Indian Institute of Information Technology and Management – Kerala (IIITM-K)

M.Sc. in Computer Science

Data Structures and Algorithms

(Mid-Term Exam 1)

Submitted By:

Ravi Prakash

M.Sc. Computer Science (Cyber Security)

Sem. I

Dated on: December 17th, 2020

Questions

Following 3 questions were provided in the question paper:

A. *Ramu wants to store 1 million images, 2 million audio files, and 1 billion documents in a structured way on his computer. His computer has no limitations on memory, nor processor speed. His computer usually achieves processing frequency beyond 1 billion hertz, and uses 50 billion terabytes of RAM. The hard drive capacity is infinite, with average use of 500 trillion terabytes. Size of his computer is less than the size of the human brain. What is the best data structure for Ramu to store the files, and how can he efficiently store and retrieve the files at the highest possible speeds. Give your answer with explanations including relevant python codes.*
(3 marks)

B. Write a python program making use of any data structures to automatically detect, identify, rearrange and correct the words in jumbled sentences.
Such as:
1. ORF _ MECO FEEFOC
2. effect ew ni did uchm sales last ton eary pimrovement eunron si oodg

Explain your solution in detail.
(3 marks)

C. *Capture a photo of your neighborhood, and label each of the objects. Create a tree structure with each node representing an object and link weights showing the similarity between them. What will be the best way to implement this? Which data structure will you use and why? Show with an example code, providing detailed explanations of the pros and cons of your approach.*
(4 marks)

Before the Solutions

Before the solution, I would like to inform that:

1. All questions are consist of following main sections -
    a. Prerequisites
    b. Problem Understanding
    c. Storage Model Description
    d. Pros & Cons of the Solution
    e. Python Code

2. At least one key algorithm is written in every solution with its sample execution, as well.

3. Except ont image (*trie* i.e. taken from google search results), all the model images are self created by me.

4. A running instance of provided codes can be accessed here (notebook link below):

    https://colab.research.google.com/drive/1UGP4adW7RwfD-3S-_1vIo534C7VVa_VN?usp=sharing

    (this notebook contains exactly the same code that is submitted in this report)

Solutions

Solution A:

"Ramu wants to store 1 million images, 2 million audio files, and 1 billion documents in a structured way on his computer. His computer has no limitations on memory, nor processor speed. His computer usually achieves processing frequency beyond 1 billion hertz, and uses 50 billion terabytes of RAM. The hard drive capacity is infinite, with average use of 500 trillion terabytes. Size of his computer is less than the size of the human brain. What is the best data structure for Ramu to store the files, and how can he efficiently store and retrieve the files at the highest possible speeds. Give your answer with explanations including relevant python codes."

--

Prerequisite

Before going into the solution, we need to keep following in the mind:

1. *Binary Tree*
2. *Max Heap*
3. *Stack*

Binary Tree:

- Maximum possible number of children for every node: **2**

Max-Heap:

- Can be implement using a binary tree
- The node with the maximum value is the root of the tree
- Nodes with lowest keys in the heap are the leaves.

Stack:

- It's a linear data structure.
- Elements can be added at the top most position, only.
- Elements can be deleted from the top most position, only.
- Insertion operation of a new element is specified as: **push**
- Deletion operation of a new element is specified as: **pop**

About the files that Ramu has:

1. **Document Files:** Mainly document files are used to store the textual data. But, it is also possible to store some media files along with the text data.
   Some common formats of the document files are: *doc, odt, rtf etc..*

2. **Image Files:** These are the compressed files that organize and store digital images. These images are generally captured front the cameras and represent a sequence of the colours or rather essay pixels
   Some common formats of the image files are: *jpg, png, svg, ppf* etc..

3. **Audio Files:** Audio files mainly store the sampled audio signals in a compressed format. These samples of analog audio signal are collected at a fixed interval of time.
   Some common formats of the audio files are: *mp3, mpeg, wav* etc..

Storage of Data

Image and **Audio** files (along with Videos, too) are collectively called the **media files**. Hence, initially we can classify all the available files in two parts:

a. Documents
b. Media

Here, I introduce the initial Tree (data structure) to store the available data:

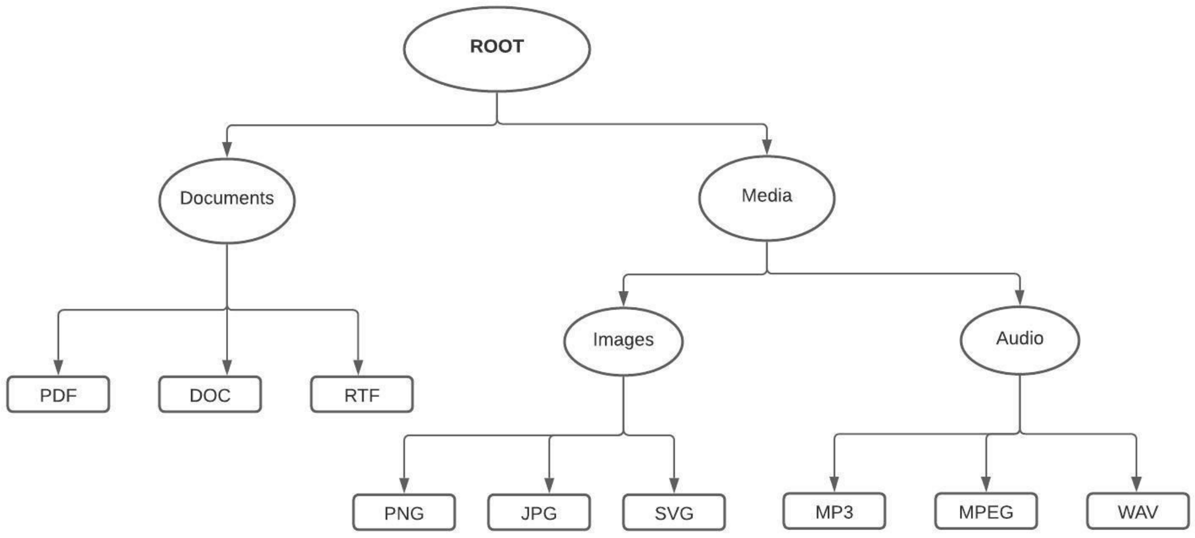

Fig. 1.1

(here the leaves can be far more than 3, depending upon the data provided)

Now, it gives an idea that all the leaves can hold identical types of files.
That means all the images with Jpg format will be stored under "jpg". Similarly, doc files will be stored under node "doc" of the **document** subtree.

