MoveMi : CS110 Final Project

Introduction

Prompt

This project addresses Prompt 1: using data from a publicly available dataset, build an application that uses concepts from CS110.

Purpose

Content Disovery is broken in the age of Social media

Content discovery and self-expression are hampered by the typical use of platforms today.

Firstly, content consumptions is changing rapidly among teens. Time spent reading and even on traditional media is rapidly declining, and teens are instead spending large amounts of time on social media. The average teenager spends just 7 minutes reading a day, but spend 2.5 hours a day on social media and another 2 hours on television/youtube.

One reason why reading has declined so rapidly is because social media and search engines do not not support content discovery of written works effectively. Content created on the web tend to be on centralized platforms such as Reddit/Facebook/Youtube instead of smaller websites/blogs.

This gives rise to 2 modes of content discovery.

In the first mode of content discovery, through search engines such as google, the user needs to know exactly what he wants and can find it. However, this leads to the vast majority of content ends up undiscovered, even when it could add great value to the end user's life. Search engines aid discovery of the known, but not exploration of the unknown, which greatly limits the scope of content users discover.

In the second mode of content discovery, through social networks, individuals discover content through their friends and share content with them. This expands the scope of content discovery relative to search engines, to incorporate exploration of the unknown. However, the social element leads to a lack of sharing. Individuals are less likely to voice out their opinions due to a fear of offending as well as a assumption that not many people would share their interests. Because all interactions on social media are not anonymous, fear of social judgement also has an effect, which reduces the extent to which individuals respond to content through comments, which further reinforces the lack of social attention shown to content posted.

Self-expression is therefore not widespread, genuine or fulfilling

In general, these factors lead to content being posted on social media becoming less in-depth and specialized, and more attention-seeking and mainstream - in genral the shift is away from content articles and instead towards photos, selfies and the like.

This, in turn, affects self-expression. More validation is provided for mainstream, attention-seeking posts, and social validation is a powerful incentive, especially among youth because they are continuing to form a self-identity. Therefore, youth tend to express themselves through self-validating posts such as photos, instead of through the written word which takes more effort but provides less validation. Moreover,with a lack of access to content that they find relatable and accessible, many individuals simply do not read enough, and do not know how to self-express

Solution: Content Discovery through anonymous self-expression

The purpose of this algorithm is to implement a third model of content discovery - discovery of content through anonymous self-expression in the form of writing, similar to diary-writing. Individuals are aided and urged to write a entry where they are vulnerable and express their feelings (anonymously). Their diary entry is analyzed and the users are returned relatable content that will touch them and add value to their lives.

The purpose of this is 2-fold. Firstly, it will be able to provide content that is much more valuable to the end user. While initially the content served to the end user is either through his/her social network or through an intentional search, both of which are not effective when the end user is unclear what he is searching for in the first place , now the content is exactly tailored to the end user's emotional state or life circumstances, and is able to enrich and add value to his/her life. A person who expresses his insecurity over his exam grades, for example, would get directed to posts that are also about exam grades which mostly closely match his own.

Second, it will make genuine self-expression much easier. While previously, genuine, writing-baseed self-expression was done in isolation due to the lack of social validation. Now there is a third, more -powerful incentive to self-express - discovering valuable content that will touch and enrich the end user.

The content scope is limited to diary entries or other forms of emotionally invested medium-to-long forms of writing, in order to be best able to provide content that the end user will find moving.

Overview of Process

The algorithm has 2 broad processes - indexing and retrieval. Every diary entry goes through both of these processes.

In indexing, the is analyzed and stored so that it can be easily discovered if it is deemed relevant to a later query. The diary entry is run through Google's Natural Language processing API to discover the content category as well as the main entities present in the entry, and using hash tables, search trees and heaps, this content is indexed. (This process will be explained in greater detail later)

In retrieval, the diary entry's content categories and entities are analyzed and other diary entries that have the most similar content categories and entities are returned.

This paper will first outline the overall process and the high-level view of the data structures and algorithms used in designing this app, and the reasons behind their choice. Second, the overall algorithm will be explained, Third, the details of the implementations of heaps, hash tables and trees will be elaborated on. Lastly, the overall algorithm performance and complexity will be analyzed.

Data structures

Story Array
Purpose: An array containing all stories

Category Tree

Purpose: This is a hash table that stores Category Objects.
Subsumed Under: None
Data: [Category Object] x n, where n = length of CategoryTree hash table

Category Object

Purpose: This represents one category, and stores stories that are classified under that category
Subsumed Under: Category Tree Data Structure
Data:
ParentCategory - Index of Parent CategoryObjects
ChildrenCategories - Array of indices of children CategoryObjects
Entities Array - Unique array of Entity Objects

Entity Object

Purpose: This is a max-heap that is present for each entity in each category.
Subsumed Under: Category Object
Data:
Entity Max Heap, represented as an array.

