Practical Assignment

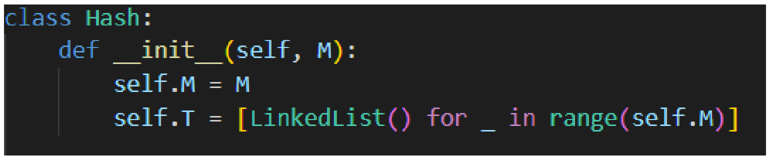
BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

Present the structure of the hash table.

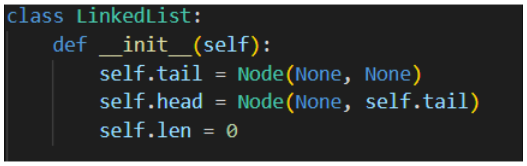
The Hash class takes in one integer value which is the size of the hash table. M stores this integer value this is one of the Hash classes properties. The other property T, the table of linked lists, will be initialized by M when the object is created. The table is M sized.



```
class Hash:
    def __init__(self, M):
        self.M = M
        self.T = [LinkedList() for _ in range(self.M)]
```

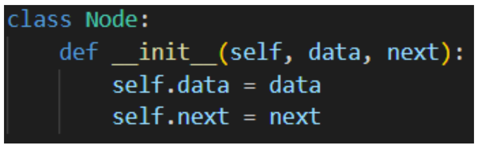
LinkedList class has three properties tail pointer which points to the last node of the linked list, head pointer which points to the first node and len which stores the length of the list which is updated when adding or removing nodes from linked list.

Initially tail points to a node which contains None and head points to a node which has no value but has a link to tail pointer.



```
class LinkedList:
    def __init__(self):
        self.tail = Node(None, None)
        self.head = Node(None, self.tail)
        self.len = 0
```

The Node class takes in two values data and next. Data is the value that the node has and next is a pointer to the next node.



```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

1.2 Hash function

What hashing function you used and why?

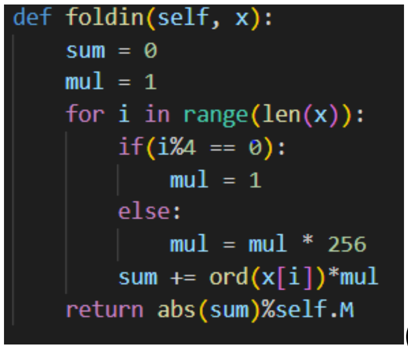
I used remainder hashing for integers because it was a simple and effective way to hash them, and the distribution was good enough. So, I divide by the integer value with the size of the table and then the remainder of this division is the hash value.

I use string folding function for strings because it balances out the table well compared to other hash functions like summing the ascii values of the characters and calculating the remainder from there. This is because when the table is bigger like in hash_3.1 the distribution is more even with string folding.

1.3 Methods

What methods (including the required) your hash table has? Explain briefly how do they work?

- foldin

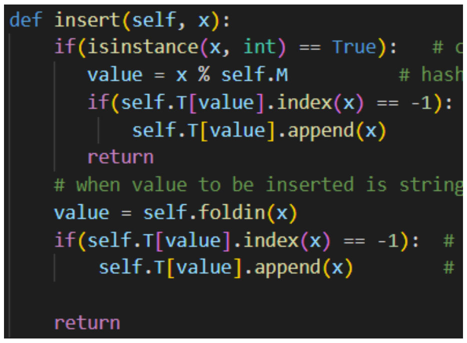