Now, the challenge is how Ramu should store any separate kind of data item in each node. So, here I introduce the usage of Max-Heap.

Let, the key for our max heap be the time, when new data is added, such that the recent data must have the highest key.
Now, one can imagine each of the above leaves referring to one max-heap, where all the nodes represent a similar type of data stored at different intervals of time. Due to being a max-heap, the latest data will always lie on the top or on the root of each max-heap.

For an instance, we can imagine **jpg** node of Media subtree to be like:

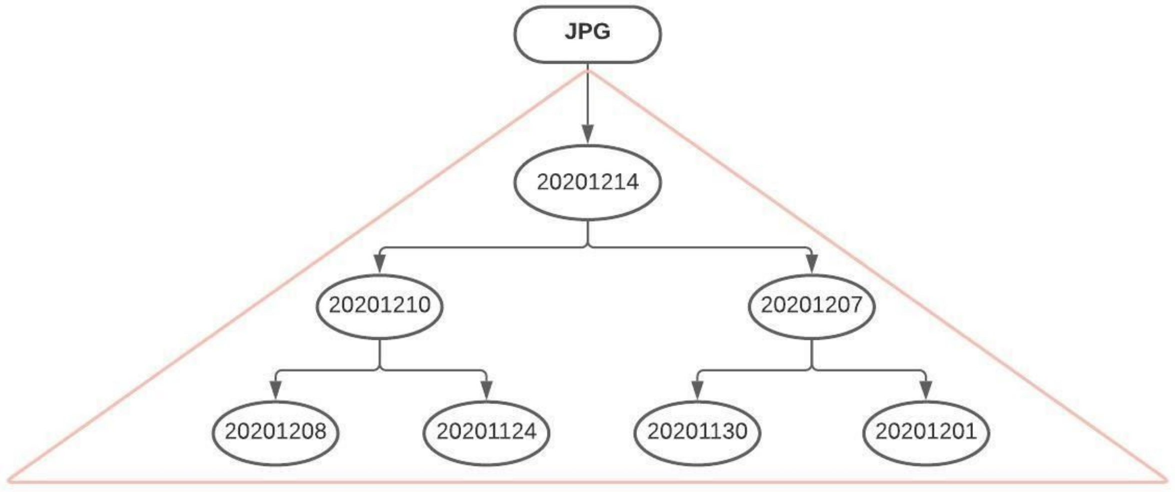

Fig. 1.2

Here, the part enclosed in the triangle is the **Max-Heap**. Every node of this heap represents an image of **.JPG** format. The key of these nodes are the date, when they were added in **YYYYMMDD** format. For a better we will use a <u>**Date-Time combination for the keys**</u> in actual implementation ***(YYYYMMDDhhmmss)***.

For an instance, **20201214** means that the node (or rather say - file) was added on **December 14, 2020**

And hence, the new model of the tree can be considered as described below:

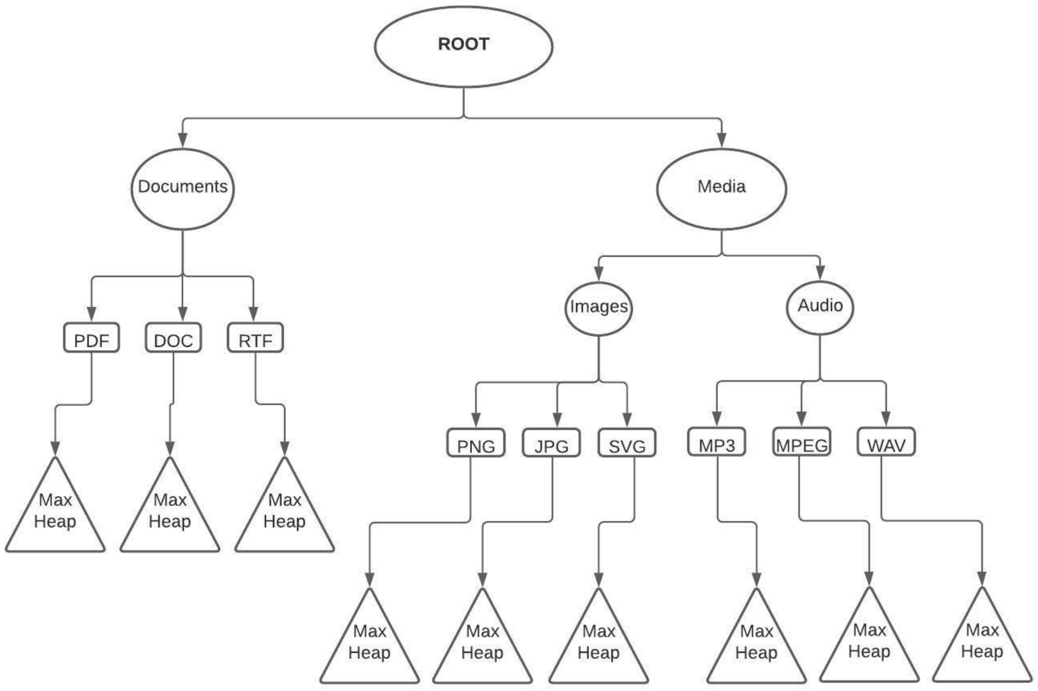

Fig. 1.3

<u>Traversing through any of the Max-Heaps</u>

Generally a max heap looks like the following one:

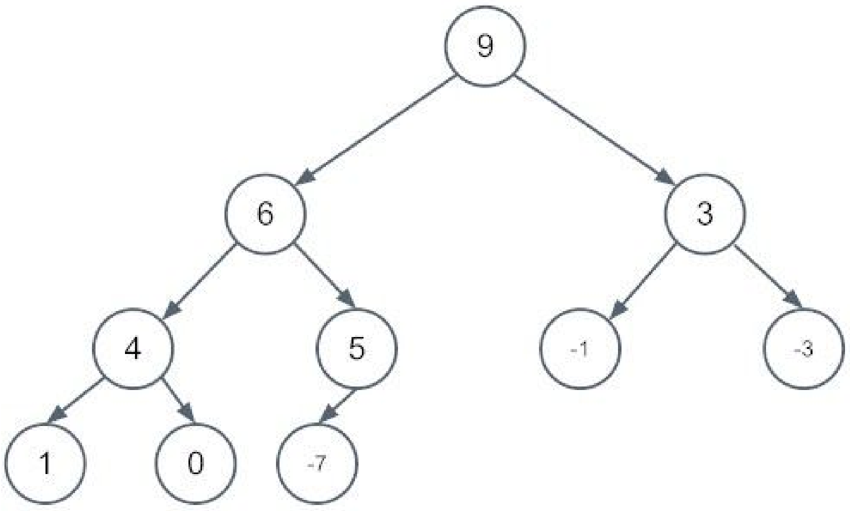

Fig. 1.4

Now, in our case the keys will be Dates-Time values, such that every new node will have highest key.
Our aim is to traverse all the nodes in such a way that the node with the largest key can be displayed on the top, whereas one with the least key comes at the bottom.

