ROBDD Used Functions

- Here we start by impporting the libraries we're going to use like
- `graphviz` which is used for drawing the ROBDD
- `numpy` which helped in dealing with arrays as we're used to MATLAB syntax
- `random` which helps us create different images when calling the same class so they have different names
- `colorama` which helps us print using different colors

In [1]:
```
import numpy as np
import graphviz
import random
import colorama
from colorama import Fore

class Node:
    def __init__(self, value):
        self.value = value
        self.lc = None
        self.rc = None
```

We here created class ROBDD which is called upon whenever a new boolean function is entered

First item in the class

`__init__`

It takes one input which is the boolean equation and creates the needed parameters to complete computations like

- `Var` which contains the variables extracted from the equation
- `nNodes` which contains the number of nodes
- `nVars` which counts the number of variables
- While loop which arranges the variables
- Rest of the functions which will be called during the initialization of the ROBDD

In [2]:
```
class ROBDD:
    # Init the ROBDD with nVars = number of variables in the expression
    def __init__(self, eq):
        self.Var = []
        self.eq = eq
        self.Vardict = {}
        self.nNodes = 0
        self.x = []
        self.nVars = 0
        self.parser(self.eq, self.Var, 0)
```

```
        self.i=0
        while self.i<len(self.Var)-1:
            self.arrange(self)
            self.i += 1
        self.i = 0

        for item in range(self.nVars):
            self.x.append(0)

        self.initT(self)
        self.initH(self)
        self.build(0)

        self.u = np.zeros((len(self.T),1), dtype = int)
        self.rename(self)

        self.newu=[]
        self.parser(self.u, self.newu, 1)
        self.newu=['0']+['1']+self.newu
        self.noding(2, 0)

        self.visited=set()
        self.graph = graphviz.Digraph()
        self.create_graph(self.newu[(len(self.newu)-1)], self.graph,self.visited)
        self.graph.render('robdd'+str(random.randint(0,1000))+'', format='png', vie

        self.satisfiability()
```

`initT`

This function is used to create the table in which the

- Level Number `i`
- Low Child `LC`
- High Child `HC`

will be stored

In [3]:
```
    # Init the Table with 0,1 entries
    @staticmethod
    def initT(self):
        self.T = 2*[3*[None]]
        self.T[0] = [self.nVars + 1,-1,-1]
        self.T[1] = [self.nVars + 1,-2,-2]  # Using a different dummy entry -2, to
        self.nNodes = 2
```

`initH`

This is the HashTable where different rows of the Table are concatenated `i+LC+HC` together and stored here to be compared with different upcoming entries where the hashed values

are compared

In [4]:
```
    # Init the HashTable with hashes for 0 and 1
    @staticmethod
    def initH(self):
        self.H = 2 * [None]
        self.H[0] = self.hashH(self,0)
        self.H[1] = self.hashH(self,1)


    # Create Hashes for 0 and 1
    @staticmethod
    def hashH(self,nodeNum):
        return hash(str(self.T[nodeNum][0])+str(self.T[nodeNum][1])+str(self.T[node
```

Here we have different functions which are critical in creating our ROBDD

member

This checks if the upcoming node was previously added by comparing it to the entries in HashTable

lookup

Returns the index of the matched entry

add

Adds a new Entry to our Table

insert

Adds a hashed Entry corresponding to the one added in our Table

make

Uses lookup and member to check if the entry to be added exists if not then it adds new entry using add and insert

build

Creates the Entries to be added then sends them to make

In [5]:
```
    # Check if an entry exists in H
    def member(self, i, l, h):
        if hash(str(i) + str(l) + str(h)) in self.H:
            return True
        else:
            return False
```

```
    # Return the index of item with a given (i,l,h) from H
    def lookup(self,i,l,h):
        return self.H.index(hash(str(i) + str(l) + str(h)))


    # Add an entry to the table T, with values(i,l,h)
    def add(self,i,l,h):
        self.T.append([i,l,h])
        self.nNodes = self.nNodes + 1
        return self.nNodes - 1


    # Add an entry to H, with (i,l,h)
    def insert(self,i,l,h):
        self.H.append(hash(str(i) + str(l) + str(h)))


    # Add an entry to the T table.
    def make(self,i,l,h):
        if l == h:
            return l
        elif self.member(i,l,h):
            return self.lookup(i,l,h)
        else:
            u = self.add(i,l,h)
            self.insert(i,l,h)
            return u


    # Create the Table and the Inverse Hash Table.
    def build(self,i=0):
        if i>=self.nVars:
            return self.my_eva()
        else:
            self.x[i] = 0
            l = self.build(i+1)
            self.x[i] = 1
            h = self.build(i+1)
            return self.make(i,l,h)
```

