d:
        pos_moves.append(gen(state, i, b))
    return [move for move in pos_moves if move not
                                    in visited_states]

def bfs(src, target)
    queue = []
    queue.append(src)
    cost = 0

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)
        print("queue \n", source)
        cost = cost + 1
        if source == target:
            print("Success")
            print("path cost", cost)
            return

        moves = possible_moves(source, exp)
        for move in moves:
            if move not in exp and move not in queue:
                queue.append(move)

src = [1,2,3,4,5,6,0,7,8]
target = [1,2,3,4,5,6,7,8,0]
bfs(src, target)
```

OUTPUT

queue :

[0, 2, 3, 1, 5, 6, 4, 7, 8]

[1, 2, 3, 5, 0, 6, 4, 7, 8]

1 | 2 | 3
4 | 5 | 6
7 | 0 | 8

queue :

[1, 2, 3, 5, 0, 6, 4, 7, 8]

[1, 2, 3, 4, 0, 6, 7, 5, 8]

[1, 2, 3, 4, 5, 6, 7, 8, 0]

0 | 2 | 3
1 | 5 | 6
4 | 7 | 8

queue :

[2, 0, 3, 1, 5, 6, 4, 7, 8]

[1, 0, 3, 5, 2, 6, 4, 7, 8]

1 | 2 | 3
4 | 0 | 6
7 | 5 | 8

queue :

[1, 2, 3, 4, 6, 0, 7, 5, 8]

1 | 2 | 3
4 | 5 | 6
7 | 8 | 0

Program-3
Tic Tac Toe

```
[redacted] = [[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]]

def print_board():
    for row in board:
        print("|".join(row))
        print("-" * 5)

def check_winner(player):
    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or
           all([board[j][i] == player for j in range(3)]):
            return True
    if all([board[i][i] == player for i in range(3)]) or
       all([board[i][2-i] == player for i in range(3)]):
        return True
    return False

def is_Full():
    return all([cell != " " for row in board for cell in row])

def minimax(depth, ismaximizing):
    if check_winner("X"):
        return -1
    if check_winner("O"):
        return 1
```

```
if is_full():
    return 0
if is_maximizing:
    max_eval = float("-inf")
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
            eval = minimax(depth+1, False)
            board[i][j] = " "
            max_eval = max(max_eval, eval)
    return max_eval
else:
    min_eval = float("inf")
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "X"
                eval = minimax(depth+1, True)
                board[i][j] = " "
                min_eval = min(min_eval, eval)
    return min_eval.

def ai_move():
    best_move = None
    best_eval = float("-inf")
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                eval = minimax(0, False)
                board[i][j] = " "
```

```
    if eval > best_eval:
        best_eval = eval
        best_move = (i, j)
Return best_move


while not is_full() and not check_winner("X") and not
 check_winner("O"):
    print_board()
    row = input("Enter row: ")
    col = int(input("Enter column: "))
    if board[row][col] == " ":
        board[row][col] = "X"
        if check_winner("X"):
            print_board()
            print("You win!")
            break
        if is_full():
            print_board()
            print("It's a draw!")
            break
        ai_row, ai_col = ai_move()
        board[ai_row][ai_col] = "O"
        if check_winner("O"):
            print_board()
            print("AI wins!")
            break
    else:
        print("Cell is already occupied").
```

OUTPUT