This is possible if we can successfully store all the nodes in a stack in ascending order of their keys. It's so because, once this thing is done, we'll be able to **pop** them one by one and get them in required order.

So, I've defined the following algorithm for the same:

```
----------------------------------------------------------------------------------------------
module is printData (node):
     currentList = leftList = rightList = []

     If node.left is not NULL:
          leftList = node.printData (node.left)
     end if
     If node.right is not NULL:
          rightList = node.printData (node.right)
     end if

     currentList.join(leftList)
     currentList.join(rightList)
     currentList.join(node.key)
     sort currentList
     for index in  to len(currentList):
          stackS.push(currentList[index])

     return stackS
end module
----------------------------------------------------------------------------------------------
```

For above (*Fig. 1.4*) max-heap, this module will return: **[-7, -3, -1, 0, 1, 3, 4, 5, 6. 9]**

Possible Operation

As Ramu's aim is to access his files in the best possible (efficient) way, the above special kind of **tree data structure** is best for Ramu which provides him a number of operations that he can perform easily. Because of his fantastic computer system, he needs not to worry about any of the dependencies.

Following are few of the main operations that Ramu can perform:

1. He can view all the available files present only in a specific category. like:
   a. Documents
   b. Media (both images and audios)
   c. Images
   d. Audios

2. He can view all the available files of a specific type in any of the three given categories alone. Like:
   a. Pdf files only
   b. Jpg files only
   c. Mp3 files only

3. Ramu can view these file in two different orders, i.e.:
   a. Latest data first and oldest at the bottom
   b. Oldest data first and latest at the bottom

4. He can search a specific file as well, in any of the following manner:
   a. By date added
   b. By name

Pros:

1. Recent file can be accessed quickly.
2. Hierarchical classification of files is done.
3. Ramu can perform many operations in various forms.
4. The attached can enable Ramu to implement the solution.

Cons:

1. BTree and B+ Tree could have been the best method that we have not implemented.
2. We implemented max-heap over B+ Tree because it was comparatively easy to implement & less time consuming.
3. As the data will grow, **max-heapify might take a little longer**. Although Ramu's processor has no limits, yet it's always a good practice to minimize the processor's time as much as possible.

