Author: Carlos Fernando Castaneda

Class: CS 2302

Date Modified: November 21, 2019

Instructor: Diego Aguirre

Assignment: Lab 5 LRU Cache

TA: Gerardo Barraza

**Introduction:**

In programming, caches are optimizing instructions that a computer can utilize in order to manage a data of information stored on the computer. This can improve performance by keeping recent or often-used data items in memory locations that are faster to access than normal memory stores. In this lab I must solve two problems, problem one involves implementing a particular cache named LRU (Least Recently Used), which is a cache that discards the least recently used items first, including some operations for the algorithm. The second problem, I must implement a Max Heap to print the most frequent elements appeared in a list of words.

**Proposed solution design and implementation:**

Before I begin, I want to note that the code from this heap was taken from the resources page in blackboard. I will alter some lines and methods, but essentially, I wanted to use a resource given to lessen the workload on the lab.

The implementation of this lab is pretty straightforward for me, since the lab sheet already gives me enough directions to know what to do. For the LRU method, I have to first define the constructors for the cache since its necessary for it anyways. I also need to find the required methods asked from the lab sheet, including get(), which returns the value of an existing key, put(), which inserts a new value of a given key, size(), which returns the size, and max\_capacity(), which returns the max capacity of the cache. I also must find a way to make all of the methods run at O(1) time.

For the heaps, like mentioned above, I will implement the code given to us from the class resource page. However, I will add necessary precautions so that extracts the maximum string, as well as having the ability to read a string instead of an integer.

Lastly, I want to add another section on the opening menu where the user can either problems from the lab. This will be very helpful to me later as it will help me determine the run times of the separate classes that will be included in my program.

**Experimental results**:

Before I begin explaining my results, like in the last lab, I wanted to note that I split my program into three separate files, one for the main method, one for the LRU Cache, and one for the max heaps implementation. I mainly had to do this due to the fact that the two problems have different Node functions, and I found that my implementation was simpler if I just split the situations into their own class, and I just imported them into the main program like how the professor did in our midterm 1 programs.

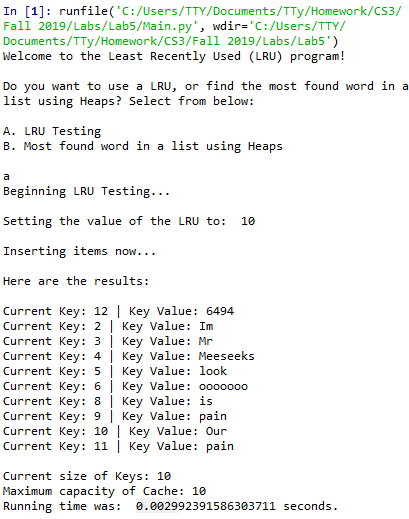
The implementation of this lab is pretty straightforward for me, since the lab sheet already gives me enough directions to know what to do. For the LRU method, I first defined the constructors for the cache to have an idea on what altercations I had to do later on the algorithm. For the get() method, I needed for it to return -1 if the key didn’t exist, as that was required from the lab sheet, as well as having it iterate until the given key was found. For the method put(), I decided to follow exactly what the link in the lab sheet said, and it guided me to be the “heart” of the cache, as most of the LRU commands follow from this method. For size() and max\_capacity(), I decided to have these values set in the constructor before the program runs, and the particular methods just returns the value that corresponds to it. I also made the methods so that no matter what they could at O(1) time.

For the heaps, like mentioned above, I implemented the code given to us from the class resource page. However, I added some altercations including that the entirety of the code works with strings instead of integers. I also added a percolate down method that works exclusively for extracting the maximum integer of a list. I also added some methods that read a text file, so they can import a set of integers to be evaluated for problem 2. I also decided to bring in a print method that not only prints the required strings, but also prints out the times that word has appeared.

As mentioned above, I added a section in the user interface that will allow them to access either problem from the lab by either pressing a for the LRU cache, or b for the heaps. As for the run times, I have included the results at the end of the document before the conclusion.