Algorithm Overview

Pre-requisites
1) Relevant libraries are installed
2) Output.txt inserted in local directory
3) Google Natural Language API is activated in Google Cloud Console (My personal API key is provided for now)

```
!pip install --upgrade google-api-python-client
!pip install pyhash

from googleapiclient.discovery import build
import pyhash
lservice = build('language', 'v1', developerKey='AIzaSyC_BioVBQys8caW3iEUvQ7Ejt_C2b
```

```
Requirement already up-to-date: google-api-python-client in /usr/local/lib/py
Requirement already satisfied, skipping upgrade: httplib2<1dev,>=0.9.2 in /us
Requirement already satisfied, skipping upgrade: uritemplate<4dev,>=3.0.0 in
Requirement already satisfied, skipping upgrade: six<2dev,>=1.6.1 in /usr/loc
Requirement already satisfied, skipping upgrade: google-auth>=1.4.1 in /usr/l
Requirement already satisfied, skipping upgrade: google-auth-httplib2>=0.0.3
Requirement already satisfied, skipping upgrade: cachetools>=2.0.0 in /usr/lo
Requirement already satisfied, skipping upgrade: pyasn1-modules>=0.2.1 in /us
Requirement already satisfied, skipping upgrade: rsa>=3.1.4 in /usr/local/lib
Requirement already satisfied, skipping upgrade: pyasn1<0.5.0,>=0.4.1 in /usr
Requirement already satisfied: pyhash in /usr/local/lib/python3.6/dist-packag
```

Step 1: Initialize Classes

There are 2 classes to be initialized - maxHeap and category. These are explained above

```
class maxHeap:

  def __init__(self):
    self.salienceValues = []


  def push(self, data):
    self.salienceValues.append(data)
    self.floatUp(len(self.salienceValues) - 1)
  def getMax(self):
    return self.salienceValues[0]

  def delete(self,data):
    index = self.salienceValues.index(data)

    if len(self.salienceValues) > 2:
      self.swap(index, len(self.salienceValues) - 1)
      returnValue = self.salienceValues.pop()
      self.bubbleDown(index)
    elif len(self.salienceValues) <= 2:
      returnValue = self.salienceValues.pop()
    return returnValue

  def swap(self, i, j):
    self.salienceValues[i], self.salienceValues[j] = self.salienceValues[j], self.s

  def floatUp(self, index):
    parent = index//2
    arrayInput = self.salienceValues

    if index <= 0:
      return
    elif arrayInput[index]["salienceValue"] > arrayInput[parent]["salienceValue"]:
      self.swap(index, parent)
      self.floatUp(parent)

  def bubbleDown(self, index):
    left = index * 2
    right = index * 2 + 1
    largest = index
    if len(self.salienceValues) > left and self.salienceValues[largest]["salienceVa
      largest = left
    if len(self.salienceValues) > right and self.salienceValues[largest]["salienceV
      largest = right
    if largest != index:
      self.swap(index, largest)
      self.bubbleDown(largest)

class storyCategory :
    def __init__(self, name):
        self.name = name
        self.parent = None
        self.children = []
        self.entities = [None]*500

    def findParent(self):
      return self.parent
```

Step 2: Define the Hash Function

The Hash function will be used for
1)Hashing the category name to decide which position in the CategoryTree array to insert the story in
2)Hashing the entity to decide which position in the Entities array, present in each Category Object, to insert the story in

```
def hashFunction(item, arrayLength,seed):
  fp = pyhash.fnv1_32(seed)
  return (fp(item) % arrayLength)
```

Step 2: Storage of Story in StoryArray

The post is stored in an array, and its a dictionary containing the story and its index is returned . This was conducted in the storeStory() function

```
def storeStory(story,storyArray):
  #In this function,the quote is stored in an array, and the index of the array is

  storyArray.append(story)
  storyIndex = storyArray.index(story)

  storyDict = {}
  storyDict["story"] = story
  storyDict["index"] = storyIndex
  return storyArray,storyDict
```

Step 3: Analyze story using Google Natural Language API to get story's content category and entities

The post is fed through the Google Natural Language AI to find out what content category it is in, and what entities it has. This is stored in a storyObject. This was conducted in the classify() function

```
def classify(storyDict):

  #This function takes in a quote, runs it through the Google Natural Language API
  story = storyDict["story"]
  StoryEntitiesArray = []

  entitiesResponse = lservice.documents().analyzeEntities(
    body={
      'document': {
         'type': 'PLAIN_TEXT',
         'content': story
      }
    }).execute()
  try:
    entities = entitiesResponse['entities']
  except:
    entities = ["None"]
  for i in range(0,len(entities)):
    entityName = entities[i]['name']
    entitySalience = entities[i]['salience']
    entityIndex = storyDict["index"]
    entitiesDictionaryEntry = {}
    entitiesDictionaryEntry["Name"] = entityName
    entitiesDictionaryEntry["salienceValue"] = entitySalience
    entitiesDictionaryEntry["index"] = entityIndex

    StoryEntitiesArray.append(entitiesDictionaryEntry)

  response = lservice.documents().classifyText(
    body={
      'document': {
         'type': 'PLAIN_TEXT',
         'content': story
      }
    }).execute()
  try:
    category = response['categories'][0]['name']
  except:
    category = "None"

  overallDictionary = {}
  overallDictionary["Story"] = story
  overallDictionary["Category"] = category
  overallDictionary["Entities"] = StoryEntitiesArray
  overallDictionary["Index"] = storyDict["index"]
  return overallDictionary
```