Python Code

```
import sys
from datetime import datetime

## for formatted output in python-notebook
from IPython.display import clear_output

##-----------------------------------------------------##



class maxHeap:
    def __init__(self, maxsize = 100):
        self.maxsize = maxsize
        self.size = 0
        self.Heap = [0] * (self.maxsize + 1)
        self.data = [''] * (self.maxsize + 1)
        self.Heap[0] = sys.maxsize
        self.FRONT = 1

    def parent(self, pos):
        return pos // 2

    def leftChild(self, pos):
        return 2 * pos

    def rightChild(self, pos):
        return (2 * pos) + 1

    def isLeaf(self, pos):
        if pos >= (self.size//2) and pos <= self.size:
            return True
        return False

    def swap(self, fpos, spos):
        self.Heap[fpos], self.Heap[spos] = (self.Heap[spos],
                                            self.Heap[fpos])
        self.data[fpos], self.data[spos] = (self.data[spos],
                                            self.data[fpos])
    def maxHeapify(self, pos):
        if not self.isLeaf(pos):
            if (self.Heap[pos] < self.Heap[self.leftChild(pos)]
or
                self.Heap[pos] <
self.Heap[self.rightChild(pos)]):

                if (self.Heap[self.leftChild(pos)] >
                    self.Heap[self.rightChild(pos)]):
                    self.swap(pos, self.leftChild(pos))
                    self.maxHeapify(self.leftChild(pos))

                else:
                    self.swap(pos, self.rightChild(pos))
                    self.maxHeapify(self.rightChild(pos))

    def insert(self, key, data):
        if self.size >= self.maxsize:
            self.size += 100
            stretchKey = [0] * (100)
            stretchData = [''] * (100)
            self.Heap.extend(stretchKey)
            self.data.extend(stretchData)

        self.size += 1
        self.Heap[self.size] = key
        self.data[self.size] = data

        current = self.size

        while (self.Heap[current] >
               self.Heap[self.parent(current)]):
            self.swap(current, self.parent(current))
            current = self.parent(current)


    def listDatewise(self,pos = 1):
      currentList = leftList = rightList = []

      if self.Heap[self.leftChild(pos)] > 0:
        leftList = self.listDatewise(self.leftChild(pos))
      if self.Heap[self.rightChild(pos)] > 0:
        rightList = self.listDatewise(self.rightChild(pos))

      currentList.extend(leftList)
      currentList.extend(rightList)
      currentList.append(pos)
      currentList.sort()

      return currentList

    def getDate(self,date):
      date = str(date)
      year = date[:4]
      month = date[4:6]
      day = date[6:8]
      date = "{}/{}/{}".format(day,month,year)
      return date


    # Function to print the data-parts of the heap
    def PrintData(self):
      listToPop = self.listDatewise()

      while len(listToPop) != 0:
        print(self.data[listToPop.pop()])

    def PrintDataOld(self):
        if self.size == 0:
          return False
        for i in range(1, (self.size // 2) + 1):
            print(str(self.data[i]))
            print(str(self.data[2 * i]))
            if self.data[2 * i + 1] != '':
              print(str(self.data[2 * i + 1]))
        return True

    def searchFile(self, filename, pos = 1):
      flag = False
      if self.size >= pos:
        if self.data[pos] == filename:
          flag = True
          thisDate = self.getDate(self.Heap[pos])
          print("'{}' was saved on {}".format(filename,
thisDate))
```

```
        if not flag:
          flag = self.searchFile(filename,self.leftChild(pos))
        if not flag:
          flag = self.searchFile(filename,self.rightChild(pos))
      return flag

    def searchDatewise(self, date, pos = 1):
      flag = False
      gotList = []

      if self.size >= pos:
        if self.Heap[pos]//1000000 == date:
          gotList.append(self.data[pos])
        leftList = self.searchDatewise(date,self.leftChild(pos))
        if leftList:
          gotList.extend(leftList)
        rightList =
self.searchDatewise(date,self.rightChild(pos))
        if rightList:
          gotList.extend(rightList)

      return gotList


    def extractMax(self):
        popped = self.Heap[self.FRONT]
        self.Heap[self.FRONT] = self.Heap[self.size]
        self.size -= 1
        self.maxHeapify(self.FRONT)

        return popped


##----------------------------------------------------##




###  for saperate heaps of jpg png...
class maxHeapHolder:
  def __init__(self):
    self.data = {}

  def __addChild(self, key):
    self.data[key] = maxHeap()

  def getMaxHeap(self, name):
    if name not in list(self.data.keys()):
      self.__addChild(name)
    return self.data[name]

  def getHeapList(self):
    return self.data

###
###

class mainNode:
  def __init__(self, data = None):
    self.data = data
    self.left = None
    self.right = None

class internalNode:
  def __init__(self, name = None):
    self.name = name
    self.children = maxHeapHolder()

  def getMaxHeap(self, name):
    return self.children.getMaxHeap(name)

  def getHeapList(self):
    return self.children.getHeapList()


##----------------------------------------------------##


class mainTree:
  def __init__(self):
    self.tree = mainNode()
    self.tree.left = internalNode('Documents')
    self.tree.right = mainNode('Media')
    self.tree.right.left = internalNode('Images')
    self.tree.right.right = internalNode('Audio')

    self.doc = []
    self.img = []
    self.audio = []

  ####
  def getDocTree(self):
    return self.tree.left
  def getMediaTree(self):
      return self.tree.right
  def getImgTree(self):
      return self.tree.right.left
  def getAudioTree(self):
      return self.tree.right.right
  ####

  ####
  def heapHolderName(self, fName):
    nameList = fName.split('.')
    return nameList[1]
  ####

  ####
  def getDocHeap(self, fName):
    extention = self.heapHolderName(fName)
    self.doc.append(extention)
    doc = self.getDocTree()
    fileHeap = doc.getMaxHeap(extention)
    return fileHeap

  def getImgHeap(self, fName):
    extention = self.heapHolderName(fName)
    self.img.append(extention)
    img = self.getImgTree()
    fileHeap = img.getMaxHeap(extention)
    return fileHeap

  def getAudioHeap(self, fName):
    extention = self.heapHolderName(fName)
    self.audio.append(extention)
    audio = self.getAudioTree()
    fileHeap = audio.getMaxHeap(extention)
    return fileHeap

  def getHeapList(self, type):
    thisList = []

    if type == 'doc':
      doc = self.getDocTree()
      thisList = doc.getHeapList()
    if type == 'img':
      img = self.getImgTree()
      thisList = img.getHeapList()
    if type == 'audio':
      audio = self.getAudioTree()
      thisList = audio.getHeapList()

    return thisList

  ####


  def printDoc(self):
    docList = self.getHeapList('doc')
    for k in list(docList.keys()):
      docList[k].PrintData()

  def printImg(self):
    imgList = self.getHeapList('img')
    for k in list(imgList.keys()):
      imgList[k].PrintData()

  def printAudio(self):
    audioList = self.getHeapList('audio')
    for k in list(audioList.keys()):
      audioList[k].PrintData()

  def searchByDate(self, date):
    tempDate = date.split('/')
    tempDate.reverse()
    tempDate = ''.join(tempDate)
    key = int(tempDate)

    gotList = []

    docList = self.getHeapList('doc')
    print(key)
    for k in self.doc:
      gotFiles = docList[k].searchDatewise(key)
      if gotFiles:
        gotList.extend(gotFiles)

    imgList = self.getHeapList('img')
    for k in self.img:
      gotFiles = imgList[k].searchDatewise(key)
      if gotFiles:
        gotList.extend(gotFiles)

    audioList = self.getHeapList('doc')
    for k in self.audio:
      gotFiles = audioList[k].searchDatewise(key)
      if gotFiles:
        gotList.extend(gotFiles)

    return gotList

  #####

  def addData(self):
    print("1) Add a Document\n2) Add an Image\n3) Add an
Audio\n")
    ch = input("\nYour choise: ")

    if ch == '1':
      fName = input("Enter filename: ")
      heap = self.getDocHeap(fName)
    elif ch == '2':
      fName = input("Enter filename: ")
      heap = self.getImgHeap(fName)
    elif ch == '3':
      fName = input("Enter filename: ")
      heap = self.getAudioHeap(fName)
    else:
      print("Invalid input!")
      return False

    now = datetime.now()
    key = int(now.strftime("%Y%m%d%H%M%S")) # uniqueTime

    heap.insert(key, fName)
    return True
  def viewFile(self):
    print("1) View Documents\n2) View Images\n3) View
Audio\n\n** Any other key to view all!\n")
    ch = input("\nYour choise: ")

    if ch == '1':
      self.printDoc()
    elif ch == '2':
      self.printImg()
    elif ch == '3':
      self.printAudio()
    else:
      self.printDoc()
      self.printImg()
      self.printAudio()

  def searchFile(self):
    print("1) Search by Name\n2) Search by Date")
    ch = input("Your choise: ")
    searchFlag = False

    if ch == '1':
      thisFile = input("Enter filename: ")
      ext = self.heapHolderName(thisFile)

      if ext in self.doc:
        docList = self.getHeapList('doc')
        searchFlag= docList[ext].searchFile(thisFile)
      elif ext in self.img:
        imgList = self.getHeapList('img')
        searchFlag = imgList[ext].searchFile(thisFile)
      elif ext in self.audio:
        audioList = self.getHeapList('audio')
        searchFlag = audioList[ext].searchFile(thisFile)

    elif ch == '2':
      thisDate = input("Enter the date (dd/mm/yyyy): ")
      gotList = self.searchByDate(thisDate)
      if gotList:
        print("\nFiles saved on",thisDate,":\n")
        for files in gotList:
          print(files)
        searchFlag = True
      else:
        searchFlag = False


    if not searchFlag:
      print("File does not exist!!")
    return searchFlag

##------------------------------------------------------##


def menu():
  print("""
         Main-Menu
        -----------\n
        1) Add New File
        2) List Files
        3) Search or Open File(s)

        ** Any other key to exit!
        """)
  return input("\nEnter your choice: ")


##------------------------------------------------------##


if __name__ == '__main__':
  thisMainTree = mainTree()

  while True:
    clear_output()
    now = datetime.now()
    ch = menu()

    ## Decision
    if ch == '1':
      clear_output(wait=True)
      thisMainTree.addData()
      input("\nPress Enter!!")

    elif ch == '2':
```

```
  clear_output(wait=True)
  thisMainTree.viewFile()
  input("\nPress Enter!!")

elif ch =='3':
  clear_output(wait=True)
  thisMainTree.searchFile()
  input("\nPress Enter!!")

else:
  clear_output(wait=True)
  print("Good Bye!")
  break
```

Solution B:

"Write a python program making use of any data structures to automatically detect, identify, rearrange and correct the words in jumbled sentences"
Such as:

1. ***ORF _ MECO FEEFOC***
2. ***effect ew ni did uchm sales last ton eary pimrovement eunron si oodg***

"Explain your solution in detail."