```python
def foldin(self, x):
    sum = 0
    mul = 1
    for i in range(len(x)):
        if(i%4 == 0):
            mul = 1
        else:
            mul = mul * 256
        sum += ord(x[i])*mul
    return abs(sum)%self.M
```

(picture 1)

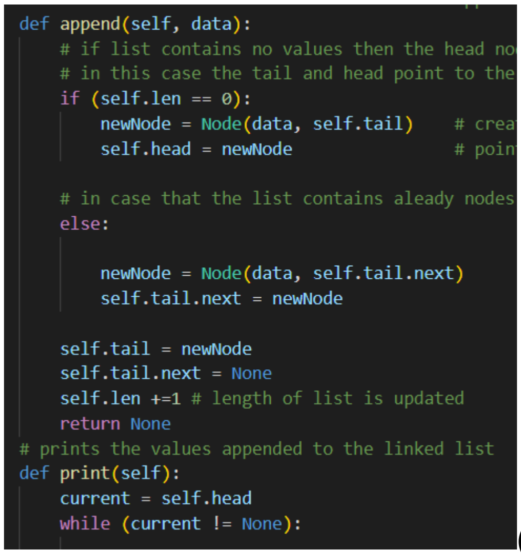
The foldin method is the string folding function used to hash strings in the Hash class. It takes in a string which then processes it four bytes at a time that creates long integer values that are summed together in the end. This sum is the divided by the size of the table and the remainder of this which is the hash value is returned as the output. I used the openDSA course material to help to do this function.

- insert



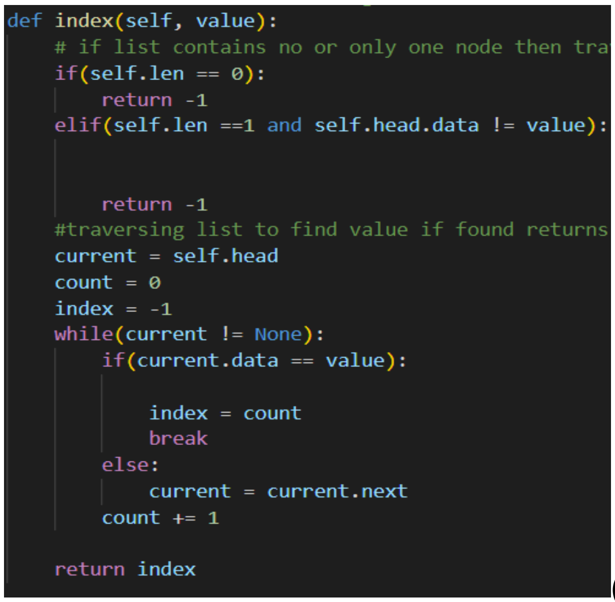
```python
def insert(self, x):
    if(isinstance(x, int) == True):   # c
        value = x % self.M          # hash
        if(self.T[value].index(x) == -1):
            self.T[value].append(x)
        return
    # when value to be inserted is string
    value = self.foldin(x)
    if(self.T[value].index(x) == -1):  #
        self.T[value].append(x)        #

    return
```

(picture 2)



```python
def append(self, data):
    # if list contains no values then the head no
    # in this case the tail and head point to the
    if (self.len == 0):
        newNode = Node(data, self.tail)    # crea
        self.head = newNode                # poin

    # in case that the list contains aleady nodes
    else:

        newNode = Node(data, self.tail.next)
        self.tail.next = newNode

    self.tail = newNode
    self.tail.next = None
    self.len +=1 # length of list is updated
    return None
# prints the values appended to the linked list
def print(self):
    current = self.head
    while (current != None):
```

(picture 3)

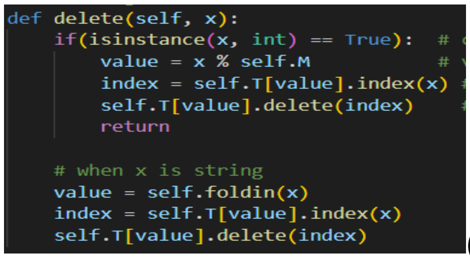


```python
def index(self, value):
    # if list contains no or only one node then tra
    if(self.len == 0):
        return -1
    elif(self.len ==1 and self.head.data != value):


        return -1
    #traversing list to find value if found returns
    current = self.head
    count = 0
    index = -1
    while(current != None):
        if(current.data == value):

            index = count
            break
        else:
            current = current.next
        count += 1

    return index
```

(picture 4)

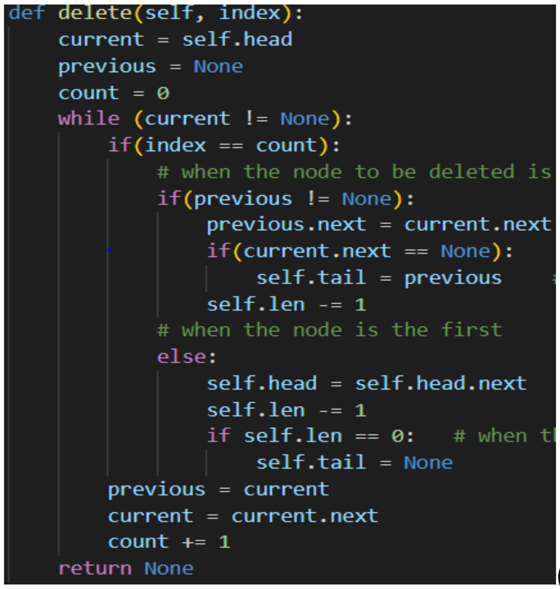
The insert function takes the value to be inserted and checks whether the value an integer with isinstance function which returns a boolean value. If the value is an integer, then it is hashed by taking the remainder of the integer and the self.M which is the size of the hash table. If the value is a string then insert uses the foldin(see pic 1) function which uses string folding technique to hash the value. Once we have the hash value then we go to that index in the hash table. From there we use the LinkedList classes index(pic 4) method which returns -1 if the value is not already in the list. This way, we do not get duplicate values in the table. If the value doesn’t exist, we use the LinkedList method append(pic 3) which appends the value to the end of the linked list object.

- delete