Below you will find the results from various states in the program (option a is the LRU program, while option b is the heap) …

My results from running option a:

****

My results from running option B, using ‘test.txt’ as my file for the list:

**A screenshot of a social media post

Description automatically generated**

**Run times (Binary Search Trees):**

|  |  |  |
| --- | --- | --- |
| **Time** | **LRU Cache** | **Heap** |
| **1** | **0.0029 seconds** | **16.0015 seconds** |
| **2** | **0.0009 seconds** | **7.1851 seconds** |
| **3** | **0.0009 seconds** | **6.0672 seconds** |

**Graph representation:**

Obviously looking at the data you can see that the cache runs way faster than the heap, but in all honesty, that wasn’t the intent of the lab. I thought it would be appropriate to shoe the running times of both methods not for comparison, but to prove that the methods of the cache ran at O(1) time, which was a requirement of this lab.

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using multiple python files to create a cache algorithm LRU, as well as using a heap to find the most frequent words form a given list. I was able to implement algorithms that run in O(1) for the cache, as well as using the percolate methods for the heap.

**Appendix :**

**Main.py**

'''

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: November 14, 2019

Instructor: Diego Aguirre

Assingment: Lab 5 LRU (1 0f 3)

TA: Gerardo Barraza

Purpose: To practice using queues and

hash tables to create a LRU cache.

'''

#Used to calculate the time for each option on the lab

import time

#Imports the LRU cache and the heap programs to test out the lab

import LRU\_Cache as lru

import heaps as heap

#Method that

def LRU\_Test():

int = 10

print("Setting the value of the LRU to: ",int)

print("")

new\_lru = lru.LRU(int)

print("Inserting items now...")

print("")

new\_lru.put(1, "Hi")

new\_lru.put(2, "Im")

new\_lru.put(3, "Mr")

new\_lru.put(4, "Meeseeks")

new\_lru.put(5, "look")

new\_lru.put(6, "at")

new\_lru.put(8, "mee")

new\_lru.put(9, "6712")

new\_lru.put(10, "Our")

new\_lru.put(11, "very")

new\_lru.put(11, "existence")

new\_lru.put(8, "is")

new\_lru.put(9, "pain")

new\_lru.put(6, "pain")

new\_lru.put(11, "pain")

new\_lru.put(6, "ooooooo")

new\_lru.put(12, "6494")

print("Here are the results: ")

print("")

new\_lru.print\_inorder()

def main():

print("Welcome to the Least Recently Used (LRU) program!")

print("")

print("Do you want to use a LRU, or find the most found word in a list using Heaps? Select from below:")

print("")

print("A. LRU Testing")

print("B. Most found word in a list using Heaps")

user\_selection = input()

count = 0

if (user\_selection == "A" or user\_selection == "a"):

print("Beginning LRU Testing...")

print("")

start1 = time.time()

LRU\_Test()

end1 = time.time()

print('Running time was: ', end1 - start1, 'seconds.')

elif (user\_selection == "B" or user\_selection == "b"):

print("Beginning Heap Search...")

start2 = time.time()

print("")

print("Please input the filename you want to input for the heap:")

filename = input()

heapWord, dict = heap.MaxHeap.file(filename)

heapSize = heapWord.size()

print("")

print("There are currently", heapSize, "words in the heap. How many words of that heap do you want to see printed?")

printed = int(input())

heapWord.print\_descending(printed)

print("")

end2 = time.time()

print('Running time was: ', end2 - start2, 'seconds.')

elif count == 3:

print("ERROR: Too many incorrect statements!")

print("Farewell.")

else:

print("ERROR: Input not valid!")

count+=1

main()

main()

**heaps.py**

'''

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: November 14, 2019

Instructor: Diego Aguirre

Assingment: Lab 5 LRU (3 0f 3)

TA: Gerardo Barraza

Purpose: To practice using queues and

hash tables to create a LRU cache.

'''

#Used to return error messages to the user if the need arises

import math

# This class in the program is used to create objects of a max Heap

class MaxHeap(object):

def \_\_init\_\_(self):

self.tree = []

#Will return if its empty or not

def is\_empty(self):

return len(self.tree) == 0

#Will return the index of the parent

def parent(self, i):

if i == 0:

return -math.inf

parent\_list = self.tree[(i - 1) // 2]

return parent\_list[1]

#Will return the index of the right child

def right\_child(self, i):

c = 2 \* i + 2

if c >= len(self.tree):

return -math.inf

c\_list = self.tree[c]

return c\_list[1]

#Will return the index of the left child

def left\_child(self, i):

c = 2 \* i + 1

if c >= len(self.tree):

return -math.inf

c\_list = self.tree[c]

return c\_list[1]