--

Prerequisite

Before going into the solution, we need to keep following in the mind:

1. Tries
2. PyEnchant (library)

Tries:

- Trie is a special kind of data structure which is a type of the tree.
- Tries are used to store strings that can be visualized like a graph.
- Each node of a trie consists of at max 26 children.

PyEnchant:

- This is a spell-checking library in python that includes many functions to deal with the words of english language.
- One can verify the existence of a word from english language using the check() method of *Dict* that is defined in this library.

As per the given problem, we have to identify the misspelled word(s) in the given sentence, detect them and rearrange to correct them.
As per the given examples, it is clear that *the word length and the frequency of all the letters must remain exactly the same as in the provided sentence.*

Although, it's not specified whether we have to correct the sentence also or not. Hence, I have only focused on correcting the words.

Data Structure

As specified earlier, a trie is a good way to store english words.
These are highly applicable in the *cross-word* and other *language word games*. For an instance, we can have a look on the following trie that holds 4 distinct words in it.

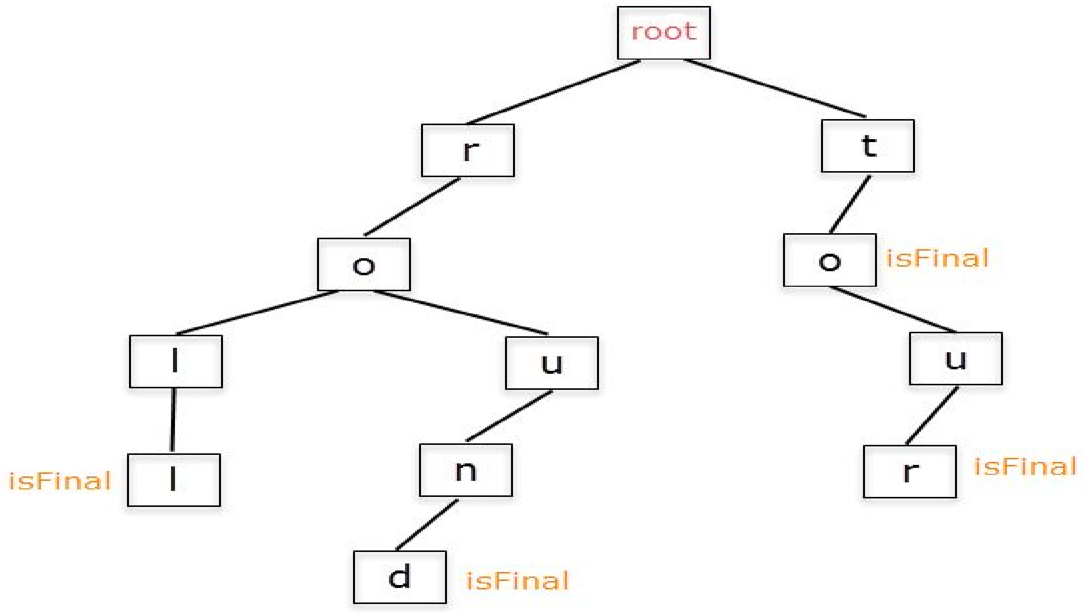

Fig. 2.1

Valid words from above trie are:

- to
- tour
- roll
- round

I aim to use a **Trie** to store all the valid words in it so that we can easily not only detect the mis-spelled words, but can also correct them.

Word Correction

In this approach, initially I would check for every word of the sentence whether it is correct or not. To do so we can use tries. But a challenge is how to find those words that are not in trie.

So, in that situation we can use the *check()* function from **PyEnchant** library. It can help to validate the spellings from different versions of languages viz.: *en_GB, en_US, de_DE, fr_FR*.

A sample usage of the same is can be:

```
import enchant
myDict = Dict(en_US)
myDict..check("Hello")   # returns True
myDict..check("Hlo")   # returns False
```

I have written the following algorithm to automatically detect and correct the mis-spelled words of a sentence:

```
1. Take a sentence
2. wList = list of words of sentence
3. For W in wList:
    a. if W exists in trie:
        i.   scan next word (step 3)
    b. else:
        i.   perm = All possible permutations of letters in word W
       ii.   For P in perm:
               1. if P in trie:
                   a. Correct the W in wList
                   b. Goto step 3
               2. Else search using PyEnchant
                   a. if found:
                       i.   Add P in trie
                      ii.   Correct W in wList
                     iii.   Goto step 3
    c. end if
4. correctSentence = join words of wList
5. print correctSentence
```

Initially, the program will need to use the external library. But, as much as it rectifies the errors, the method will become self-dependent.
It is also possible that after correcting a few hundred of sentences, the program will be as perfect as it won't require the **PyEnchant** at all for common word correction.

This program is a best practice to understand the concept of **Supervised Machine Learning**, where the machine learns from the errors and tries to recall as many corrections as possible.
In my case, if the program gets to know about a new valid sentence from english dictionary once, it recalls the same and never needs to go to the dictionary again for verification.

<u>Pros:</u>

1. A good implementation to understand the concept of **Supervised Machine Learning** technique.
2. Good use of a **tries** i.e. a type of Tree Data Structure.
3. Fast and automatic.