Step 4: Store content category in Category tree, initialize relationships

The content category is run through a hash function, and in the corresponding index in the hash array, a subcategory object with attributes p for parent subcategory , c for children subcategory and E for entity hashtable are stored

```
def initializeRelationships(categoryTree,storyObject):
  categoryTotal = storyObject["Category"]
  final = []
  currentString = ""
  for i in range(1,len(categoryTotal)):
    if(categoryTotal[i]) == '/':
      final.append(currentString)
      currentString = ""
    else:
      currentString += categoryTotal[i]
  final.append(currentString)

  hashSeed = 3
  insertPosition = hashFunction(final[-1],len(categoryTree),3)


  '''
  Open addressing is implemented
  '''
  if categoryTree[insertPosition] == None:
    categoryTree[insertPosition] = storyCategory(final[-1])

    for i in range(1,len(final)):
      parentPosition = hashFunction(final[-i-1],len(categoryTree),3)
      categoryTree[parentPosition] = storyCategory(final[-i-1])
      categoryTree[parentPosition].children.append(hashFunction(final[-i],len(categ

      childPosition = hashFunction(final[-i],len(categoryTree),3)
      categoryTree[childPosition].parent = hashFunction(final[-i-1],len(categoryTre

  while categoryTree[insertPosition].name != final[-1]:
    hashSeed += 1
    insertPosition = hashFunction(final[-1],len(categoryTree),hashSeed)
    if categoryTree[insertPosition] == None:
      categoryTree[insertPosition] = storyCategory(final[-1])

      for i in range(1,len(final)):
        parentPosition = hashFunction(final[-i-1],len(categoryTree),3)
        categoryTree[parentPosition] = storyCategory(final[-i-1])
        categoryTree[parentPosition].children.append(hashFunction(final[-i],len(cat

        childPosition = hashFunction(final[-i],len(categoryTree),3)
        categoryTree[childPosition].parent = hashFunction(final[-i-1],len(categoryT




  #if categoryTree[insertPosition] is some other guy, then hash again with a new se

  return categoryTree[insertPosition]
```

Step 4: Store content category in Category tree, initialize relationships

The entities of the post are run through a hash function. In the corresponding index in the entity hashtable E, which contains a max heap, the node at the root of the max-heap, which has the highest salience, is appended to returnStoryPointers, an array of the relevant stories that will be returned to the user.

Then, the story itself is appended each of the max heaps which correspond to its entities

```
def insertStoryIntoTree(newCategory,categoryTree,storyObject):

  '''
  This variables stores the pointers to the stories in the storyArray that are
  most relevant
  '''
  returnStoryPointers = []
```

```
    '''
    For each entity in the story being analyzed, first the category it is
    classified under is searched for stories that share the same entities and
    are hence relevant
    '''
    for i in range(0,len(storyObject["Entities"])):
      entity = storyObject["Entities"][i]
      entityName = entity["Name"]
      entitySalience = entity["salienceValue"]
      entityIndex = entity["index"]
      storageIndex = hashFunction(entityName,len(storyObject["Entities"]),3)

      if (newCategory.entities[storageIndex] != None):  ##AND whatever is in entities
        index = newCategory.entities[storageIndex].getMax()["index"]
        if (index not in returnStoryPointers) and (index != storyObject["Entities"][0
          returnStoryPointers.append(index)

        '''
        #This optional piece of code expands the functionality such that instead
        #of returning the top 1 story per entity, it returns the top n stories,
        #with n defined by the variable numberPerEntity. It is not activated
        #because of the insufficient number of stories so far.

        poppedStoriesStorageArray = []
        numberPerEntity = 3
        for i in range(0,numberPerEntity):
          poppedStoriesStorageArray.append(newCategory.entities[storageIndex].pop(new
          returnStoryPointers.append(newCategory.entities[storageIndex].getMax())

        for i in range(0,len(poppedStoriesStorageArray)):
          newCategory.entities[storageIndex].push(poppedStoriesStorageArray[i])

        '''
      #else if not equal none but not same as being queried, hash with a different se


      else:
        newCategoryMaxHeap = maxHeap()
        newCategory.entities[storageIndex] = newCategoryMaxHeap

      '''
      After exploring the category for similar stories, the story being analyzed
      is inserted into the tree
      '''
      newCategory.entities[storageIndex].push(entity)
    return returnStoryPointers
```

