

# **RIPPLE DOWN RULES**

Disusun untuk memenuhi tugas  
Mata Kuliah IF4070 Representasi Pengetahuan dan Penalaran



## **Disusun oleh:**

Fazel Ginanda	13521098
Akhmad Setiawan	13521164
Satria Octavianus Nababan	13521168

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
NOVEMBER 2024**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>PENDAHULUAN.....</b>	<b>3</b>
<b>PENJELASAN KODE PROGRAM.....</b>	<b>4</b>
1. Library.....	4
2. Kelas Rule.....	4
3. Kelas Tree.....	6
4. Fungsi Untuk Save dan Load File Pohon RDR Dengan Format .pkl.....	7
5. Fungsi Untuk Save Visualisasi Pohon RDR Dengan Format .png.....	8
<b>CARA KERJA PROGRAM.....</b>	<b>9</b>
1. Pilih Cara Penggunaan RDR.....	9
2. Membuat Pohon Baru.....	9
3. Menggunakan Pohon Yang Sudah Disimpan.....	15
<b>LAMPIRAN.....</b>	<b>16</b>

## PENDAHULUAN

*Ripple Down Rules* (RDR) adalah suatu metode dalam sistem berbasis pengetahuan yang mengadopsi pendekatan penalaran dengan pengecualian atau penalaran berbasis aturan. Dalam RDR, pohon pengetahuan dibangun secara iteratif, dimulai dari kesimpulan *default* dan dikembangkan berdasarkan input data yang diberikan oleh pakar. Setiap kali pakar tidak setuju dengan kesimpulan yang diberikan oleh RDR terhadap suatu data baru, maka aturan baru ditambahkan ke dalam pohon untuk menangani kasus tersebut. Metode ini sangat berguna dalam mengelola sistem berbasis pengetahuan yang dinamis, terutama dalam sistem inferensi atau sistem diagnosis berbasis aturan.

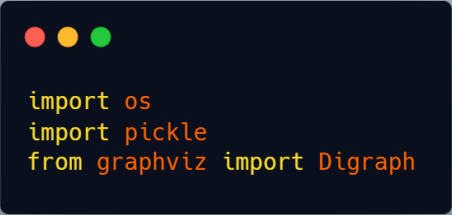
Program ini mengimplementasikan metode RDR dengan beberapa fitur sebagai berikut.

1. Membuat pohon basis pengetahuan yang memiliki struktur pohon biner dengan cabang kanan sebagai TRUE dan cabang kiri sebagai FALSE
2. Membuat pohon pengetahuan dari awal basis pengetahuan yang kosong atau memuat pohon yang sudah ada dalam bentuk file pickle (.pkl)
3. Menyimpan pohon basis pengetahuan yang telah dimodifikasi dengan penambahan aturan baru dalam format file pickle (.pkl)
4. Menghasilkan dan menyimpan visualisasi pohon pengetahuan dalam format gambar dengan ekstensi .png
5. Memungkinkan pakar untuk menambahkan data baru, memberikan kesimpulan berdasarkan data yang ditambahkan, dan memperbarui pohon pengetahuan dengan aturan baru jika kesimpulan tidak sesuai dengan pendapat pakar

## PENJELASAN KODE PROGRAM

Program ini ditulis dalam bentuk Jupyter Notebook dengan ekstensi .ipynb. Adapun komponen program yang dibentuk:

### 1. Library



```
import os
import pickle
from graphviz import Digraph
```

Library `os` digunakan untuk mengelola file dan direktori, memastikan tempat penyimpanan yang tepat untuk pohon RDR dan visualisasinya. Library `pickle` digunakan untuk menyimpan dan memuat pohon RDR yang berisi *rules* sehingga pengguna tidak perlu membangun pohon pengetahuan dari awal setiap kali menjalankan program. Library `graphviz.Digraph` digunakan untuk memvisualisasikan struktur pohon pengetahuan secara grafis dan memudahkan pengguna untuk memahami tentang bagaimana aturan-aturan dalam sistem RDR bekerja dan berhubungan satu sama lain.