<u>Cons:</u>

1. Unable to understand the context of the sentence.

Python Code

```
#!apt install enchant
#!pip install PyEnchant
import enchant

#########################
#       Trie Node       #
#########################
class TrieNode:
    def __init__(self):
        self.children = [None]*26
        # isEndOfWord = True signifies end of the word
        self.isEndOfWord = False

###################
#       Trie      #
###################
class Trie:
    def __init__(self):
        self.root = self.getNode()

    def getNode(self):
        return TrieNode()

    def __charToIndex(self,ch):
        return ord(ch)-ord('a')

    ################################

    ## insert new word in trie
    def insert(self,key):
        traverse = self.root
        length = len(key)
        for level in range(length):
            index = self.__charToIndex(key[level])
            if not traverse.children[index]:
                traverse.children[index] = self.getNode()
            traverse = traverse.children[index]

        traverse.isEndOfWord = True
    ## search a word in trie
    def search(self, key):
        traverse = self.root
        length = len(key)
        for level in range(length):
             index = self.__charToIndex(key[level])
             if not traverse.children[index]:
                 return False
             traverse = traverse.children[index]

        return traverse != None and traverse.isEndOfWord


################################
#  Word-Permutation generator  #
################################
def wordPermutation(front, last=''):
  finalList = []
  if len(front) == 0:
    finalList.append(last)
  else:
    for index in range(len(front)):
      finalList.extend(
                     wordPermutation(front[0:index] +
                                     front[index+1:],
                     last+front[index]))
  return finalList


################################
#  Main Class for Correction   #
################################
def correctSen(sentence):
  thisTrie = Trie()
  myDict = enchant.Dict("en_US")

  wList = sentence.split(" ")
  for index in range(len(wList)):
    word = wList[index]
    if thisTrie.search(word):
      continue
    else:
      tmpWord = list(word)[1:]
      tmpWord.sort()
      tmpWord = word[0]+''.join(tmpWord)
      permList = wordPermutation(tmpWord)
      permList.remove(word)
      for pWord in permList:
        if thisTrie.search(pWord):
          break
        elif myDict.check(pWord):
          thisTrie.insert(pWord)
          wList[index] = pWord
          break

  finalSentence = " ".join(wList)
  return finalSentence


################################
#       Driver Function        #
################################
if __name__ == '__main__':
  while True:
    print("\nHit enter without any input to get exit!")
    sen = input("Enter a sentence with jumbled words: ")
    if sen is '':
      break
    else:
      print("After spell-correction: ",correctSen(sen))
```

Solution C:

"Capture a photo of your neighborhood, and label each of the objects. Create a tree structure with each node representing an object and link weights showing the similarity between them. What will be the best way to implement this? Which data structure will you use and why? Show with an example code, providing detailed explanations of the pros and cons of your approach."

Prerequisite

Before going into the solution, we need to keep following in the mind:

1. *Types & Composition of Images*
2. *Image Processing Libraries*

Types & Composition of Images:

- Images are digitally compressed forms of the visual data that is often collected from the Cameras or other optical sensing devices.
- 2 widely classified types of images (graphics) are: **Vector** and **Raster Graphics**.
- There are many different forms of compressions for these graphics like JPG, PNG & SVG etc..
- Along with many details, all the images can be considered as a **group of pixels**, representing colour at every distinct position.
- Majourly based on the colours, images can be classified into 3 main categories:
    1. Black-and-White images (1 bit per pixel)
    2. Grayscale images (1 byte per pixel)
    3. RGB (Coloured) images (3 bytes per pixel)
- We can figure out different **objects** based on the variation of colours in an image.

Image Processing Libraries:

- There are various libraries in python like ….. Which are helpful in working with the image files.
- Some common operations performed on the images using these library functions are like: scaling, clipping, reshaping and colour manipulation.

Object Detection from An Image

Object Labeling

In order to detect some objects from the images, I'll prefer to **separate the groups of pixels** from the same image, **based on their distance and variance of colours**.

These distinct groups can be considered as distinct **objects** or the parts of a big object.

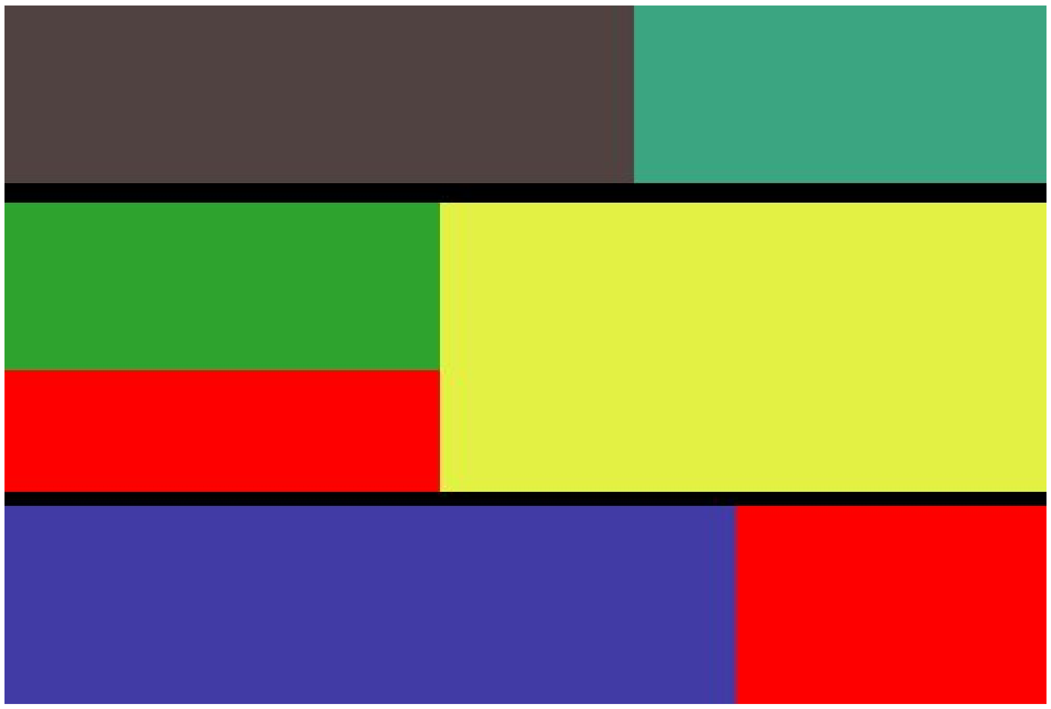

Fig. 3.1

To separate objects on the basis of the colours (similar) we can scan pixels row-wise with adjacent columns. Before that, it will be a good idea to convert the given image into grayscale.

This image (Fig. 3.1) is only for understanding the algorithm. The actual image will definitely by much varied in terms of the colours and objects in it.

An algorithm to extract the **one** object from an image is given on the next page.

Following algo best describes this solution:

```
module is getObject(image, begRowIndex, begColIndex):
     image  = convertToGrayscale(image)

     width, height
     allObjects = []

     for row in (begRowIndex to height-1):
          if (row > 0) and (pixelVal-image[row, col+1] <= 10):
               break
          end if
          for col in (begColIndex to width-1):
               pixelVal = image[row, col]
               if (col+1 != width) and (pixelVal-image[row,
                                                   col+1] <= 10):
                    thisObject.append(pixelVal)
               else:
                    break
               end if
          end for
     end for

     return thisObject
end module
```

This algorithm extracts a series of the pixels from an image and returns as a list or array.