```
  |  |
--|--|--
  |  |
--|--|--
  |  |
```

Enter row : 0
Enter col : 0

```
X|  |
 |0|
--|--
 |  |
```

Enter row : 2
Enter col : 2

```
X|0|
 |0|
--|--
 |  |X
```

Enter row : 2
Enter col : 1

```
X|0|
 |0|
--|--
0|X|X
```

Enter row : 0
Enter col : 1

```
X|0|X
X|0|0
0|X|X
```

It's a draw

Iterative Deepening Search

Aim:

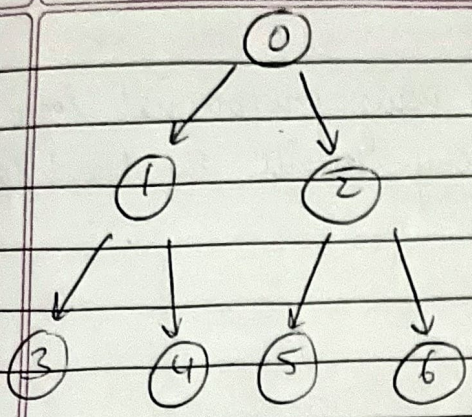
Implement Iterative Deepening Search

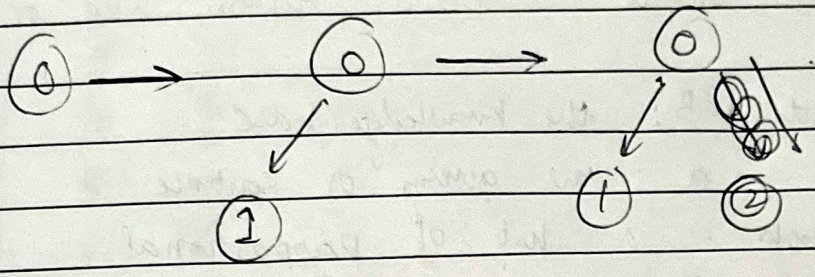
```
from collection import defaultdict

class graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(last)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DLS(self, src, target, maxDepth):
        if src == Target:
            return True
        if maxDepth <= 0:
            return False
        if i in self.graph[src]:
            if self.DLS(i, target, maxDepth-1):
                return True
        return False

    def IDDFS(self, src, target, maxDepth):
        for i in range(maxDepth):
            if self.DLS(src, target, 1):
                return True
        return False
```

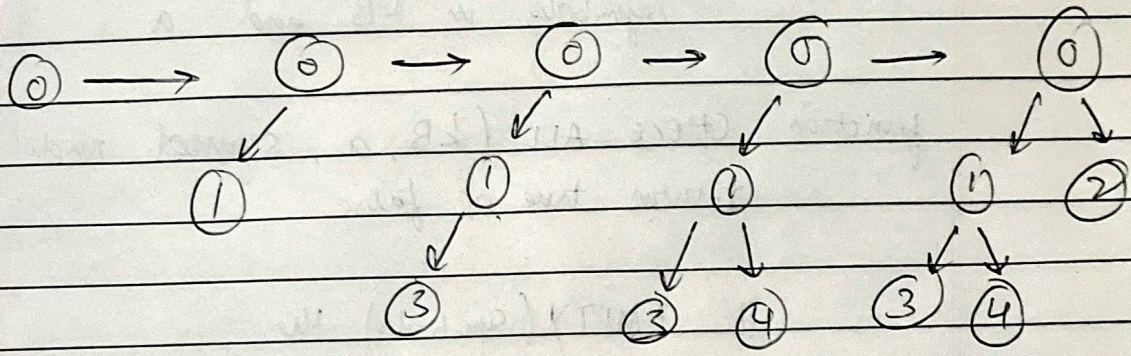
```
n = int (input ("Enter number of vertices : "))

g = Graph (n)

e = int (input ("Enter the number of Edge : "))
for _ in range (e):
    u,v = map (int, input ("Enter edge(u,v) : ").split())
    g. addEdge (u,v)

src = int (input ("Enter the source node : "))
target = int (input ("Enter the target node : "))
maxDepth = int (input ("Enter maximum depth : "))