### 2. Kelas Rule



```
class Rule:
    def __init__(self, condition, conclusion, case):
        self.condition = condition
        self.conclusion = conclusion
        self.case = case

    def isSatisfied(self, data):
        if isinstance(self.condition, bool):
            return self.condition
        elif isinstance(self.condition, list):
            return set(self.condition).issubset(set(data))
        else:
            return False

    def print(self):
        print(str(self.condition) + " -> " + self.conclusion + " | Kasus Kunci: " + str(self.case))
```

Kelas Rule digunakan untuk merepresentasikan sebuah aturan dalam pohon pengetahuan. Setiap aturan memiliki tiga komponen utama:

1. *Condition*: Kondisi yang harus dipenuhi agar aturan ini dapat diterapkan. Kondisi ini bisa berupa nilai boolean True atau daftar kondisi. Kondisi True digunakan untuk rule 0 dan *rule* yang dibuat sebagai revisi atas *rule* sebelumnya. Dengan kondisi True, semua kasus yang mencapai simpul *rule* itu selalu memenuhi rule tersebut. Sementara itu, daftar kondisi direpresentasikan dengan struktur data *array of string*.
2. *Conclusion*: Kesimpulan yang diambil jika kondisi pada aturan dipenuhi. Kesimpulan terbatas pada satu jenis klasifikasi. Oleh karena itu, pada sistem ini kesimpulan direpresentasikan sebagai sebuah string.
3. *Case*: Kasus atau data yang mendasari pembuatan aturan ini. Kasus direpresentasikan dengan struktur data *array of string*.

Metode *isSatisfied(data)* digunakan untuk memeriksa apakah semua kondisi dalam sebuah *rule* dipenuhi oleh data yang diberikan. Cara kerja metode *isSatisfied(data)* adalah memeriksa apakah sebuah *rule* merupakan himpunan bagian dari data yang diberikan. Jika sebuah *rule* merupakan himpunan bagian dari data yang diberikan, data tersebut mengandung fakta-fakta yang menjadi kondisi dari *rule* tersebut. Dengan demikian, *rule* tersebut dapat dipenuhi oleh data tersebut.

Metode *print()* digunakan untuk menampilkan informasi tentang aturan ini. Metode ini menampilkan kondisi, kesimpulan, serta kasus yang menyebabkan aturan tersebut dibentuk.

### 3. Kelas Tree

```
class Tree:
    def __init__(self, root):
        self.root = root
        self.left = None
        self.right = None

    def search(self, data, tree):
        if self.root:
            if self.root.isSatisfied(data):
                tree = self
                if self.right:
                    return self.right.search(data, tree)
            else:
                if self.left:
                    return self.left.search(data, tree)
        return tree

    def insertRule(self, rule):
        if self.root:
            if not self.right:
                self.right = Tree(rule)
            else:
                currentTree = self.right
                while currentTree.left:
                    currentTree = currentTree.left
                currentTree.left = Tree(rule)
        return self

    def printPreorder(self):
        self.root.print()
        if self.left:
            self.left.printPreorder()
        if self.right:
            self.right.printPreorder()
```

Kelas Tree digunakan untuk merepresentasikan pohon pengetahuan dalam bentuk pohon biner. Setiap simpul di dalam pohon adalah sebuah objek *rule*. Pohon ini memiliki dua cabang:

1. Cabang kiri: Mewakili kondisi False.
2. Cabang kanan: Mewakili kondisi True.

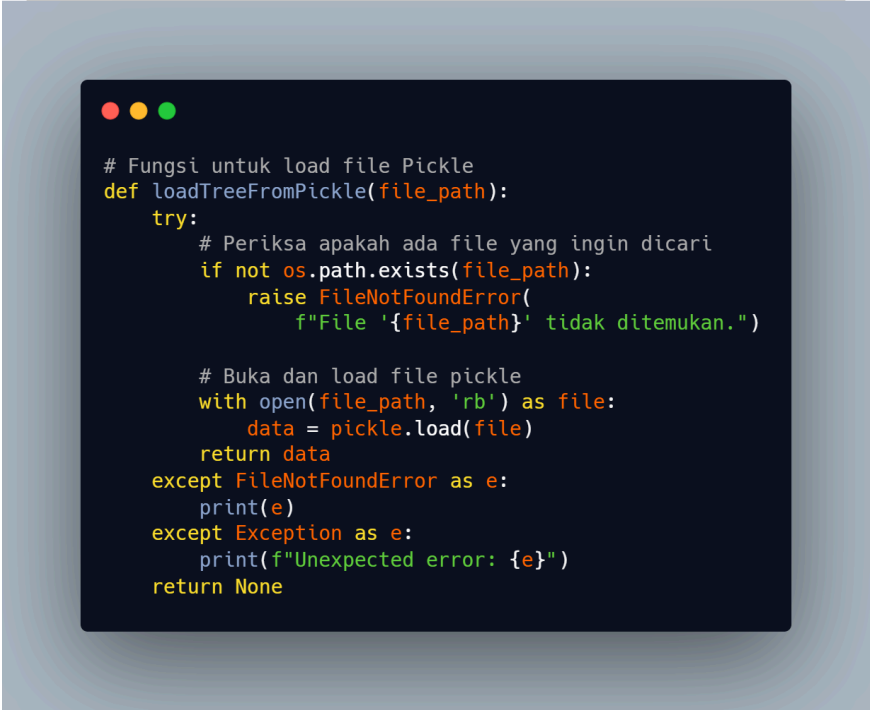
Metode *search(data, tree)* digunakan untuk mencari kesimpulan berdasarkan data yang diberikan. Cara kerja metode *search* adalah sebagai berikut. Metode ini menerima dua parameter, yaitu *data* dan *tree*. Parameter *data* merupakan data baru yang akan ditentukan kesimpulannya. Sementara itu, parameter *tree* merupakan inisialisasi *tree* sebagai *return value* dari metode tersebut. Pertama, diperiksa apakah pohon RDR saat ini memiliki simpul

akar. Jika tidak, maka dikembalikan *tree* yang diberikan pada parameter. Jika pohon memiliki simpul akar, maka akan dilakukan pencarian lebih lanjut. Selanjutnya, diperiksa apakah *data* memenuhi rule yang terdapat pada simpul tersebut. Jika rule pada simpul tersebut dipenuhi, maka value *tree* diubah menjadi simpul tersebut. Hal ini menggunakan konsep dasar bahwa setiap simpul pada suatu pohon juga merupakan pohon. Selain itu, perubahan value *tree* tersebut bertujuan untuk menandai bahwa simpul tersebut merupakan simpul dengan aturan terakhir yang dipenuhi oleh *data*. Selanjutnya, apabila simpul tersebut memiliki anak kanan, maka dilakukan pemanggilan fungsi *search* secara rekursif dengan parameter *data* yang sama dan *tree* yang sudah diubah untuk anak kanan tersebut. Akan tetapi, hal yang berbeda terjadi apabila rule pada sebuah simpul tidak dipenuhi oleh *data*. Value dari *tree* tidak diubah. Hal itu disebabkan oleh rule yang terakhir dipenuhi tidak berubah. Pada kasus ini, program memeriksa apakah simpul tersebut memiliki anak kiri. Jika simpul tersebut memiliki anak kiri, maka akan dilakukan pemanggilan fungsi *search* secara rekursif dengan parameter *data* dan *tree* yang sama untuk anak kiri tersebut. Return value dari metode ini adalah objek Tree. Alasan penggunaan Tree sebagai *return value* dari metode ini adalah agar objek Tree tersebut dapat dimanfaatkan untuk melakukan operasi *insert* atau manipulasi pohon ketika pakar tidak setuju dan ingin menambahkan aturan baru.