Step 5: Searching of parent and sibling categories

If the number of stories returned from searching for entities at a particular category is less than 15, then the parent and sibling categories will also be searched for the same entities

```
def searchParentSiblingCategories(returnStoryPointers,newCategory,categoryTree,stor
    '''
    If there are less than 15 stories returned, the parent and sibling categories
    are also explored for stories that share the same entities
    '''
    familyCategories = []

    if newCategory.parent != None:
      parentCategory = categoryTree[newCategory.parent]
      familyCategories.append(parentCategory)

      siblingCategories = parentCategory.children
      for child in siblingCategories:
        if categoryTree[child] != newCategory:
          siblingCategory =  categoryTree[child]
          familyCategories.append(siblingCategory)

    for entity in storyObject["Entities"]:
      entityName = entity["Name"]
      entitySalience = entity["salienceValue"]
      storageIndex = hashFunction(entityName,len(storyObject["Entities"]),3)


      if (newCategory.entities[storageIndex] != None):
        for familyCategory in familyCategories:
          if familyCategory.entities[storageIndex] != None:
            index = familyCategory.entities[storageIndex].getMax()["index"]
            if (index not in returnStoryPointers) and (index != storyObject["Entities
              returnStoryPointers.append(index)

    return returnStoryPointers
```

Test case

Data Scraping from Humans of New York

This idea was heavily inspired by the form and popularity of the "Humans of New York" facebook page, which posts photojournalist entries where individuals are photographs and tell a story. The stories tend to be real, authentic and cross geographical, cultural and economic boundaries, and the page has exploded in popularity. Therefore, the dataset used was obtained from the Humans of New York facebook page using a custom-built web scraper that ran on Selenium, ActionDriver and BeautifulSoup Libraries. This program needs to be locally run, with the dependencies installed, because it initiates, controls and tracks a new web browser object (Firefox), which cannot be done via an online python notebook environment. 84 stories were downloaded to give a sample dataset

```
'''
#This code is commented out in order to prevent interferance with the "Run all"
#function in Python notebooks, as it is meant to be be run locally

from selenium import webdriver
import time
import csv
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Firefox(executable_path = '/usr/local/bin/geckodriver')
driver.get('https://www.facebook.com/pg/humansofnewyork/photos/?ref=page_internal')
x = input("PROCEED?")
urlStorageArray = []
storyStorageArray = []

storyRange = 3
#Gets all urls
for i in range(storyRange):
    print("FRACTION DONE:",i,"/",storyRange)
    time.sleep(3)
    print("NEXT")
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight)")

a = driver.find_elements_by_css_selector('a')


for element in a:
    string = element.get_attribute('href')
    if "photos/a" in string:
        urlStorageArray.append(string)
```

```
text_file = open("Output.txt", "w", encoding='utf-8')
for i in range(0,len(urlStorageArray)):
    print("COMPLETED: ", i, "/", len(urlStorageArray))
    item = urlStorageArray[i]
    driver.get(item)
    element = driver.find_element_by_xpath('/html/body/div[1]/div[3]/div[1]/div/div
    storyStorageArray.append(element.text)
    text_file.write(storyStorageArray[i] + '\n')

text_file.close()

'''
```

```
'\n#This code is commented out in order to prevent interferance with the "Rur
```

Pulling and storage of sample data into an array

The sample data, which is stored in a file "Output.txt" in my Google Drive, is opened and its contents are stored in an array.

```
diaryEntries = []

from google.colab import drive
drive.mount('/content/drive')
with open("/content/drive/My Drive/Output.txt", "r", encoding='utf-8') as f:
    lines = f.readlines()

for line in lines:
    newVar = (line)
    diaryEntries.append(newVar)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
```

Running of algorithm

The last entry of the 84 stories pulled from the Humans of New York page is used as a test case, and the rest of the entries are used as training data.

It can be seen that the algorithm works well, with the majority of the stories returned being similar in content to the first story, about a genocide in Rwanda.

```
storyArray = []#Keeps all stories
categoryTree = [None] * 6250 #Keeps all categories. Has category objects, which con

'''
For each story, the 5 steps are performed
'''
for i in range(0,len(diaryEntries)):

  story = diaryEntries[i]

  #Step 1
  storyArray,storyPointer = storeStory(story,storyArray)  #O(1) wrt current Size
```

```
    #Step 2
    storyObject = classify(storyPointer) #O(1)

    #Step 3
    newCategory = initializeRelationships(categoryTree,storyObject) #O(1)

    #Step 4
    returnStoryIndices = insertStoryIntoTree(newCategory,categoryTree,storyObject)

    #Step 5
    if len(returnStoryIndices) < 15:
      returnStoryIndices = searchParentSiblingCategories(returnStoryIndices,newCatego

testCaseStory = diaryEntries[-1]

output = []
print("Input")
printStatement = "Diary Entry:" + str(testCaseStory)
print(printStatement)
output.append(printStatement)

print("Output")
for indice in returnStoryIndices:
  printStatement = "Returned Related Story:" +str(storyArray[indice])
  print(printStatement)
  output.append(printStatement)


with open("/content/drive/My Drive/FinalOutput.txt", "w", encoding='utf-8') as f:
  for outputEntry in output:
    f.write(outputEntry)
```

```
Input
Diary Entry:(1/4) “When I was twelve years old, they undressed a Tutsi girl i

Output
Returned Related Story:(1/3) “The genocide was an opportunity to get rich. Mu

Returned Related Story:(9/9) “This is a picture of my father before the genoc

Returned Related Story:“There was a huge puzzle after the genocide. How do yo

Returned Related Story:(1/7) “I inherited this orphanage from my father. I wa

Returned Related Story:(2/6) “We stayed in the house for two nights. On the t

Returned Related Story:(1/9) “My father was well respected in the community.

Returned Related Story:(3/3) “When I woke up, I found myself lying alongside

Returned Related Story:“My father was a talented engineer. He could fix any t

Returned Related Story:(2/4) “There had always been permission to kill any Tu

Returned Related Story:(1/4) “When we heard the president had been killed, I

Returned Related Story:(7/7) “While I was being interviewed on the radio, Car

Returned Related Story:(8/9) “There were twelve people in my family before th

Returned Related Story:(1/3) “My husband and I were shopkeepers at the time o
```