if g. IDDFS (src, target, maxDepth):
    print (f"Target {target} is reachable from source
            {src} with max depth {maxDepth}")
else
    print (f"Target {target} is NOT reachable from source
            {src} with max depth {maxDepth}").
```

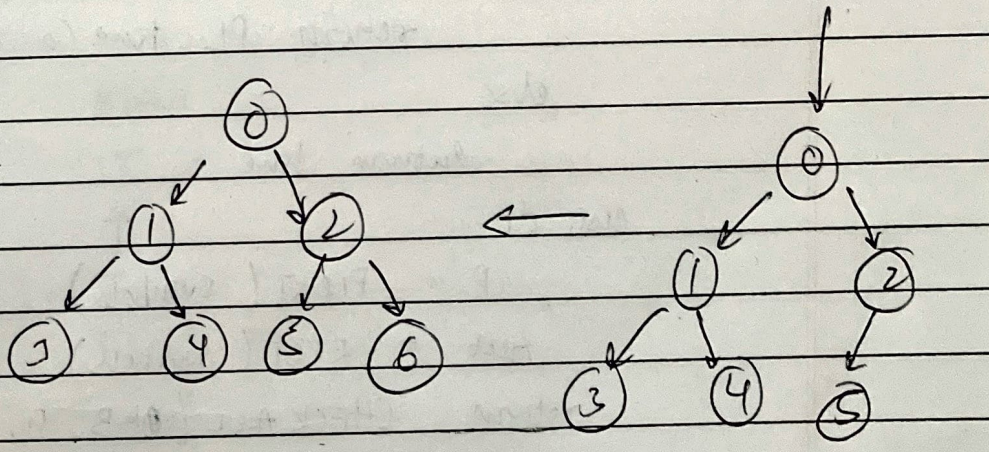
OUTPUT

```
Enter no of Vertices : 8
Enter no of edges : 7
Enter edge (u,v) : 0 1      Enter edge (u,v) : 0 2
Enter edge (u,v) : 1 3      Enter edge (u,v) : 1 4
Enter edge (u,v) : 2 5      Enter edge (u,v) : 2 6
Enter edge (u,v) : 3 7