Interesting things:

1. **Extracts pixels easily** as the image is initially converted to the **grayscale**
2. On receiving the pixel-list the **actual object is separated and from the main coloured image** and **labeled**.

In case if we call the above algorithm in the Fig. 3.1, it will result like below:

`getObject(fig3_1, begRowIndex = 0, begColIndex = 0)`

a. **fig3_1** = signifies Fig 3.1 (assumed)
b. **begRowIndex** = 0 (first row of pixels)
c. **begColIndex** = 0 (first column of pixels)

The algorithm proceeds as follows:

1. Conversion to grayscale format

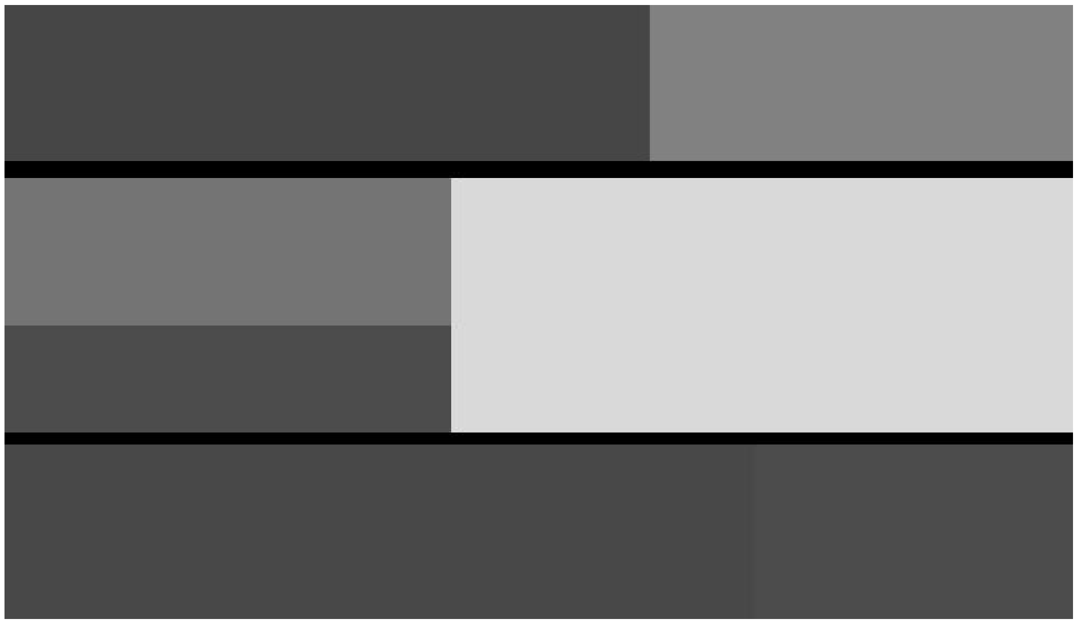

Fig. 3.2

2. Initializing the scanning pixel position (begRowIndex, begColIndex): marked Red

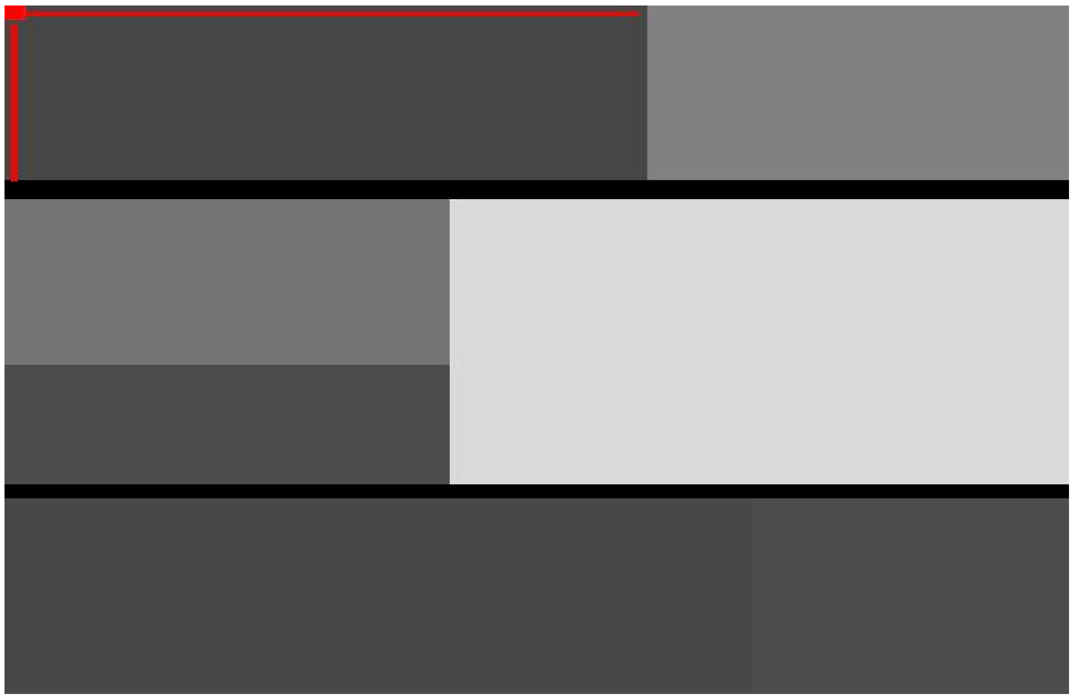

Fig. 3.3

Here, pixels are scanned column-wise until the same sequence of pixel colour (+/- 10).