my_eva

iterates over the number of values assigning corresponding values for the variables in `Vardict`
eg. `ABC==>000` , when we evaluate using `eval` we want it to substitute `A=0 B=0 C=0`

In [6]:
```
    def my_eva(self):
        q=0
        for i in self.Var:

            self.Vardict[i]=self.x[q]
            q+=1
```

```
        if(eval(self.eq,self.Vardict) == True):
            return 1

        elif (eval(self.eq,self.Vardict) == False):
            return 0

        else: return eval(self.eq,self.Vardict)
```

Parser

The parser here has 2 functionalities

- Change unknown operators to python for their corresponding operators in python libraries
- Extracts the number of different variables and appends them to the array `Var`

In [7]:
```
    def parser(self, eq, Var, switch):
        for i in eq:
            if((i >= 'a') and (i <= 'z')):
                if(i in Var):
                    continue
                else: Var.append(i)
        if(switch == 0):
            eq = eq.replace("&", " and ").replace("+", " or ").replace("!", " not "
            self.eq = eq
            self.Var = Var
        else:
            self.u = eq
            self.newu = Var
```

arrange

Arranges the values appended to Var ascendingly
eg. input is `B&C+A` then `Var=[A,B,C]`

In [8]:
```
    @staticmethod
    def arrange(self):
        i = 0
        while i < (len(self.Var)-1):
            if(self.Var[i]>self.Var[i+1]):
                temp=self.Var[i]
                self.Var[i]=self.Var[i+1]
                self.Var[i+1]=temp
            i = i + 1
        self.nVars = len(self.Var)
```

rename

At first Variables where treated as numbers where `A=0` `B=1` but for the values to be displayed in their respected character form the array needed to be changed taken into

consideration that 2 different nodes of the same variable meant that there were 2 different instances of the same variable so it needed to have different naming like `C1 C2`

In [7]:
```
    @staticmethod
    def rename(self):
        t_mod = self.T
        for i in range(len(self.u)):
            self.u[i] = i
        t_mod = np.array(self.T)
        t_mod = np.append(self.u, t_mod, axis = 1)

        t_mod = t_mod.tolist()

        duplicate = len(self.Var)*[0] #Len = 6
        for i in range(2,len(t_mod)):
            duplicate[t_mod[i][1]] += 1


        index = len(t_mod)*[0] #Len = 6

        for i in range(0,len(self.Var)):
            for k in range(1, duplicate[i] + 1):
                for j in range(0, len(t_mod)):
                    if (i == t_mod[j][1] and index[j] == 0):
                        if(duplicate[i] > 1):
                            t_mod[j][0] = ''+str(self.Var[i])+''+str(k)+''
                            index[j] = 1
                            break
                        else:
                            t_mod[j][0] = self.Var[i]
                            index[j] = 1
                            break

        for i in range(2,len(t_mod[0])):
            for k in range(len(t_mod)):
                for j in range(len(t_mod)):
                    if( k == t_mod[j][i]):
                        t_mod[j][i] = t_mod[k][0]

        self.u = np.array(t_mod)[:,0].tolist()
        t_mod = np.array(t_mod)[:,1:4].tolist()
        self.T = t_mod
```

noding

We needed to make nodes of the variables and their connection to their respective low and high child to be mapped in a diagram