Enter source node : 0
Enter target : 7
Enter max depth : 2
    Target 7 is not reachable from 0 with maxdepth 2
```









Implement 8-puzzle problem using A* algo

A* algo

Optimised value $f(n) = g(n) + h(n)$

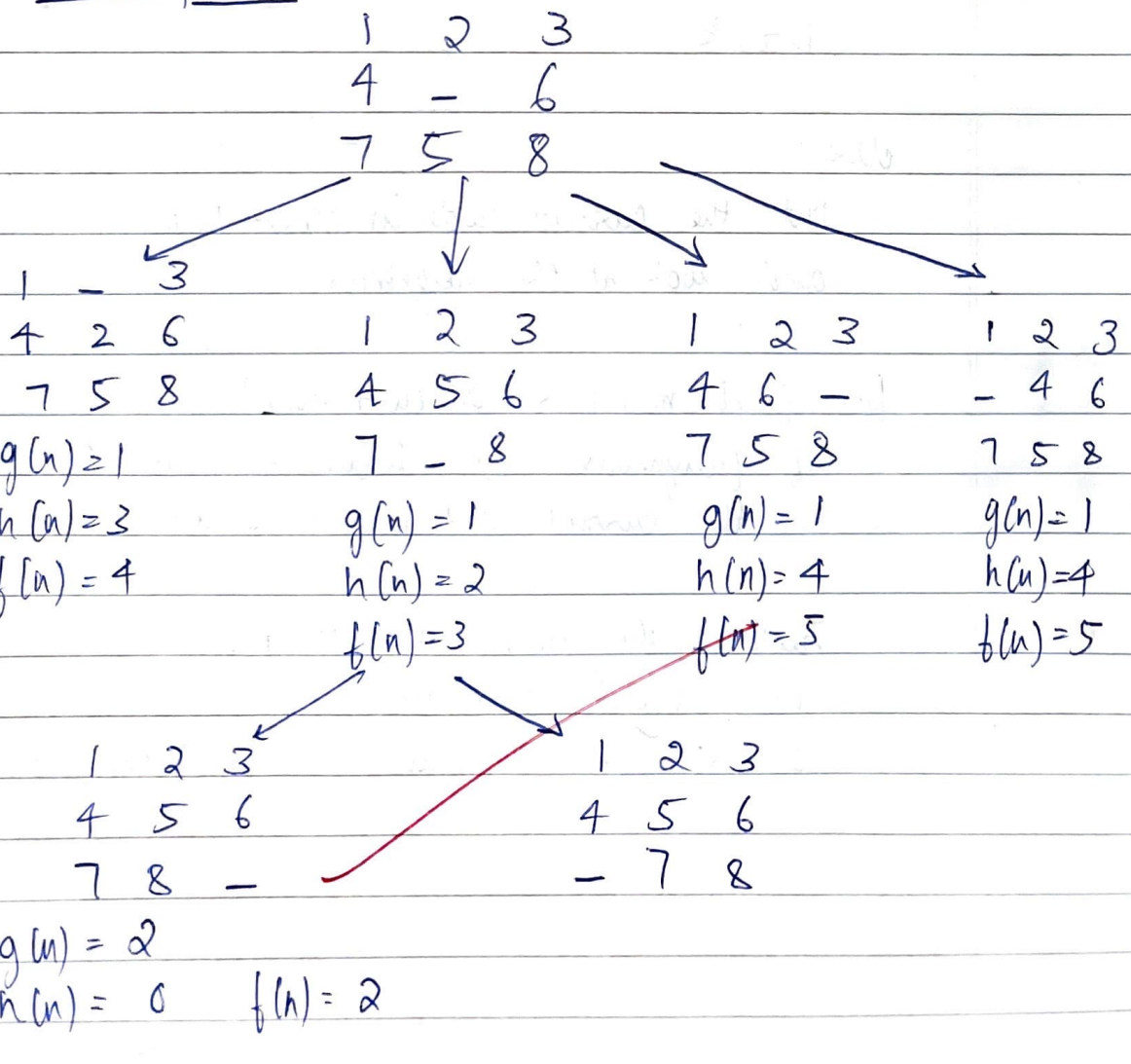
Start State

1	2	3
4	_	6
7	5	8

Goal State

1	2	3
4	5	6
7	8	_

State Space Tree:



Algorithm :

Input : Graph (Gr) - Heuristic value h(n), start state (s), Goal node (Gr)

Output : Cost of the path

- Make an open list containing only start node
- Make an empty closed list

while (goal state not reached) :
    Consider the node with the lowest f(n) value in the open list

if (node is destination node):
    print the cost
    break

else :
    put the current node in closed list and look at the neighbours

for (each neighbour of curr node) :
    if (neighbour has lower g value) :
        the current and is in closed list.

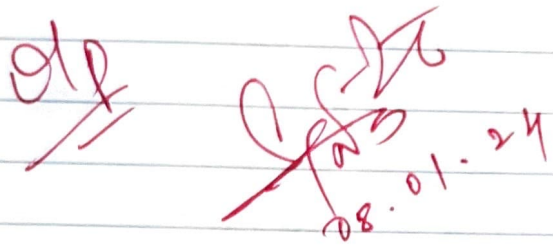
    replace the neighbour with the new, lower g value
    current node is now the neighbour's parent.

else if (current g value is lower and this neighbour is in open list):
replace the neighbour with the new lower g value.

change the neighbour's parent to our current node

else if (the neighbour is not in both the lists)

add it to the open list and set it's g value



08.01.24

Program 6

Create a knowledge bot using propositional logic and show that the given query entails the knowledge base or not.

Algorithm

function Entails ? (kB, a) return true or false

input kB: the knowledge base
a : the query or sentence
Symbols : a list of propositional symbols in kB and a

function CHECK-ALL (kB, a, symbol, model)
return true or false

if EMPTY (symbols) the
if PL_true ? (kB, model) then
return PL-true (a, model)
else
return true
else do
P = FIRST (symbols)
rest = REST (symbols)
return CHECKALL (kB, a, rest, extend (p, true, model))
and CHECKALL (kB, a, rest, extend (p, false, model))

Knowledge base (Truth Table)

KB : $(p \vee q) \wedge (p \wedge \neg r)$

p	q	r	p ∨ q	¬r	p ∧ ¬r	(p∨q) ∧ (p∧¬r)
T	T	T	T	F	F	F
T	T	F	T	T	T	T
T	F	T	T	F	F	F