3. Returning the list of pixels (that aligns with the extracted object)

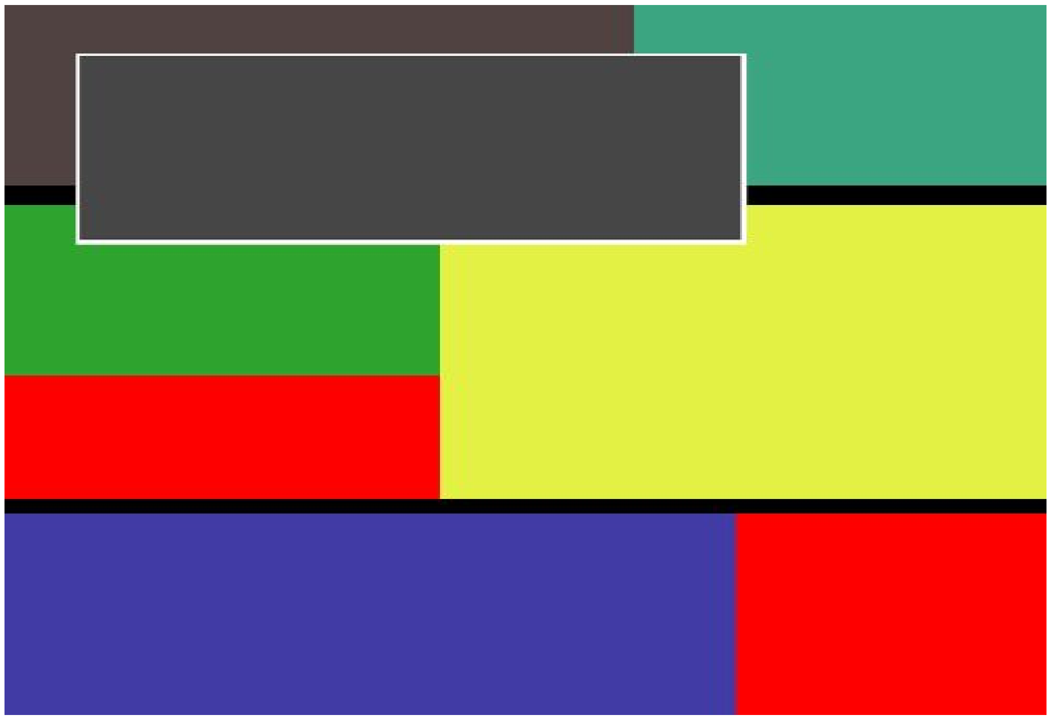

Fig. 3.4

As it is clear from the above returned object's pixel-list aligns with the actual object i.e. the top-left block (with light-brown shade) in the coloured image.
Hence, this object is extracted and labeled, say **Obj1**.

Similarly, all the different coloured blocks can be considered as different objects. Further these **objects can be stored separately**. But, these objects can have some kinds of relations among them. Hence, we can assign some weight to these objects, depending upon the intensity of the relation between them.

<u>**Weight Assignment**</u>

The weight of the objects can be assigned on following 2 basis:

1. Relation between 2 or multiple objects (eg. distance and colour variations)
2. Qualities of blocks, individually (eg. size)

For instance, in Fig 3.1, the Yellow block can have the maximum weight whereas the 2 black blocks have least weight.
Although, both the black blocks can have the same parent and similarly, two red blocks can also be linked in a similar manner.

Storing the Labeled Objects

The main **objectives** behind labeling, weight allocation and separate storage of the blocks of an image are:

1. Image recognition
2. Searching the existence of a sequence of pixels in a specific poration
3. **Reshaping, resizing** and doing some **colour manipulation** in a specific portion or rather **objects** of the images (and bringing effects into the related objects, too).

The following format of the **tree** will be best to implemented:

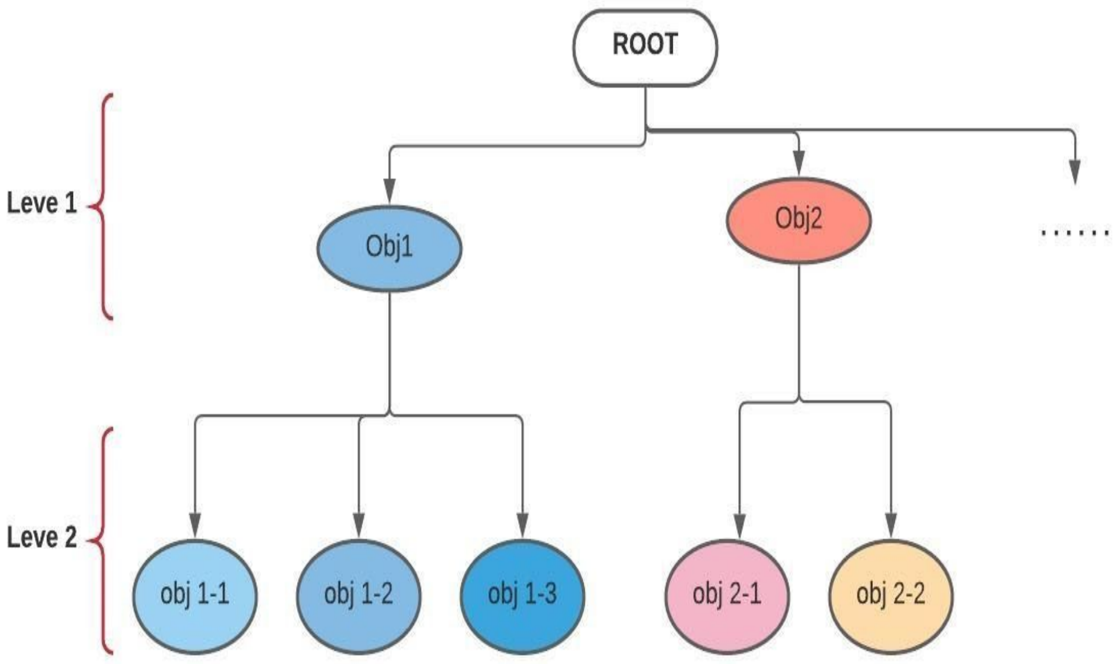

Fig. 3.5

Part of the Tree:

Describing Fig 3.5 in details:

Level 1:

1. Each node will have **completely distinct colours**
2. Each node represent parts of distinct **objects**
3. If an object has multiple parts, part in level one will have **maximum weight**.
4. Obj1 and Obj2 are completely distinct in colour.

Level 2:

1. Nodes with **similar colours** are stored as the part of **same sub-tree**
2. **Weights** assigned to these nodes (sub-objects) are **less than the parent**.
3. Obj1-1, Obj1-2 and Obj1-3 have similar colour as **Obj1**, and Obj2-1, Obj2-2 and Obj1-3 have similar colour as **Obj2** and so on.

The tree can grow even more in the same manner.

Pros:

1. Easy to understand.
2. Splitting of detected objects can be done on a very large scale as much as desired (based on the threshold, as in above algo it's kept as **10**).
3. Labeling and the weight assignment is done considering many relating between the objects.

Con:

1. Use of graphs had been much better than trees.
2. Tricky and slightly complex to implement.

** Unfortunately, as per the question, usage of tree data structure was my restriction.*

Python Code

Please accept my apologies for not submitting the python code as I'm continuously facing some errors even after debugging it several times.