Metode *insertRule(rule)* digunakan untuk menambahkan aturan baru ke dalam pohon jika diperlukan. Cara kerja dari metode ini adalah sebagai berikut. Metode ini menerima satu parameter, yaitu sebuah objek Rule. Metode *insertRule* dipanggil oleh objek Tree yang merupakan simpul rule yang aktif terakhir. Objek Tree tersebut didapatkan melalui pemanggilan metode *search* yang dilakukan ketika inferensi. Pertama, diperiksa apakah pohon tersebut memiliki simpul. Jika tidak, maka tidak dilakukan perubahan terhadap pohon tersebut. Jika pohon tersebut memiliki simpul, maka dilakukan proses lebih lanjut. Pertama, diperiksa apakah pohon tersebut memiliki anak kanan. Jika tidak memiliki anak kanan, maka akan langsung dibentuk simpul baru sebagai anak kanan dari pohon tersebut. Akan tetapi, jika pohon tersebut memiliki anak kanan, maka pencarian dilanjutkan dengan menelusuri jalur melalui anak kanan tersebut. Anak kanan dari simpul tersebut merupakan aturan pertama yang dievaluasi FALSE untuk data tersebut. Oleh karena itu, aturan baru akan dibentuk sebagai anak kiri dari anak kanan tersebut. Apabila telah terdapat anak kiri, maka dilakukan penelusuran terus-menerus hingga mencapai anak kiri terakhir yang menjadi simpul daun dari pohon RDR. Pada simpul daun itu lah, dibentuk anak kiri sebagai simpul baru yang berisi *rule* yang diberikan pada parameter.

Metode *printPreorder()* digunakan untuk menampilkan seluruh pohon pengetahuan dalam urutan *pre-order*, yaitu mengunjungi root terlebih dahulu, lalu anak kiri, kemudian anak kanan.

#### 4. Fungsi Untuk Save dan Load File Pohon RDR Dengan Format .pkl



```
# Fungsi untuk load file Pickle
def loadTreeFromPickle(file_path):
    try:
        # Periksa apakah ada file yang ingin dicari
        if not os.path.exists(file_path):
            raise FileNotFoundError(
                f"File '{file_path}' tidak ditemukan.")

        # Buka dan load file pickle
        with open(file_path, 'rb') as file:
            data = pickle.load(file)
        return data
    except FileNotFoundError as e:
        print(e)
    except Exception as e:
        print(f"Unexpected error: {e}")
    return None
```

Fungsi *loadTreeFromPickle(file\_path)* digunakan untuk memuat pohon pengetahuan yang telah disimpan dalam format file pickle. Jika file tidak ditemukan, program akan menampilkan pesan error.



## 5. Fungsi Untuk Save Visualisasi Pohon RDR Dengan Format .png

```
# Fungsi untuk mengubah RDR tree menjadi graph menggunakan library Digraph agar dapat divisualisasikan
def addNodesAndEdges(tree, graph, parent_id = None, position = ""):
    if tree is None:
        return

    # Buat identifier unik untuk sebuah simpul
    nodeId = id(tree)

    # Buat label untuk kondisi dari sebuah rule
    if isinstance(tree.root.condition, bool) and tree.root.condition is True:
        conditionText = "Kondisi: True"
    else:
        conditionText = f"Kondisi: {'', '.join(tree.root.condition)}"

    # Buat label untuk kesimpulan dari sebuah rule
    conclusionText = f"Kesimpulan: {tree.root.conclusion}"

    # Buat label untuk kasus kunci pada sebuah rule
    if not tree.root.case:
        caseText = "Kasus Kunci: []"
    else:
        caseText = f"Kasus Kunci: {'', '.join(tree.root.case)}"

    ruleLabel = f"{conditionText}\n{conclusionText}\n{caseText}"

    # Tambahkan node yang baru dibuat ke dalam graph
    graph.node(str(nodeId), ruleLabel)

    # Jika ada parent, maka tambahkan edge
    if parent_id is not None:
        if position == "left":
            graph.edge(str(parent_id), str(nodeId), xlabel = "Left")
        elif position == "right":
            graph.edge(str(parent_id), str(nodeId), xlabel = "Right")

    # Panggil fungsi secara rekursif untuk anak kiri dan anak kanan
    addNodesAndEdges(tree.left, graph, nodeId, position = "left")
    addNodesAndEdges(tree.right, graph, nodeId, position = "right")
```

```
# Fungsi to memvisualisasikan Ripple Down Rule (RDR) tree
def visualizeRdrTree(RdrTree, treeImagePath):
    dot = Digraph(comment="Ripple Down Rule Tree")
    dot.attr(rankdir = "TB", nodesep = "0.5", ranksep = "0.5")
    addNodesAndEdges(RdrTree, dot)
    dot.render(treeImagePath, format='png', cleanup=True)
```