T	F	F	T	T	T	T
F	T	T	T	F	F	F
F	T	F	T	T	F	F
F	F	T	F	F	F	F
F	F	F	F	T	F	F

The knowledge base entails the query

OUTPUT

Enter rule : $(p \vee q) \wedge (p \wedge \neg r)$

Enter query : p

Truth table

KB	Alpha
F	T
T	T
F	T
T	T
F	F
F	F
F	F
F	F

Knowledge base entails query

If query = r,

The knowledge base doesn't entail query.

Program 7

Create a knowledge base using propositional logic and prove the given query using resolution

Algorithm

function PL-RESOLUTION (KB, α) return true or false

inputs: KB → the knowledge Base, a sentence in propositional logic

α → the query sentence in proportional clause

the set of clause in CNF representation of KB ∧ ¬α

new ← { }

loop do

for each pair of clause, C_i, C_j in clauses

resolve ← PL-RESOLVE(C_i, C_j)

if resolve contains empty clause

return true

new ← new ∨ resolve

if new ⊆ clause then return false

clause ← clause ∨ new

KB

p

p ∧ q → r

s ∨ t → q

t

s

Query

r

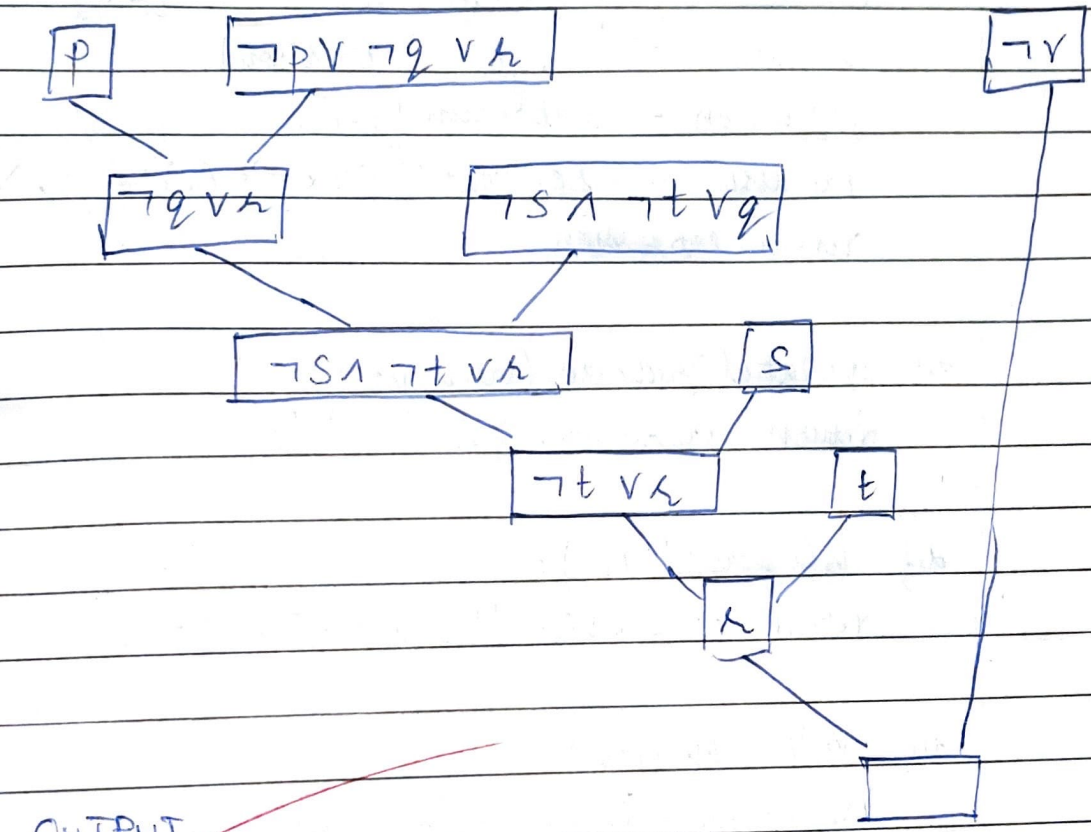
⇒ KB ∨ ¬α

Convert to CNF

$\neg (p \wedge q) \vee r \rightarrow \neg p \vee \neg q \vee r$

$\neg (s \vee t) \vee q \rightarrow \neg s \wedge \neg t \vee q$

Resolution



OUTPUT

True

Program 8

Implement unification in first order logic

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\(.),(?!.\))", expression)
    return expression

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"
```

```
def unify (exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant (exp1) and isConstant (exp2):
        if exp1 != exp2:
            return False

    if isConstant (exp1):
        return [(exp1, exp2)]

    if isConstant (exp2):
        return [(exp2, exp1)]

    if isVariable (exp1):
        if checkOccurs (exp1, exp2):
            return False
        else
            return [(exp2, exp1)]

    if isVariable (exp2):
        if checkOccurs (exp2, exp1):
            return False
        else
            return [(exp1, exp2)]

    if getInitialPredicate (exp1) != getInitialPredicate (exp2):
        print ("Predicates do not match. Cannot be unified")
        return False
```