Explanation of algorithmic design

Hash table

Analysis

A hash table hashes the input, generates the index of the hash table where the input should be stored and stores the input in that index. In this application, hash tables were used to store categories, and within categories hash tables were used to store entities. Noteworthy is the fact that while normal hash tables allow for collisions, my implementation of hash tables simply combines the entities in order to reduce the complexity of the program.

Insertion and Access time complexity

A hash table has an average case insertion and access time complexity of O(1), as compared to an array which also has a insertion time complexity of O(1) but has an access time complexity of O(n). The complexity of the hash table depends on the effectiveness of the hash function in spreading out the data throughout the hash table, as well as the size of the hash table. If the hash table is small or if the hash functions fails to give a uniform distribution of hash table indexes, collisions would occur where the same index would have many entries stored. Open addressing is implemented, hence the function will be re-hashed to find the next available open slot. This would lead to the access time complexity approaching O(m), where m is the length of the hash table, as compared to O(1) if no collisions occur.

Implementation

Hash Function

The hash function used was the Fowler-Noll-Vo hashing function. he hash function takes in the data being hashed as an argument.

For each input element, it multiplies the offset_basis, which is a constant depending on the size of the hash(which is constant in this case), by FNV_prime.

- Offset bias is an integer whose value depends on the size of the hash but is roughly 1.0* 10^78 for for this implementation of the bloom filter, which hashes a 36-bit alphanumeric number.
- FNV Prime is a prime number seeded by an input number at initialization. Therefore, by seeding the hash function with different numbers at initialization, we can get distinct hash funcitons as the FNV_prime number is different.

The 2 terms multiplied constitute the hash that will be used on the data.The hash is then used on the data, and the result is returned.

Determining hash table size

The formula for probability collisions is the table size m divided by the number of items n, assuming a hash function that is effective in evenly distributing the entries. A locally run web scraper was created to find the total number of categories available in the Natural Language Processing API - 625. Only 10% of the avaliable categories are expected to be used, giving a n of 62.5. With a target collision rate of 1%, this was used to arrive at the hash table array size of 6250. The number of entities is arbitrary - all n stories under the same entities, leading to 16 total entities, or they could all have different entities, leading to $16n$ entities. Further analysis could help understand the optimal hash table size for entities - it is initated at 500 for now.

Max Heap

Analysis

A max heap was used because it was the most efficient data structure to get the maximum value for any given memory. It has a O(1) complexity to return the maximum value, and a O(logn) complexity to insert or delete any node, compared to sorted arrays which also have O(1) complexity to return the maximum value but have O(n) insertion time

Implementation

The heap class implemented (Copied below)has working push, delete and getMax methods, as demonstrated below

```
'''
#Commented out because it has already been implemented above, pasted here for
#reference

class maxHeap:

  def __init__(self):
    self.salienceValues = []


  def push(self, data):
    self.salienceValues.append(data)
    self.floatUp(len(self.salienceValues) - 1)

  def getMax(self):
    return self.salienceValues[0]

  def delete(self,data):
    index = self.salienceValues.index(data)

    if len(self.salienceValues) > 2:
      self.swap(index, len(self.salienceValues) - 1)
      returnValue = self.salienceValues.pop()
      self.bubbleDown(index)
    elif len(self.salienceValues) <= 2:
      returnValue = self.salienceValues.pop()
    return returnValue

  def swap(self, i, j):
    self.salienceValues[i], self.salienceValues[j] = self.salienceValues[j], self.s

  def floatUp(self, index):
    parent = index//2
    arrayInput = self.salienceValues

    if index <= 0:
      return
    elif arrayInput[index]["salienceValue"] > arrayInput[parent]["salienceValue"]:
      self.swap(index, parent)
      self.floatUp(parent)

  def bubbleDown(self, index):
    left = index * 2
    right = index * 2 + 1
    largest = index
    if len(self.salienceValues) > left and self.salienceValues[largest]["salienceVa
      largest = left
    if len(self.salienceValues) > right and self.salienceValues[largest]["salienceV
      largest = right
    if largest != index:
      self.swap(index, largest)
      self.bubbleDown(largest)

'''

import random

sampleCategoryMaxHeap = maxHeap()

'''
PUSH Method

5 items are pushed into the max heap
'''
print("PUSH")
print("Before:",sampleCategoryMaxHeap.salienceValues )
for i in range(5):
  dict = {}
  dict["Name"] = i
  dict["index"] = i
  dict["salienceValue"] = random.randint(1,40)
  sampleCategoryMaxHeap.push(dict)
print("After:", sampleCategoryMaxHeap.salienceValues,"\n")

'''
GET MAXIMUM Method

The tuple with the maximum salience Value is returned
'''
print("MAX VALUE")
maxValue = sampleCategoryMaxHeap.getMax()
print("Maximum:", maxValue,"\n")

'''
DELETE ELEMENT Method

An element is deleted from the tree
'''
print("DELETE")
print("Before:",sampleCategoryMaxHeap.salienceValues )
sampleCategoryMaxHeap.delete(dict)
print("After:", sampleCategoryMaxHeap.salienceValues)
```

```
PUSH
Before: []
After: [{'Name': 2, 'index': 2, 'salienceValue': 30}, {'Name': 3, 'index': 3,

MAX VALUE
Maximum: {'Name': 2, 'index': 2, 'salienceValue': 30}

DELETE
Before: [{'Name': 2, 'index': 2, 'salienceValue': 30}, {'Name': 3, 'index': 3
After: [{'Name': 2, 'index': 2, 'salienceValue': 30}, {'Name': 3, 'index': 3,
```