Fungsi *visualizeRdrTree(RdrTree, treeImagePath)* digunakan untuk menghasilkan visualisasi pohon pengetahuan dalam format gambar (PNG) menggunakan library Graphviz. Fungsi ini memanggil fungsi *addNodesAndEdges()* untuk menambahkan simpul dan sisi pohon ke dalam objek Digraph.

## CARA KERJA PROGRAM

Berikut adalah langkah-langkah penggunaan program.

### 1. Pilih Cara Penggunaan RDR

```
... Pilih cara penggunaan RDR
  1. Membuat tree RDR baru
  2. Menggunakan RDR yang sudah disimpan
Jawaban: 
```

Program akan meminta pengguna untuk memilih antara membuat pohon baru atau menggunakan pohon yang sudah ada dan tersimpan sebelumnya.

### 2. Membuat Pohon Baru

Ketika pengguna memilih untuk membuat pohon baru, pengguna akan diminta memasukkan kesimpulan *default*. Kesimpulan ini dapat disesuaikan dengan domain keahlian pakar sebagai pengguna.

```
⇒ Pilih cara penggunaan RDR
  1. Membuat tree RDR baru
  2. Menggunakan RDR yang sudah disimpan
Jawaban: 1

Buat Tree RDR Baru
Masukkan kesimpulan default: 
```

Misalnya dalam domain kesehatan, kesimpulan *default*-nya adalah sehat.

```
... Pilih cara penggunaan RDR
  1. Membuat tree RDR baru
  2. Menggunakan RDR yang sudah disimpan
Jawaban: 1

Buat Tree RDR Baru
Masukkan kesimpulan default: sehat
Tree RDR
True -> sehat | Kasus Kunci: []
```

Setelah itu, pengguna dapat memasukkan data baru. Pengguna dapat memasukkan data yang berisi sekumpulan fakta yang dipisahkan oleh tanda koma. Misalkan pengguna memasukkan

data demam. Dengan pohon RDR saat ini yang hanya berisi Rule 0 dengan kesimpulan default, maka program akan mengembalikan kesimpulan default, yakni sehat.

```
Apakah Anda ingin memasukkan data?
1. Ya
2. Tidak
Jawaban: 1

Masukkan data: demam
Kesimpulan: sehat
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 
```

Setelah sistem memberikan jawaban atau kesimpulan, pengguna dapat menolak kesimpulan itu dan membuat kesimpulan yang baru. Misalkan pengguna pakar ingin menambahkan aturan “jika demam, maka kemungkinan terkena flu”.

```
Masukkan data: demam
Kesimpulan: sehat
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 2

Manipulasi pohon RDR
Data saat ini: ['demam']
Kasus kunci yang terakhir dipenuhi: []
Selisih antara data saat ini dengan kasus kunci terakhir: ['demam']

Pilih kondisi untuk rule baru. Kondisi harus merupakan bagian dari selisih yang ditampilkan sebelumnya.
Masukkan kondisi untuk rule baru: demam

Masukkan kesimpulan baru: 
```

Aturan baru tersebut dibentuk dengan cara sebagai berikut. Sistem akan menentukan kondisi yang diperbolehkan untuk aturan yang akan dibentuk tersebut. Kondisi merupakan selisih antara data baru dengan data pendukung dari aturan yang aktif terakhir kali. Kemudian, sistem menampilkan fakta apa saja dari data baru yang dapat dijadikan kondisi untuk aturan baru. Setelah itu, sistem akan meminta pengguna untuk memasukkan kondisi yang dipilih dan kesimpulan untuk aturan baru tersebut. Pada kasus di atas, pengguna memilih demam untuk dijadikan kondisi dan kemungkinan flu untuk dijadikan kesimpulan dari aturan baru yang dibentuk. Dengan demikian, aturan baru dibentuk pada pohon RDR, yaitu jika demam, maka kemungkinan flu dengan kasus kunci demam.

```
Masukkan kesimpulan baru: kemungkinan flu
Rule baru: ['demam'] -> kemungkinan flu | Kasus Kunci: ['demam']

Tree RDR setelah dimanipulasi
True -> sehat | Kasus Kunci: []
['demam'] -> kemungkinan flu | Kasus Kunci: ['demam']
```