#Will insert the new item and percolate up so that it balances the heap

def insert(self, item):

self.tree.append(item)

self.percolate\_up(len(self.tree) - 1)

#Method that helps balance of the heap correctly by swapping items until the correct order is met while also creating the correct placementy of each item.

def percolate\_up(self, i):

if i == 0:

return

#Sets the parent index and the value of i to compare in the percolation

indexParent = (i - 1) // 2

parent\_val = self.tree[indexParent]

i\_value = self.tree[i]

#Checks if the count equals the value

if parent\_val[1] == i\_value[1]:

parent\_word = list(parent\_val[0])

i\_word = list(i\_value[0])

iteration = min(len(i\_word), len(parent\_word))

#While iterating, it will check which words replaces uppercase letters with lowercase letters.

for i in range(iteration):

i\_letter = self.caps(i\_word[i])

parent\_letter = self.caps(parent\_word[i])

#If the letter is greater than its parent, then it swaps

if i\_letter > parent\_letter:

i\_value[0], parent\_val[0] = parent\_val[0], i\_value[0]

i\_value[1], parent\_val[1] = parent\_val[1], i\_value[1]

self.percolate\_up(indexParent)

#Otherwise it will return blank

elif i\_letter < parent\_letter:

return

#Checks if the count is less than the value

if parent\_val[1] < i\_value[1]:

i\_value[0], parent\_val[0] = parent\_val[0], i\_value[0]

i\_value[1], parent\_val[1] = parent\_val[1], i\_value[1]

self.percolate\_up(indexParent)

#Method that helps balance of the heap correctly by swapping items until the correct order is met while also creating the correct placementy of each item.

def percolate\_down(self, i):

#Sets the value of i to traverse the heap.

i\_value = self.tree[i]

#If the i value is less than the max of the left and right child, return blank

if i\_value[1] >= max(self.left\_child(i), self.right\_child(i)):

return

#If the i value is greater than the right child, set the index to the left child, otherwise set the index to the right child

if self.left\_child(i) > self.right\_child(i):

max\_child\_index = 2 \* i + 1

else:

max\_child\_index = 2 \* i + 2

#Sets a list of max values through iterate again

max\_list = self.tree[max\_child\_index]

i\_value[0], max\_list[0] = max\_list[0], i\_value[0]

i\_value[1], max\_list[1] = max\_list[1], i\_value[1]

self.percolate\_down(max\_child\_index)

#Method that finds the max value of a heap.

def extractMax(self):

#If the length of the heap is less than 1, return none.

if len(self.tree) < 1:

return None

#If the length of the heap is 1, pop the item on the tree.

if len(self.tree) == 1:

return self.tree.pop()

#Sets the root of the tree, and later pops the number as well as percolating down the heap.

root = self.tree[0]

self.tree[0] = self.tree.pop()

self.percolate\_down(0)

#Returns the root value

return root

#Takes the filename given by the user and inserts all of the words into a given heap

def file(filename):

#Creates a new heap and dictionary to place items inside.

dict = {}

heapWord = MaxHeap()

#While the file is open, it adds each word into the dictionary.

with open(filename, encoding='windows-1252') as textFile:

#For every line in the file, it will grab every word.

for line in textFile:

string = line.split()

word = str(string)

word\_string = word[2:len(word) - 2]

#If the word is in the dictionary, it will add to the count variable if duplicates of a word are found in the file.

if word\_string in dict:

count = dict.get(word\_string) + 1

update\_pair = {word\_string : count}

dict.update(update\_pair)

else:

dict[word\_string] = 1

#Inserts the info from the dictionary into the heap

for key in dict:

heapWord.insert([key, dict[key]])

#Returns the heap back to the user

return heapWord, dict

#Method activated by make key that will find the lower case letter by comparing it to its caps counterpart

def caps(self, char):

cap\_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'N', 'M', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

lower\_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'n', 'm', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

lower\_char = ''

for i in range(len(cap\_list)):

if (char == cap\_list[i]):

lower\_char = lower\_list[i]

return lower\_char

return char

#Method that returns the size of the heap

def size(self):

return len(self.tree)

#Prints the items of a heap in descending order

def print\_descending(self, num\_of\_times):

count = 0

if (len(self.tree) < num\_of\_times):

print("Invalid number. Try again")

count+=1

if count == 3:

print("ERROR: Too many incorrect statements!")

print("Farewell.")