```
def delete(self, x):
    if(isinstance(x, int) == True):   # c
        value = x % self.M            # v
        index = self.T[value].index(x) #
        self.T[value].delete(index)    #
        return

    # when x is string
    value = self.foldin(x)
    index = self.T[value].index(x)
    self.T[value].delete(index)
```

(picture 5)

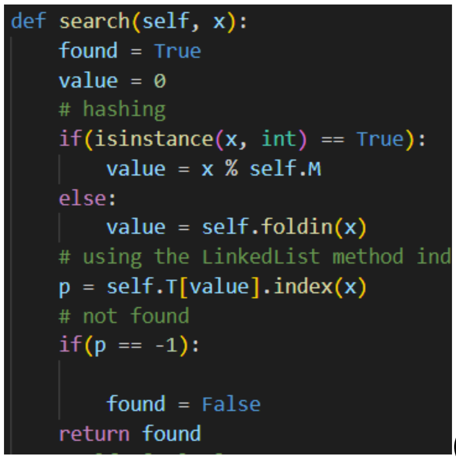


```
def delete(self, index):
    current = self.head
    previous = None
    count = 0
    while (current != None):
        if(index == count):
            # when the node to be deleted is
            if(previous != None):
                previous.next = current.next
                if(current.next == None):
                    self.tail = previous    #
                self.len -= 1
            # when the node is the first
            else:
                self.head = self.head.next
                self.len -= 1
                if self.len == 0:   # when t
                    self.tail = None
        previous = current
        current = current.next
        count += 1
    return None
```

(picture 6)

The Hash classes Delete method takes the value to be removed as an input and returns nothing. To locate the value, it is hashed exactly as in the insert method explained above. When we have the hashed value, we then find the index in which the value is in the linked list using the index method of the LinkeList class (pic 4). When we have the index at where the value is found in the list, we use the LinkedLists delete method (pic 6) which takes the index as an input and traverses the list until it finds the node at the ´correct index. From there we update the linked lists tail pointer and depending on the list size the head pointer too, we also update the list size of the linked list.

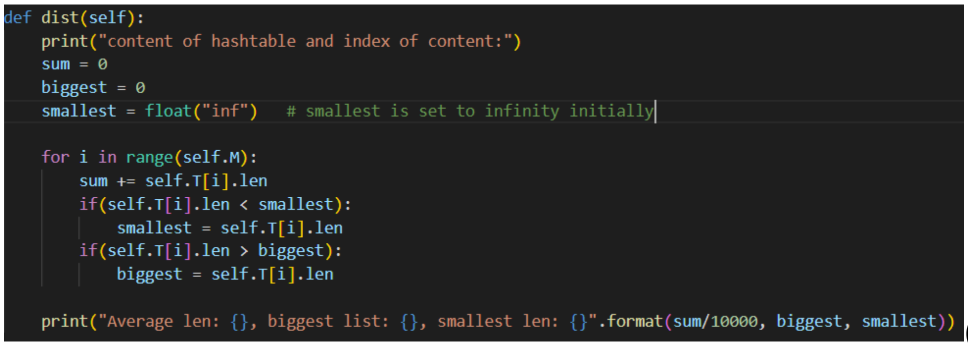
- search


```python
def search(self, x):
    found = True
    value = 0
    # hashing
    if(isinstance(x, int) == True):
        value = x % self.M
    else:
        value = self.foldin(x)
    # using the LinkedList method ind
    p = self.T[value].index(x)
    # not found
    if(p == -1):

        found = False
    return found
```
(picture 7)

The search method returns True if the searched value is found in the table and False if it is not found.

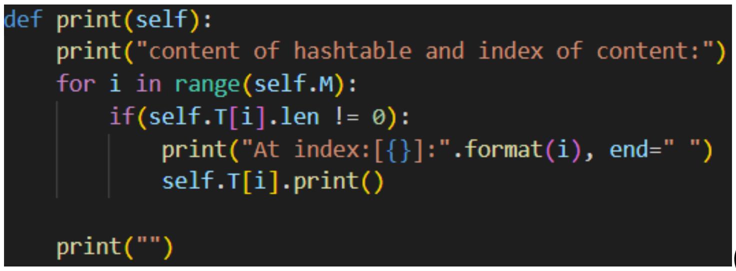
It searches the value by first hashing it so that we go to the correct linked list in the table. Strings are hashed with string folding in the foldin method (pic 1) and integers by getting the remainder of the division of the value and the tables size. Once we have identified the correct linked list in the table, we use the LinkedList classes index method which returns –1 if the value is not found in this case we return False from the search method.

- dist