Pembuatan aturan baru ini dapat terus dikembangkan oleh pengguna sebagai pakar berdasarkan pendapat dalam domain keahliannya. Misalkan ditambahkan aturan baru sebagai pengecualian setelah demam, yakni gatal, maka kesimpulannya bukan kemungkinan flu, melainkan alergi.

```
Apakah Anda ingin memasukkan data?
1. Ya
2. Tidak
Jawaban: 1

Masukkan data: demam, gatal
Kesimpulan: kemungkinan flu
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 
```

Pengetahuan yang terdapat pada RDR saat ini belum memiliki aturan untuk pengecualian pada kasus gatal, sehingga sistem masih mengembalikan kesimpulan saat demam, yakni kemungkinan terkena flu. Pengguna kemudian dapat menolak kesimpulan ini dan menambahkan pengecualian pada kasus gatal dengan kesimpulan alergi.

```
Manipulasi pohon RDR
Data saat ini: ['demam', 'gatal']
Kasus kunci yang terakhir dipenuhi: ['demam']
Selisih antara data saat ini dengan kasus kunci terakhir: ['gatal']

Pilih kondisi untuk rule baru. Kondisi harus merupakan bagian dari selisih yang ditampilkan sebelumnya.
Masukkan kondisi untuk rule baru: gatal

Masukkan kesimpulan baru: alergi
Rule baru: ['gatal'] -> alergi | Kasus Kunci: ['demam', 'gatal']

Tree RDR setelah dimanipulasi
True -> sehat | Kasus Kunci: []
['demam'] -> kemungkinan flu | Kasus Kunci: ['demam']
['gatal'] -> alergi | Kasus Kunci: ['demam', 'gatal']
```

Setelah aturan baru terbentuk, akan diperoleh kesimpulan yang berbeda antara input data **demam** dengan input data **demam dan gatal** (dipisahkan oleh tanda koma menjadi **demam, gatal**)

```
Masukkan data: demam
Kesimpulan: kemungkinan flu
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 1
```

Dari gambar di atas, dapat dilihat jika hanya demam, maka kesimpulannya kemungkinan terkena flu.

```
Masukkan data: demam, gatal
Kesimpulan: alergi
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 1
```

Sementara itu, jika demam dan gatal, maka kesimpulan penyakitnya adalah reaksi alergi.

Selanjutnya, misalkan data yang dimasukkan adalah demam dan pusing. Pengetahuan yang terdapat pada RDR saat ini belum memiliki aturan untuk pengecualian pada kasus pusing, sehingga aturan yang dipenuhi hanya aturan “jika demam, kemungkinan flu”. Oleh karena itu, sistem mengembalikan kesimpulan kemungkinan flu.

```
Apakah Anda ingin memasukkan data?
1. Ya
2. Tidak
Jawaban: 1

Masukkan data: demam, pusing
Kesimpulan: kemungkinan flu
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 2
```

Kesimpulan ini dapat diubah dengan pengecualian baru ketika kasus demam dan pusing, misalkan kesimpulannya adalah migraine.

```

Manipulasi pohon RDR
Data saat ini: ['demam', 'pusing']
Kasus kunci yang terakhir dipenuhi: ['demam']
Selisih antara data saat ini dengan kasus kunci terakhir: ['pusing']

Pilih kondisi untuk rule baru. Kondisi harus merupakan bagian dari selisih yang ditampilkan sebelumnya.
Masukkan kondisi untuk rule baru: pusing

Masukkan kesimpulan baru: migraine
Rule baru: ['pusing'] -> migraine | Kasus Kunci: ['demam', 'pusing']

Tree RDR setelah dimanipulasi
True -> sehat | Kasus Kunci: []
['demam'] -> kemungkinan flu | Kasus Kunci: ['demam']
['gatal'] -> alergi | Kasus Kunci: ['demam', 'gatal']
['pusing'] -> migraine | Kasus Kunci: ['demam', 'pusing']