#Sets the first word automatically in case there is no most common word

most\_common\_word = self.tree[0]

#Iterates through the whole heap to find the max

for i in range(num\_of\_times):

word = self.extractMax()

print("Current Word:", word[0], "|", "Number of times:", word[1])

print("")

print("Result: ")

print("The most seen word was :", most\_common\_word[0])

print("This word appeared", most\_common\_word[1],"number of times.")

print("")

**LRU\_Cache.py**

*'''*

*Author: Carlos Fernando Castaneda*

*Class : CS 2302*

*Date Modified: November 14, 2019*

*Instructor: Diego Aguirre*

*Assingment: Lab 5 LRU (2 0f 3)*

*TA: Gerardo Barraza*

*Purpose: To practice using queues and*

*hash tables to create a LRU cache.*

*'''*

*#Creates the node values necessary*

*class Node(object):*

*def \_\_init\_\_(self, key, val, prev = None, next = None, empty = False): #Constructor for the node (doubly linked list node).*

*self.key = key*

*self.val = val*

*self.prev = prev*

*self.next = next*

*self.empty = empty*

*#Creates the LRU\_Cache class necessary to perform the operations for problem 1*

*class LRU(object):*

*def \_\_init\_\_(self, max\_capacity, size = 0, head = None, tail = None, table = {}): #Constructor for LRU. Set up like a doubly linked list with a hash table.*

*self.size = size*

*self.head = head*

*self.tail = tail*

*self.table = table*

*self.max\_capacity = max\_capacity*

*#Method that creates the nodes, the necessary linked list, and populates a new hash table to traverse through the cache.*

*def put(self, key, val):*

*#If list is empty, it will create a new node and make it its head.*

*if (self.head is None):*

*self.head = Node(key, val)*

*self.tail = self.head*

*self.table[key] = self.head*

*self.size += 1*

*return*

*#If the key is already in the table, then it overrides the key values.*

*if (key in self.table):*

*if(key == self.head.key):*

*self.head.val = val*

*node = self.table.get(key)*

*node.val = val*

*return*

*#If the size is at its maximum capacity or greater.*

*if (self.size >= self.max\_capacity):*

*#If the tail node is none, it then overrides the head.*

*if (self.tail.next is None):*

*self.head.key = key*

*self.head.val = val*

*node = Node(key, val)*

*self.table[key] = node*

*self.tail = self.head*

*return*

*#Otherwise, it overrides the next node that needs to be overridden.*

*else:*

*node = self.tail.next*

*node.key = key*

*node.val = val*

*self.tail = node*

*self.table[key] = node*

*return*

*#If the list is not at its full capacity, it adds the nodes to the dictionary.*

*node = Node(key, val)*

*self.tail.next = node*

*node.prev = self.tail*

*self.tail = node*

*self.table[key] = node*

*self.size = self.size + 1*

*return*

*#Uses a dictionary to get a value of the key if it currently exists in the cache.*

*def get(self, key):*

*#If the current head is none, it will return -1, indicating that its empty.*

*if (self.head is None):*

*return -1*

*#Sets the current node to the current key*

*curr\_node = self.table.get(key)*

*#If the node is None, it will return -1, indicating there is nothing*

*if curr\_node == None:*

*return -1*

*#Returns the value of the current node*

*return curr\_node.value*

*#Prints the list in order of the LRU cache until it reaches the end of the list.*

*def print\_inorder(self):*

*#If the head is None, it will return an error message.*

*if self.head is None:*

*print("ERROR! The current head is None. Farewell.")*

*return*

*#Sets an iterative value for the current head*

*iter = self.head*

*#While the ieration is not none, it will print the iterative key and its value for each key.*

*while(iter != None):*

*print("Current Key:", iter.key, "|","Key Value:", iter.val)*

*iter = iter.next*

*print("")*

*print("Current size of Keys:", self.size)*

*print("Maximum capacity of Cache:", self.max\_capacity)*

*#Returns the number of key/value pairs that are currently stored in the cache.*

*def size(self):*

*return self.size*

*#Returns the maximum capacity that is currently applied in the cache.*

*def max\_capacity(self):*

*return self.max\_capacity*

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provide inappropriate assistance to any student in the class.

