**Due: November 26, 2023**

**Assignment Completion**

In order to for this assignment to be considered completed you must submit:

* a completed analysis of functions listed for (part A)
* a successful run for part C (green check for part C in actions)

Note: Assignment completion is the minimum requirements for your assignment. It does not mean you will receive full marks.

**Assignment Objectives:**

In this assignment, you will:

* Complete an analysis of multiple related functions for a class
* Implement hash tables

**Restrictions**

**As this assignment is about implementing data structures, you are not allowed to make use of any python libraries or use builtin python data structures and functions unless otherwise stated.**

**Overview**

In this assignment you will look at several implementations of a Table. There are three tasks:

1. Analyze the member functions of the class SortedTable (code for this is provided below). This table is created using a sorted list. It is not necessarily implemented well. You will analyze the listed functions.
2. Offer suggestions on how to make the SortedTable more efficient. After analyzing the functions, look at how it is done and come up with some changes that will make the code more efficient
3. Implement a Hash table
   * using chaining for collision resolution
   * using linear probing for collision resolution

**Table functions overview**

We will explore 3 ways to implement a Table. Each method will have a different internal structure but the functionality will be the same. A table stores a set of key-value pairs which are also referred to as ***records***

The following specification describes an overview of the general functionalities of the table but it does not specify any specifics in terms of implementation. In otherwords, it describes what it can do but doesn't specify any internal data strctures

def \_\_init\_\_(self, capacity=32):

The initializer for the table defaults the initial table capacity to 32. It creates a list with capacity elements all initialized to None.

def insert(self, key, value):

This function adds a new key-value pair into the table. If a record with matching key already exists in the table, the function does not add the new key-value pair and returns False. Otherwise, function adds the new key-value pair into the table and returns True. If adding a record will cause the table to be "too small" (defined in each class), the function will grow to acommodate the new record.

def modify(self, key, value):

This function modifies an existing key-value pair into the table. If no record with matching key exists in the table, the function does nothing and returns False. Otherwise, function modifies the existing value into the one passed into the function and returns True.

def remove(self, key):

This function removes the key-value pair with the matching key. If no record with matching key exists in the table, the function does nothing and returns False. Otherwise, record with matching key is removed and returns True

def search(self, key):

This function returns the value of the record with the matching key. If no record with matching key exists in the table, function returns None

def capacity(self):

This function returns the number of spots in the table. This consists of total spots available in the table.

def \_\_len\_\_(self):

This function returns the number of Records stored in the table.

**Part A: Analyze the listed member functions of the SortedTable (40%)**

Analyze the following functions with respect to the number of Records stored in the SortedTable

def insert(self, key, value):

def modify(self, key, value):

def remove(self, key):

def search(self, key):

def capacity(self):

def \_\_len\_\_(self):

**Note: the code below is not necessarily the best way to write a table using a sorted array. Outside of syntactic constructs, do not actually use this code for anything... its has a lot of inefficiencies.**

class SortedTable:

# packaging the key-value pair into one object

class Record:

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

def \_\_init\_\_(self, cap=32):

# this initializes a list of of capacity length with None

self.the\_table = [None for i in range(cap)]

self.cap = cap

def insert(self, key, value):

if (self.search(key)!=None):

return False

if(len(self) == self.cap):

# increase the capacity if list is full

new\_table = [None for i in range(self.cap\*2)]

for i in range(self.cap):

new\_table[i]=self.the\_table[i]

self.the\_table = new\_table

self.cap \*= 2

self.the\_table[len(self)]=self.Record(key,value)

size = len(self)

for i in range (0,size-1):

for j in range(0,size-1-i):

if(self.the\_table[j].key>self.the\_table[j+1].key):

tmp=self.the\_table[j]

self.the\_table[j]=self.the\_table[j+1]

self.the\_table[j+1]=tmp

return True

def modify(self, key, value):

i = 0

while (i < len(self) and self.the\_table[i].key != key):

i+=1

if(i==len(self)):

return False

else:

self.the\_table[i].value = value

return True

def remove(self, key):

i = 0

size = len(self)

while (i < size and self.the\_table[i].key != key):

i+=1

if(i==size):

return False

while(i+1 < size):

self.the\_table[i]=self.the\_table[i+1]

i+=1

self.the\_table[i] = None

return True

def search(self, key):

i = 0

size = len(self)

while i < size and self.the\_table[i].key != key :

i+=1

if i==size:

return None

else:

return self.the\_table[i].value

def capacity(self):

return self.cap

def \_\_len\_\_(self):

i =0

count = 0

while(i < len(self.the\_table)):

if(self.the\_table[i]!=None):

count+=1

i+=1

return count

**Part B: Suggestions (10% bonus)**

Suggest 2 improvements you could make to the code that will improve its efficiency. State which function(s) would be improved by the suggested improvement. This is not a coding question. You do not need to implement your suggestion. A clear description of what you want to do is good enough.