Performance

First, the performance was analyzed for the insert operation. Building a heap takes nlogn time using a top-down heapify method. However, this program utilizes a bottom-up heapify method where each element is inserted into a almost sorted existing max-heap.

The heap with n nodes has has $n/2$ leaf nodes that cannot be swopped down. At the next level, it has $n/4$ nodes that can be moved down at most once, thus querying the swap function twice. This progresses on till we get the Geometric progression shown below, where the term of the highest order is n and therefore the overall complexity is O(n) (Wilson, 2012)

```python
from google.colab import files
from IPython.display import Image
uploaded = files.upload()
Image("heapEqn.png", width=600)
```

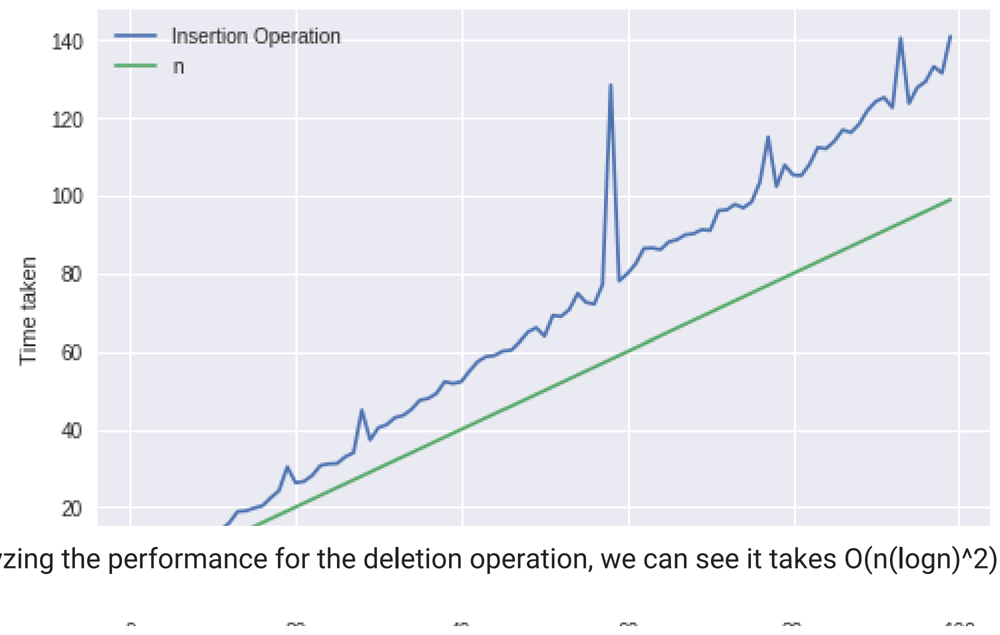
Choose Files heapEqn

- **heapEqn.png**(image/png) - 22732 bytes, last modified: 12/21/2018 - 100% done

```
Saving heapEqn.png to heapEqn.png
```