```python
def dist(self):
    print("content of hashtable and index of content:")
    sum = 0
    biggest = 0
    smallest = float("inf")   # smallest is set to infinity initially

    for i in range(self.M):
        sum += self.T[i].len
        if(self.T[i].len < smallest):
            smallest = self.T[i].len
        if(self.T[i].len > biggest):
            biggest = self.T[i].len

    print("Average len: {}, biggest list: {}, smallest len: {}".format(sum/10000, biggest, smallest))
```
(picture 8)

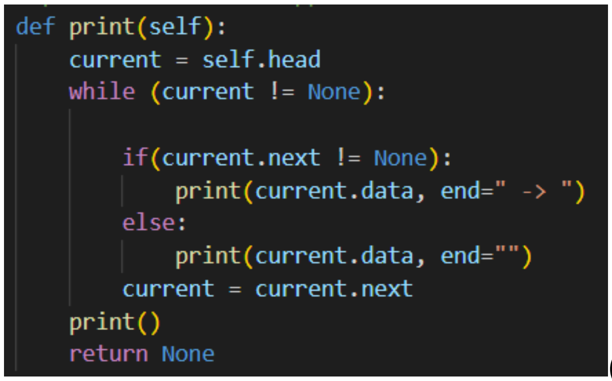
The dist method goes through the hash table, checks for the longest and shortest list in the table and prints the average list length of the lists in it.

-Print



```python
def print(self):
    print("content of hashtable and index of content:")
    for i in range(self.M):
        if(self.T[i].len != 0):
            print("At index:[{}]:".format(i), end=" ")
            self.T[i].print()

    print("")
```

(picture 9)



```python
def print(self):
    current = self.head
    while (current != None):

        if(current.next != None):
            print(current.data, end=" -> ")
        else:
            print(current.data, end="")
        current = current.next
    print()
    return None
```

(picture 10)

Goes trough the whole table and prints LinkedList objects contents if the list contains any values at that particular index in the table. The values inside the LinkedList object are printed with the LinkedList print method separated by ->. The index of the Current place in the hash table is also printed.

2. Testing and Analyzing the Hash Table

2.1 Running time analysis of the hash table

1. What is the running time of adding a new value in your hash table and why?

The foldin method which uses String folding to hash a string is Θ(1) as the string length is almost always smaller than all the values that will be inserted(n) to the table.

Adding to the linked list would be Θ(1) because we simply add the new value at the end of the linked list. It is Θ(n) in the worst case because when checking the Linked list for duplicates we might need to traverse the list to find it. This happens when the linked list is as long as the hash table but by increasing the table size the hashing should distribute the values well in the table so the table is usually Θ(1) since this way the list size is much smaller than the table size.

2. What is the running time of finding a new value in your hash table and why?

As explained above usually Θ(1).

3. What is the running time of removing a new value in your hash table and why?

Θ(1) also because we use the same methods as above to locate the value to be removed.

3. The Pressure Test

Table 1. Results of the pressure test for hash_3.1.

Step	**Time (s)**
Initializing the hash table	0.03125s
Adding the words	7.0625s
Finding the common words	2.09375s

3.1 Comparison of the data structures

Step	**Time (s) (Hash table)**	**Time (s) (Linear array)**
Adding the words	🟥 7.0625s	🟩 0.22899s
Finding the common words	🟩 2.09375s	🟥 720.22s

Which data structure was faster in adding the words from the file and why?

Adding words to the list if we don't check duplicates is faster because we simply append to the end of the list. In the hashtable adding a word is less fast because we need to first hash the value then create a new node which is to be appended to the linked list we also need to first check the linked list for duplicates which could mean traversing up to 59 words before getting to the right word in this instance because the longest list contained 60 words. However, if we check for duplicate words then the hash table is the way to go.

In which data structure was the search faster and why?

Searching is way faster in the hashtable because if the table is big enough and the distribution is good, we can make a search in constant time. In the list however we need to go through the whole list in the worst case and this means that we search in Θ(n) time which takes a lot more time than the hashtable.

3.2 Further improvements

Are you able to make the program faster?

1. Try to change the size of the hash table.

When changing the size to bigger for example table size 100000 then the initialization takes more time but the searching time is reduced and thus the insert and search methods speed up and the overall time to complete the tasks take less time.

Step	**Time (s)**
Initializing the hash table	0.5625s
Adding the words	4.1875s
Finding the common words	0.640625s

2. How well is the data distributed in the hash table?

Cases (hash table size 10000	size
Average list size	37,01
Biggest list	60
Smallest list	17

As the wordsalpha.txt file contains 370105 words and the average list size is 37, 01 the distribution is good in the hashtable.

List of references

- Course material
- Stackoverflow, 2009, https://stackoverflow.com/questions/1807026/initialize-a-list-of-objects-in-python.