```

Pada kasus seperti ini, seolah-olah simpul pusing adalah anak dari simpul demam. Namun sebenarnya, proses inferensi akan tetap melalui simpul yang telah terbentuk sebelumnya, yakni gatal, namun dalam kondisi FALSE. Oleh karena itu, simpul pusing merupakan anak kiri dari simpul gatal.

Untuk memperlihatkan perbedaannya, akan ditambahkan pengecualian baru saat kasus demam, gatal, dan kulit bernanah, yaitu kesimpulan cacar.

```

Masukkan data: demam, gatal, kulit bernanah
Kesimpulan: alergi
Apakah Anda setuju dengan kesimpulan yang diberikan?
1. Setuju
2. Tidak Setuju
Jawaban: 2

Manipulasi pohon RDR
Data saat ini: ['demam', 'gatal', 'kulit bernanah']
Kasus kunci yang terakhir dipenuhi: ['demam', 'gatal']
Selisih antara data saat ini dengan kasus kunci terakhir: ['kulit bernanah']

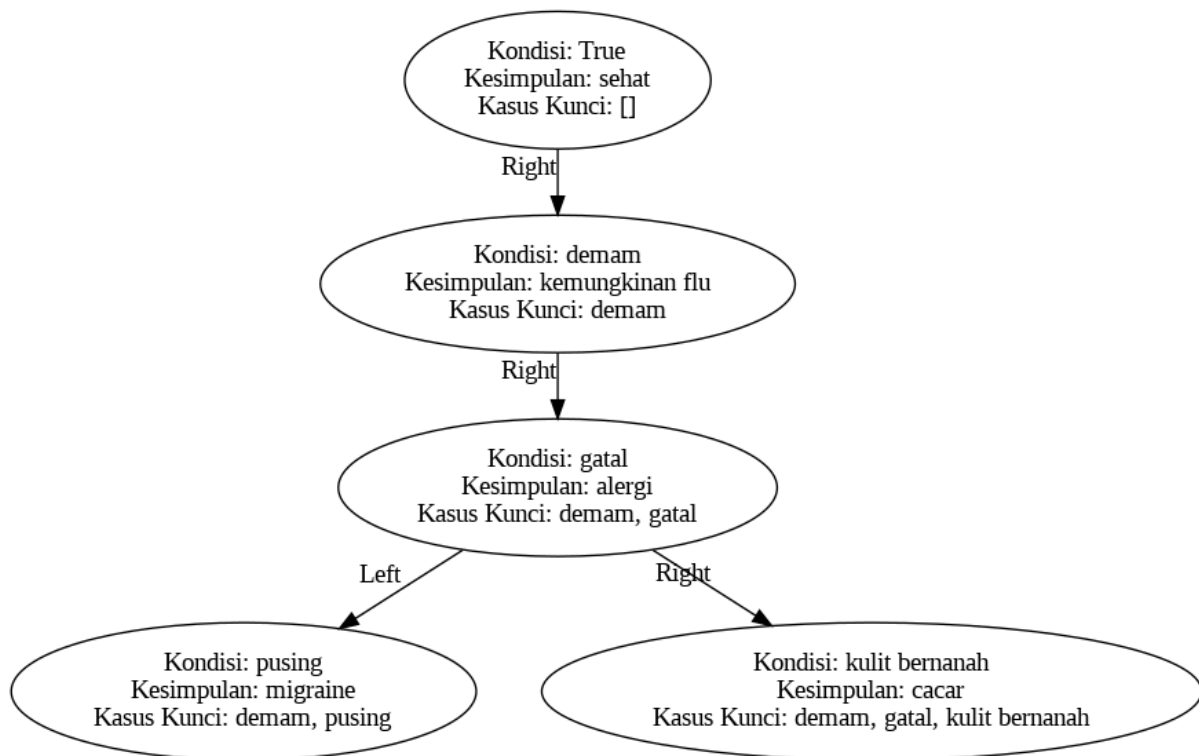
Pilih kondisi untuk rule baru. Kondisi harus merupakan bagian dari selisih yang ditampilkan sebelumnya.
Masukkan kondisi untuk rule baru: kulit bernanah

Masukkan kesimpulan baru: cacar
Rule baru: ['kulit bernanah'] -> cacar | Kasus Kunci: ['demam', 'gatal', 'kulit bernanah']

Tree RDR setelah dimanipulasi
True -> sehat | Kasus Kunci: []
['demam'] -> kemungkinan flu | Kasus Kunci: ['demam']
['gatal'] -> alergi | Kasus Kunci: ['demam', 'gatal']
['pusing'] -> migraine | Kasus Kunci: ['demam', 'pusing']
['kulit bernanah'] -> cacar | Kasus Kunci: ['demam', 'gatal', 'kulit bernanah']