In [ ]:
```
    def noding(self, col, i):
        if(i==len(self.u)):
            return
        elif(i==0 or i==1):
            self.newu[i]=Node(self.newu[i])
```

```
            self.newu[i].lc=None
            self.newu[i].rc=None
            return self.noding(col, i+1)
        else:
            self.newu[i]=Node(self.newu[i])
            self.newu[i].lc = self.newu[self.u.index(self.T[i][col-1])]
            self.newu[i].rc = self.newu[self.u.index(self.T[i][col])]
            return self.noding(col, i+1)
```

create_graph

This function is used to create the graph using the nodes provided `noding`

In [ ]:
```
    def create_graph(self, node, graph, visited):
        graph.node(node.value)
        if node.lc is not None:
            if (node.value + node.lc.value) not in visited:
                graph.edge(node.value, node.lc.value, label='0',style='dotted')
                visited.add(node.value + node.lc.value)
            self.create_graph(node.lc, graph,visited)
        if node.rc is not None:
            if (node.value + node.rc.value) not in visited:
                graph.edge(node.value, node.rc.value, label='1')
                visited.add(node.value + node.rc.value)
            self.create_graph(node.rc, graph,visited)
```

satisfiability

This function incorporates a computational algorithm that systematically explores the entire domain of possible binary input combinations for a given boolean function, rigorously evaluating the output of each combination in accordance with the truth table, and meticulously counting the total number of instances in which the function's output evaluates to true.

In [ ]:
```
    def satisfiability(self):
        c=0
        num_combinations = 2 ** len(self.Var)
        for i in range(num_combinations):
            values_sat = {}

            for j, variable in enumerate(self.Var):
                truth_value = bool(i & (1 << j))
                #values_sat[variable] = truth_value
                values_sat={variable : truth_value}
            if eval(self.eq, values_sat):
                c+=1
                if(c==1):
                    print(f'Values for which expression is satisfied: {values_sat}'
                #break
        if(c==0):
```

```
            print("Expression is not satisfiable")
        print(f'There are {c} variations that satisfy the equation')
```

printROBDD

Simple print function to print the ROBDD

In [16]:
```
def printROBDD(ROBDD):
    print('============================')
    print('|  u  |  i  |   l  |   h  |')
    print('============================')
    n = ROBDD.nNodes
    for idx in range(0,n):
        node = ROBDD.T[idx]
        print('   '+str(ROBDD.u[idx])+'     '+str(node[0])+'     '+str(node[1])+'
        print('----------------------------')
```