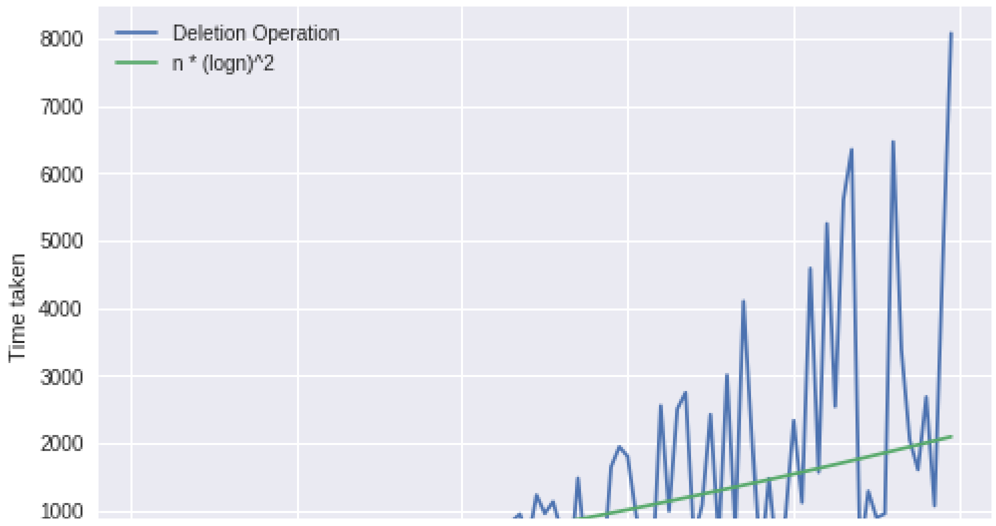
$$\frac{n}{2} + 2\frac{n}{4} + 3\frac{n}{8} + \cdots = n\left(\sum_{k=1}^{\log n} \frac{k}{2^k}\right)$$
$$= 2n - \log n - 2$$

```python
#Insert Operation


import matplotlib.pyplot as plt
import time
import math

performanceMeasurementMaxHeap = maxHeap()
timingData = []
xValues = []
complexityValues = []

for i in range(1,100):
  for k in range(1,10):
    averageTimer = []
    start_time = time.time()
    for j in range(1,i*10):
      dict = {}
      dict["Name"] = i
      dict["index"] = i
      dict["salienceValue"] = random.randint(1,40)
      performanceMeasurementMaxHeap.push(dict)
    averageTimer.append((time.time() - start_time))

  timingData.append((sum(averageTimer)/float(len(averageTimer))))
  xValues.append(i)
  complexityValues.append(i)

lowestI = timingData[0]
timingData[:] = [x / lowestI for x in timingData]

plt.plot(xValues, timingData, label = "Insertion Operation" )
plt.plot(xValues, complexityValues, label = "n" )
plt.legend()
plt.xlabel('Input Size')
plt.ylabel('Time taken, common sized')
plt.show()
```



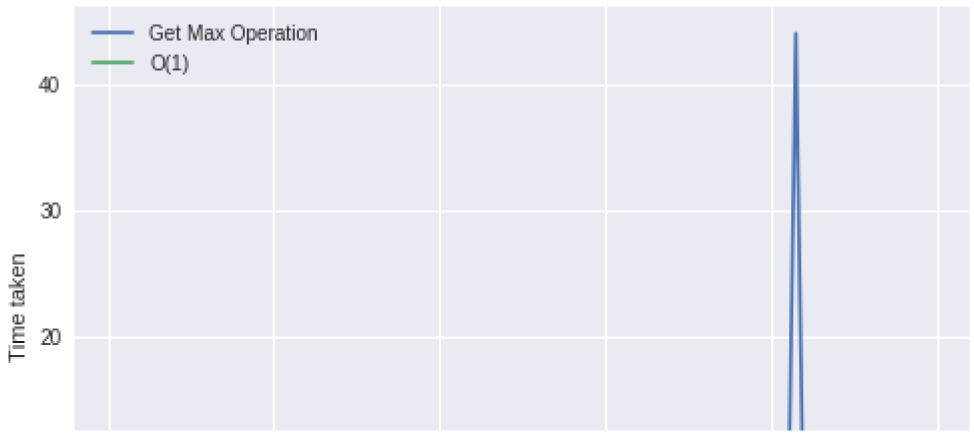
Analyzing the performance for the deletion operation, we can see it takes O(n(logn)^2).

```
#Deletion Operation


import matplotlib.pyplot as plt
import time
import math

DeletionPerformanceMeasurementMaxHeap = maxHeap()
timingData = []
xValues = []
complexityValues = []



for i in range(1,100):

  for j in range(1,10):
    averageTimer = []
    start_time = time.time()
    for k in range(0,i):
      dict = {}
      dict["Name"] = k
      dict["index"] = k
      dict["salienceValue"] = random.randint(1,40)
      DeletionPerformanceMeasurementMaxHeap.push(dict)

    start_time = time.time()
    DeletionPerformanceMeasurementMaxHeap.delete(dict)
    averageTimer.append((time.time() - start_time))

  timingData.append((sum(averageTimer)/float(len(averageTimer))))
  xValues.append(i)
  complexityValues.append(i*math.pow(math.log(i),2))
lowestI = timingData[0]
timingData[:] = [x / lowestI for x in timingData]

plt.plot(xValues, timingData, label = "Deletion Operation" )
plt.plot(xValues, complexityValues, label = "n * (logn)^2 " )
plt.legend()
plt.xlabel('Input Size')
plt.ylabel('Time taken, common sized')
plt.show()
```