```

Setelah serangkaian penambahan data baru serta aturan baru tersebut, pohon yang terbentuk akan menjadi seperti berikut.



Perhatikan bahwa ketika gatal tidak dimasukkan ke dalam data *input*, maka data *input* tersebut akan dievaluasi sebagai **False** pada simpul gatal dan cabang pohon dibentuk ke **kiri**. Sementara itu, jika data input dievaluasi sebagai **True**, cabang pohon akan dibentuk ke **kanan**. Pengecualian aturan-aturan ini dapat terus dikembangkan secara perlahan dan bertahap sehingga terbentuk sebuah basis pengetahuan besar berbentuk pohon biner.

Pohon ini dapat disimpan dalam bentuk .pkl dalam path {current\_directory}/Tree Pickles/{nama\_file} untuk nanti digunakan kembali ketika menjalankan program ini.

```
Apakah Anda ingin menyimpan tree yang sudah digunakan?
1. Ya
2. Tidak
Jawaban: 1

File akan disimpan pada direktori /content/Tree Pickles
Masukkan nama file tree tanpa extension: medical
File tree telah berhasil disimpan
```

Visualisasi pohon juga dapat disimpan dalam format .png di {current\_directory}/Tree Visualizations/{nama\_file}.

Apakah Anda ingin menyimpan visualisasi tree yang sudah digunakan?

1. Ya
2. Tidak

Jawaban: 1

File visualisasi akan disimpan pada direktori /content/Tree Visualizations

Masukkan nama file visualisasi tree tanpa extension: demo2

File visualisasi telah berhasil disimpan

### 3. Menggunakan Pohon yang Sudah Disimpan

Program akan menampilkan daftar pohon yang sudah ada dan memungkinkan pengguna untuk memilih salah satu file pickle yang tersedia di folder {current\_directory}/Tree Pickles/

Pilih cara penggunaan RDR

1. Membuat tree RDR baru
2. Menggunakan RDR yang sudah disimpan

Jawaban: 2

Daftar nama file tree yang disimpan:

1. test.pkl
2. demo.pkl

Masukkan nomor file tree yang ingin digunakan: 2

Tree RDR

True -> sehat | Kasus Kunci: []

['demam'] -> kemungkinan flu | Kasus Kunci: ['demam']

['gatal'] -> alergi | Kasus Kunci: ['demam', 'gatal']

Apakah Anda ingin memasukkan data?

1. Ya
2. Tidak

Jawaban:

Aturan-aturan yang telah tersimpan sebelumnya dapat langsung digunakan untuk melakukan inferensi terhadap data baru dan mengembangkan pohon RDR lebih lanjut.



## LAMPIRAN

### **Repository Github:**

<https://github.com/fazelginanda/RDR/tree/main>

### **Google Colab:**

[https://colab.research.google.com/drive/1Vwt65QFkOCKOfXav\\_7XcbbyKAQx6XQvn?usp=sharing](https://colab.research.google.com/drive/1Vwt65QFkOCKOfXav_7XcbbyKAQx6XQvn?usp=sharing)