Here We Start Our Testing

In [12]:
```
import numpy as np
import graphviz
import random
import colorama
from colorama import Fore

class Node:
    def __init__(self, value):
        self.value = value
        self.lc = None
        self.rc = None
class ROBDD:
    # Init the ROBDD with nVars = number of variables in the expression
    def __init__(self, eq):
        self.Var = []
        self.eq = eq
        self.Vardict = {}
        self.nNodes = 0
        self.x = []
        self.nVars = 0
        self.parser(self.eq, self.Var, 0)

        self.i=0
        while self.i<len(self.Var)-1:
            self.arrange(self)
            self.i += 1
        self.i = 0

        for item in range(self.nVars):
            self.x.append(0)

        self.initT(self)
        self.initH(self)
```

```
        self.build(0)

        self.u = np.zeros((len(self.T),1), dtype = int)
        self.rename(self)

        self.newu=[]
        self.parser(self.u, self.newu, 1)
        self.newu=['0']+['1']+self.newu
        self.noding(2, 0)

        self.visited=set()
        self.graph = graphviz.Digraph()
        self.create_graph(self.newu[(len(self.newu)-1)], self.graph,self.visited)
        self.graph.render('robdd'+str(random.randint(0,1000))+'', format='png', vie

        self.satisfiability()

      # Init the Table with 0,1 entries
    @staticmethod
    def initT(self):
        self.T = 2*[3*[None]]
        self.T[0] = [self.nVars + 1,-1,-1]
        self.T[1] = [self.nVars + 1,-2,-2]  # Using a different dummy entry -2, to
        self.nNodes = 2

        # Init the HashTable with hashes for 0 and 1
    @staticmethod
    def initH(self):
        self.H = 2 * [None]
        self.H[0] = self.hashH(self,0)
        self.H[1] = self.hashH(self,1)


    # Create Hashes for 0 and 1
    @staticmethod
    def hashH(self,nodeNum):
        return hash(str(self.T[nodeNum][0])+str(self.T[nodeNum][1])+str(self.T[node

        # Check if an entry exists in H
    def member(self, i, l, h):
        if hash(str(i) + str(l) + str(h)) in self.H:
            return True
        else:
            return False


    # Return the index of item with a given (i,l,h) from H
    def lookup(self,i,l,h):
        return self.H.index(hash(str(i) + str(l) + str(h)))


    # Add an entry to the table T, with values(i,l,h)
    def add(self,i,l,h):
        self.T.append([i,l,h])
        self.nNodes = self.nNodes + 1
        return self.nNodes - 1
```

```
    # Add an entry to H, with (i,l,h)
    def insert(self,i,l,h):
        self.H.append(hash(str(i) + str(l) + str(h)))


    # Add an entry to the T table.
    def make(self,i,l,h):
        if l == h:
            return l
        elif self.member(i,l,h):
            return self.lookup(i,l,h)
        else:
            u = self.add(i,l,h)
            self.insert(i,l,h)
            return u


    # Create the Table and the Inverse Hash Table.
    def build(self,i=0):
        if i>=self.nVars:
            return self.my_eva()
        else:
            self.x[i] = 0
            l = self.build(i+1)
            self.x[i] = 1
            h = self.build(i+1)
            return self.make(i,l,h)

    def my_eva(self):
        q=0
        for i in self.Var:

            self.Vardict[i]=self.x[q]
            q+=1

        if(eval(self.eq,self.Vardict) == True):
            return 1

        elif (eval(self.eq,self.Vardict) == False):
            return 0

        else: return eval(self.eq,self.Vardict)

    def parser(self, eq, Var, switch):
        for i in eq:
            if((i >= 'a') and (i <= 'z')):
                if(i in Var):
                    continue
                else: Var.append(i)
        if(switch == 0):
            eq = eq.replace("&", " and ").replace("+", " or ").replace("!", " not "
            self.eq = eq
            self.Var = Var
        else:
```

```
            self.u = eq
            self.newu = Var

    @staticmethod
    def arrange(self):
        i = 0
        while i < (len(self.Var)-1):
            if(self.Var[i]>self.Var[i+1]):
                temp=self.Var[i]
                self.Var[i]=self.Var[i+1]
                self.Var[i+1]=temp
            i = i + 1
        self.nVars = len(self.Var)

    @staticmethod
    def rename(self):
        t_mod = self.T
        for i in range(len(self.u)):
            self.u[i] = i
        t_mod = np.array(self.T)
        t_mod = np.append(self.u, t_mod, axis = 1)

        t_mod = t_mod.tolist()

        duplicate = len(self.Var)*[0] #Len = 6
        for i in range(2,len(t_mod)):
            duplicate[t_mod[i][1]] += 1


        index = len(t_mod)*[0] #Len = 6

        for i in range(0,len(self.Var)):
            for k in range(1, duplicate[i] + 1):
                for j in range(0, len(t_mod)):
                    if (i == t_mod[j][1] and index[j] == 0):
                        if(duplicate[i] > 1):
                            t_mod[j][0] = ''+str(self.Var[i])+''+str(k)+''
                            index[j] = 1
                            break
                        else:
                            t_mod[j][0] = self.Var[i]
                            index[j] = 1
                            break

        for i in range(2,len(t_mod[0])):
            for k in range(len(t_mod)):
                for j in range(len(t_mod)):
                    if( k == t_mod[j][i]):
                        t_mod[j][i] = t_mod[k][0]

        self.u = np.array(t_mod)[:,0].tolist()
        t_mod = np.array(t_mod)[:,1:4].tolist()
        self.T = t_mod

    def noding(self, col, i):
        if(i==len(self.u)):
```

```
            return
        elif(i==0 or i==1):
            self.newu[i]=Node(self.newu[i])
            self.newu[i].lc=None
            self.newu[i].rc=None
            return self.noding(col, i+1)
        else:
            self.newu[i]=Node(self.newu[i])
            self.newu[i].lc = self.newu[self.u.index(self.T[i][col-1])]
            self.newu[i].rc = self.newu[self.u.index(self.T[i][col])]
            return self.noding(col, i+1)

    def create_graph(self, node, graph, visited):
        graph.node(node.value)
        if node.lc is not None:
            if (node.value + node.lc.value) not in visited:
                graph.edge(node.value, node.lc.value, label='0',style='dotted')
                visited.add(node.value + node.lc.value)
            self.create_graph(node.lc, graph,visited)
        if node.rc is not None:
            if (node.value + node.rc.value) not in visited:
                graph.edge(node.value, node.rc.value, label='1')
                visited.add(node.value + node.rc.value)
            self.create_graph(node.rc, graph,visited)

    def satisfiability(self):
        c=0
        num_combinations = 2 ** len(self.Var)
        for i in range(num_combinations):
            values_sat = dict()

            for j, variable in enumerate(self.Var):
                truth_value = bool(i & (1 << j))
                values_sat[variable] = truth_value

            if eval(self.eq, values_sat):
                c+=1
                if(c==1):
                    del values_sat['__builtins__']
                    print(f'Values for which expression is satisfied: {values_sat}'
                #break
        if(c==0):
            print("Expression is not satisfiable")
        print(f'There are {c} variations that satisfy the equation')

def printROBDD(ROBDD):
    print('============================')
    print('|  u  |  i  |   l   |   h  |')
    print('============================')
    n = ROBDD.nNodes
    for idx in range(0,n):
        node = ROBDD.T[idx]
        print('   '+str(ROBDD.u[idx])+'     '+str(node[0])+'     '+str(node[1])+'
        print('----------------------------')
```