**Which improvements counts and which do not**:

* You can't change the underlying data structure. For example, "make it a hash table" would make it something different so that won't count. Fundamentally it must use a sorted python list as the underlying data structure
* A change only counts once: "Do a selection sort instead of what is written in the **len**() function" and "Do a selection sort instead of what is written in the capacity() function" is just one suggestion not two. (note this suggestion is silly, and just used as an example)

**Part C Implementation of ChainingTable *OR* LinearProbingTable (60%)**

Choose either chaining or linear probing. You don’t have to implement both!

When doing this coding portion of the assignment you can use:

* python lists
* your assignment 1 linked list
* python hash() function - don't write your own

A hash table places records based on a hash value that you get from a hash function. We will use the built in hash function in python for this part of the assignment. A hash function will return a really big number when given something to hash. We will use this function to hash the key of our key-value pair (not the value, just the key).

Example usage:

x = hash("hello world") # x will be a number with many digits, possibly negative.

cap = 32

idx = x % cap # idx is guaranteed to be a value between 0 and 31 inclusive

# because the mod operator guarantees that when you say x % n

# the result is always between 0 and n-1 inclusive

You will implement two hash tables for this assignment. One will use linear probing for collision resolution, the other will use chaining. You can use your assingment 1 linked list for chaining table if you wish.

**ChainingTable:**

A ChainingTable is a hash table which uses chaining as its collision resolution method.

def \_\_init\_\_(self, capacity=32):

The initializer for the table defaults the initial table capacity to 32. It creates a list with capacity elements all initialized to None.

def insert(self, key, value):

This function adds a new key-value pair into the table. If a record with matching key already exists in the table, the function does not add the new key-value pair and returns False. Otherwise, function adds the new key-value pair into the table and returns True. If adding the new record causes the load factor to exceed 1.0, you must grow your table by doubling its capacity

def modify(self, key, value):

This function modifies an existing key-value pair into the table. If no record with matching key exists in the table, the function does nothing and returns False. Otherwise, function modifies the changes the existing value into the one passed into the function and returns True

def remove(self, key):

This function removes the key-value pair with the matching key. If no record with matching key exists in the table, the function does nothing and returns False. Otherwise, record with matching key is removed and returns True

def search(self, key):

This function returns the value of the record with the matching key. If no reocrd with matching key exists in the table, function returns None

def capacity(self):

This function returns the number of spots in the table. This consists of spots available in the table.

def \_\_len\_\_(self):

This function returns the number of Records stored in the table.

**LinearProbingTable:**

A LinearProbingTable is a hash table which uses linear probing as its collision resolution method. You can either use the tombstone method or the non-tombstone method. The choice is yours.

def \_\_init\_\_(self, capacity=32):

The initializer for the table defaults the initial table capacity to 32. It creates a list with capacity elements all initialized to None.

def insert(self, key, value):

This function adds a new key-value pair into the table. If a record with matching key already exists in the table, the function does not add the new key-value pair and returns False. Otherwise, function adds the new key-value pair into the table and returns True. If adding the new record causes the load factor to exceed 0.7, you must grow your table by doubling its capacity

def modify(self, key, value):

This function modifies an existing key-value pair into the table. If no record with matching key exists in the table, the function does nothing and returns False. Otherwise, function modifies the changes the existing value into the one passed into the function and returns True

def remove(self, key):

This function removes the key-value pair with the matching key. If no record with matching key exists in the table, the function does nothing and returns False. Otherwise, record with matching key is removed and returns True

def search(self, key):

This function returns the value of the record with the matching key. If no reocrd with matching key exists in the table, function returns None

def capacity(self):

This function returns the number of spots in the table. This consists of spots available in the table.

def \_\_len\_\_(self):

This function returns the number of Records stored in the table.

**Submission:**

In order to for this assignment to be considered completed you must submit:

* the analysis for part A (placed in the a2.txt or doc)
* a successfuly running code for part C (submitted as a2code.txt)

Part B is optional for completion but completing part B has bonus marks

Please make sure you follow the steps listed below fully.