Analyzing the performance for the getMax operation, we can see it has complexity of about O(1)

```
#Get Max Operation


import matplotlib.pyplot as plt
import time

getMaxPerformanceMeasurementMaxHeap = maxHeap()
timingData = []
xValues = []
complexityValues = []

for i in range(1,100):
  for k in range(1,10):
    averageTimer = []
    start_time = time.time()
    for j in range(0,i):
      dict = {}
      dict["Name"] = j
      dict["index"] = j
      dict["salienceValue"] = random.randint(1,40)
      getMaxPerformanceMeasurementMaxHeap.push(dict)

    start_time = time.time()
    getMaxPerformanceMeasurementMaxHeap.getMax
    averageTimer.append((time.time() - start_time))

  timingData.append((sum(averageTimer)/float(len(averageTimer))))
  xValues.append(i)
  complexityValues.append(1)

lowestI = timingData[0]
timingData[:] = [x / lowestI for x in timingData]

plt.plot(xValues, timingData, label = "Get Max Operation" )
plt.plot(xValues, complexityValues, label = "O(1)" )
plt.legend()
plt.xlabel('Input Size')
plt.ylabel('Time taken, common sized')
plt.show()
```



Overall Complexity Analysis and performance

There are 3 main opeartions - 1) hashing and insertion of category into the category tree hashtable, 2) hashing to find the correct entities in the entities hash table 3) Withdrawing stories with similar entities from the max heap at the given position of the entities hash table, and inserting the entities of the story into the max heap.

The operations with regards to the hash tables have O(1) average time complexity, and O(n) worst case complexity. The insertion and getMax() operation, on the other hand, have O(n) and O(1) average complexities, as shown earlier in the simulations, and have O(nlogn) and O(1) worst case complexities .

Although the 3 operations are nested, each of them is only performed n times per story. For example, although it takes O(n) worst-case complexity to find the correct category index in the CategoryTree hashtable and within the category it takes O(n) worst-case complexity to find the correct category index in the Entiteis hashtable, the latter operation is only performed once. Therefore, to find the overall complexity, we do not multiply the complexities but instead add them. The overall average case complexity of the operations is therefore O(n), taking the largest of the average complexities of the 3 operations, and the overall worst-case complexity is O(nlogn), taking the largest of the worst complexities of the 3 operations. Complexities of operations such as delete were explored but are not expected to run frequently.

Conclusion

This paper, using hash tables, trees and max-heaps in conjunction with the Google Natural Language API, has provided a preliminary way in which users can discover content through self.expression. The test set worked well, with a large majority of the stories returned directly relevant to the story used as input.

However, to extend this project, there should be more data to build a better and more customized training model. Diary-writing tends to have texts that are centered around emotional topics, and the majority of topics that are under Google's Natural Language API are irrelevant. However, these categories can be used to encourage self-expression beyond emotional self-expression, in the form of intellectual self-expression. It would be best that for each of these fields, a new model is trained to give more precise categories, so as to increase the accuracy and relevance of the posts returned, as well as to reduce runtime, which would be longer if many stories are clumped together in few categories and hence the height of the max heaps in each of these categories is high, leading to longer insertion times.

Lastly, the input to this algorithm - text- can be created in other ways other than the user manually writing them. Machine learning and big data pulled from users' online habits, with their explicit consent, could be used to create a picture of their current circumstances, and this could be converted into text that could be fed into the program. I intend to do continue this part of the project in future courses, to build towards a capstone where I create a new model of content discovery where the content is meaningful and touching.

###HCs

#distributions - A distribution of the performance of various components of the algorithm at different inout sizes is created, compared to the theoretical estimate as well as discussed

#probabiity - Probability of collisions was applied as a variable to come up with the ideal hash table size

#audience - The target end user - social-media-savvy youth, is studied carefully to come up with a algorithm that can best add value to their lives.

LOs

#novelapplication - A novel combination of the Google API, heaps, trees and hash tables was used to create a working prototype to solve an interesting social problem

#optimalalgorithm - The merits of using particular data structures was discussed in depth, and the effectiveness of these choices was shown with simulations and theoretical analysis when relevant.

#complexity - The theoretical complexity of various stages of the algorithm, as well as the algorithm as a whole, is discussed, the underlying reasons are explained and the complexity is tested through simulations.

#hashing - A working hash function is implemeneted, and the benefits and drawbacks of using hash functions are discussed. Furthermore, the reasons behind the use of hash functions, their mechanisms as well as their performance are explored. Open addressing is implemented where necessary and explained.

#searchtrees - A working max heap implementation that stores objects is is provided, and its benefits and drawbacks are discussed. Furthermore, a discussion of the reasons behind the use of max heaps, the benefits and drawbacks as well as a simulation of its performance is conducted

References

Wilson, T. (2012, October 1). Inserting an element in a heap takes O(log n). Still if we insert n elements in the heap it comes out to be O(n)? Retrieved December 21, 2018, from https://www.quora.com/Inserting-an-element-in-a-heap-takes-O-log-n-Still-if-we-insert-n-elements-in-the-heap-it-comes-out-to-be-O-n