TEST

Here our Test is to take 2 unequivalent functions and check their

- `Satisfiability`
- `Equivalence`
- `ROBDD`

In [13]:
```
eq = 'a+b+c'
eq2 = '(!y&x)+(y&!x)'

ROBDD1 = ROBDD(eq)
printROBDD(ROBDD1)

ROBDD2 = ROBDD(eq2)
printROBDD(ROBDD2)

if (ROBDD1.T==ROBDD2.T):
    print(Fore.BLUE+'Both Functions are Equivalent')
else:
    print(Fore.RED+'Both Functions are not Equivalent')
```

```
Values for which expression is satisfied: {'a': True, 'b': False, 'c': False}
There are 7 variations that satisfy the equation
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
----------------------------
   1     4     -2     -2
----------------------------
   c     2     0     1
----------------------------
   b     1     c     1
----------------------------
   a     0     b     1
----------------------------
Values for which expression is satisfied: {'x': True, 'y': False}
There are 2 variations that satisfy the equation
============================
|  u  |  i  |   l  |   h  |
============================
   0     3     -1     -1
----------------------------
   1     3     -2     -2
----------------------------
   y1     1     0     1
----------------------------
   y2     1     1     0
----------------------------
   x     0     y1     y2
----------------------------
Both Functions are not Equivalent
```

Test 2

Here we test 2 equivalent cases

In [14]:
```
eq= '(!x & !y&z) + (!x&!y&!z) + (x&!y&!z) + (x&y&!z)'
eq2= '(x & ! z) + (! x & ! y)'

ROBDD1 = ROBDD(eq)
printROBDD(ROBDD1)

ROBDD2 = ROBDD(eq2)
printROBDD(ROBDD2)

if (ROBDD1.T==ROBDD2.T):
    print(Fore.BLUE+'Both Functions are Equivalent')
else:
    print(Fore.RED+'Both Functions are not Equivalent')
```

```
Values for which expression is satisfied: {'x': False, 'y': False, 'z': False}
There are 4 variations that satisfy the equation
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
----------------------------
   1     4     -2     -2
----------------------------
   y     1     1     0
----------------------------
   z     2     1     0
----------------------------
   x     0     y     z
----------------------------
Values for which expression is satisfied: {'x': False, 'y': False, 'z': False}
There are 4 variations that satisfy the equation
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
----------------------------
   1     4     -2     -2
----------------------------
   y     1     1     0
----------------------------
   z     2     1     0
----------------------------
   x     0     y     z
----------------------------
Both Functions are Equivalent
```

Test 3

Here You `(The User)` are Prompted to enter 2 boolean functions in order to check them as above

In [ ]:
```
eq = input('1st Boolean Function ')
eq2 =input('2nd Boolean Function ')

ROBDD1 = ROBDD(eq)
printROBDD(ROBDD1)

ROBDD2 = ROBDD(eq2)
printROBDD(ROBDD2)

if (ROBDD1.T==ROBDD2.T):
    print(Fore.BLUE+'Both Functions are Equivalent')
else:
    print(Fore.RED+'Both Functions are not Equivalent')
```