OUTPUT

Substitutions :

```
[('x', 'Richard')]
```

Program 9

Convert a given FOL statement into CNF

```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = '[a-z~]+\([A-Z a-z ,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.replace('~[', '')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    string = ''.join(s)
    string = string.replace('~~', '')
    return f'[{string}]' if flag else string
```

```
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range
                        (ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃],', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
    statements = re.findall('\([^()]+\)', statement)
    for S in statements:
        statement = statement.replace(S, S[1:-1])
    for predicate in getPredicates(statement):
        attributes = getAttributes(predicate)
        if ''.join(attributes).islower():
            statement = statement.replace(match[1],
                                          SKOLEM_CONSTANTS.
                                          pop(0))
        else:
            aL = [a for a in attributes if a.islower()]
            aU = [a for a in attributes if not a.islower()]
            statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.
                        pop(0)}({aL[0] if len(aL) else
                                 match[1]})')

    return statement.

print(Skolemization(fol_to_cnf("animal(y) <=> loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x [∀y [animal(y) => loves(x,y)]
                               => [∃z [loves(z,x)]]")))
print(fol_to_cnf("american(x) & weapon(y) & sells(x,y,z)
                 & hostile(z)] => criminal(x)"))
```

OUTPUT

[~animal(y) | loves(x,y)] & [~loves(x,y) | animal(y)]

[animal(G(x)) & ~loves(x, G(x))] | [loves(F(x), x)]

[~american(x) | ~weapon(y) | ~sells(x,y,z) | ~hostile(z)] | criminal(x)

Program 10

Create a KB consisting of FOL statements and prove the given query using forward reasoning

```
import re

def isVariable(x):
    return len(x)==1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').
                 split(',')
        return [predicate, params]
```

```
    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.args]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.args]

class KB:

    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1
```

```
def display(self):
    print("All facts:")
    for i,f in enumerate(set([f.expression for f in
                              self.facts])):
        print(f'{i+1}. {f}')

kb = KB()
kb.tell('missile(x) => weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America) => hostile(x)')
kb.tell('american(west)')
kb.tell('enemy(Nono, America)')
kb.tell('owns(Nono, M1)')
kb.tell('missile(x) & owns(Nono,x) => sells(West, x, Nono)')
kb.tell('criminal(x)')
kb.display()
```

OUTPUT

Querying criminal(x):

1. Criminal(West)

All facts:

1. criminal(west)
2. sells(west, M1, Nono)
3. weapon(M1)
4. hostile(Nono)
5. owns(Nono, M1)
6. American(west)
7. missile(M1)
8. enemy(Nono, America)