CS532 Homework 6

Archana Machireddy

Question 1

import math

from timeit import default\_timer as timer

import random

class ht\_element:

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

self.key = key

self.next = None

self.prev = None

def next\_power\_of\_2(x):

return 1 if x == 0 else 2\*\*math.ceil(math.log2(x))

class chained\_hash:

def \_\_init\_\_(self,t\_size):

self.size = t\_size

self.new\_size = next\_power\_of\_2(self.size)

self.T = [None] \* self.new\_size

self.power = int(math.log2(self.new\_size))

knuth\_A = (math.sqrt(5)-1)/2

# self.p=14

self.p = self.power

m = 2\*\*self.p

self.w = 32

self.s = int(knuth\_A \* 2\*\*(self.w))

def insert(self, x):

hash\_val = self.hash\_function(x.key)

if self.T[hash\_val] is None:

self.T[hash\_val] = x

else:

node = self.T[hash\_val]

x.next = node

node.prev = x

self.T[hash\_val] = x

def search(self,k):

hash\_val = self.hash\_function(k)

ans = None

node = self.T[hash\_val]

while node is not None:

if node.key == k:

ans = node

break

else:

node = node.next

return ans

def delete(self,x):

hash\_val = self.hash\_function(x.key)

if x.prev == None:

self.T[hash\_val] = x.next

else:

x.prev.next = x.next

if x.next is not None:

x.next.prev = x.prev

def hash\_function(self, k):

return k % self.new\_size

def print\_node(self,k):

hash\_val = self.hash\_function(k)

x = self.T[hash\_val]

list\_contents=[]

while x is not None:

list\_contents.append(str(x.key))

x = x.next

return " -> ".join(list\_contents)

def hash\_function\_mul(self,k):

res= k \* self.s

# print('k: ',k)

# print('k\*s: ',res)

r1 = res/(2\*\*self.w)

r0 = res%(2\*\*self.w)

# print('r1: ',r1)

# print('r0: ',r0)

final = r0 >> (self.w-self.p)

return final

def test1():

h = chained\_hash(10)

print('Hash table Initialization:')

print(h.T)

h.insert(ht\_element(7))

print('Hash table after inserting element 7:')

print(h.T)

print('Printing linked list at slot that key 7 was hashed to:')

print(h.print\_node(7))

h.insert(ht\_element(8))

h.insert(ht\_element(24))

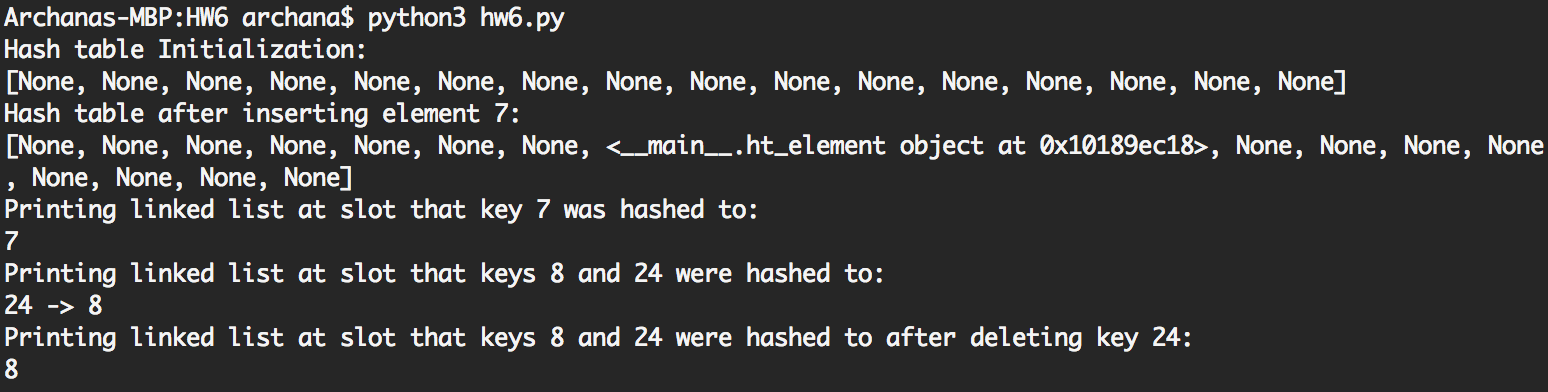
print('Printing linked list at slot that keys 8 and 24 were hashed to:')

print(h.print\_node(8))

print('Printing linked list at slot that keys 8 and 24 were hashed to after deleting key 24:')

h.delete(h.search(24))

print(h.print\_node(8))



Question 2

class ht\_element2(ht\_element):

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

super().\_\_init\_\_(key)

self.value = value

def test2():

h = chained\_hash(10)

h.insert(ht\_element2(7,5))

h.insert(ht\_element2(8,6))

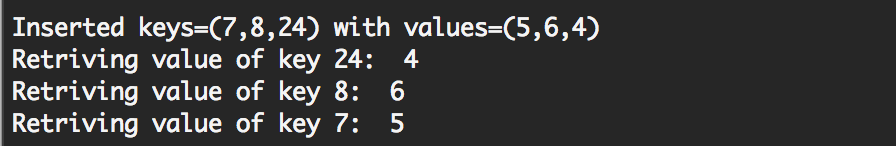
h.insert(ht\_element2(24,4))

print('Inserted keys=(7,8,24) with values=(5,6,4)')

print('Retriving value of key 24: ', h.search(24).value)

print('Retriving value of key 8: ',h.search(8).value)

print('Retriving value of key 7: ',h.search(7).value)



Question 3

def mul\_hash(k):

knuth\_A = (math.sqrt(5)-1)/2

p=14

m = 2\*\*p

w = 32

s = int(knuth\_A \* 2\*\*(w))

res= k \* s

print('k: ',k)

print('k\*s: ',res)

r1 = res/(2\*\*w)

r0 = res%(2\*\*w)

print('r1: ',r1)

print('r0: ',r0)

final = r0 >> (w-p)

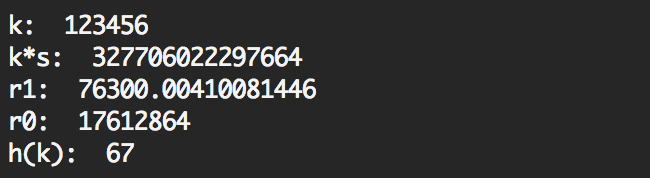
return final

def test\_mul\_hash():

k=123456

check = mul\_hash(k)

print('h(k): ', check)



Integrating it in the chained hash class is shown in solution to question 1.

Question 4

class Node(object):

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

self.parent = self

self.value = value

self.rank = 0

def \_\_str\_\_(self):

return self.value

class forests():

def \_\_init\_\_(self,values=[]):

self.set = [Node(value) for value in values]

def make\_set(self,x):

# x.parent = x

# x.rank = 0

self.set.append(Node(x))

def union(self,x,y):

self.link(self.find\_set(x),self.find\_set(y))

def link(self,x,y):

if x.rank > y.rank:

y.parent = x

else:

x.parent = y

if x.rank == y.rank:

y.rank = y.rank + 1

def find\_set(self,x):

if x != x.parent:

x.parent = self.find\_set(x.parent)

return x.parent

def test\_forests():

a= forests(['a','b','c','d','e'])

print(len(a.set))

sets = [str(a.find\_set(x)) for x in a.set]

print("set representatives:\t\t", sets)

print("number of disjoint sets:\t", len(set(sets)))

a.union(a.set[0],a.set[2])

sets = [str(a.find\_set(x)) for x in a.set]

print("set representatives:\t\t", sets)

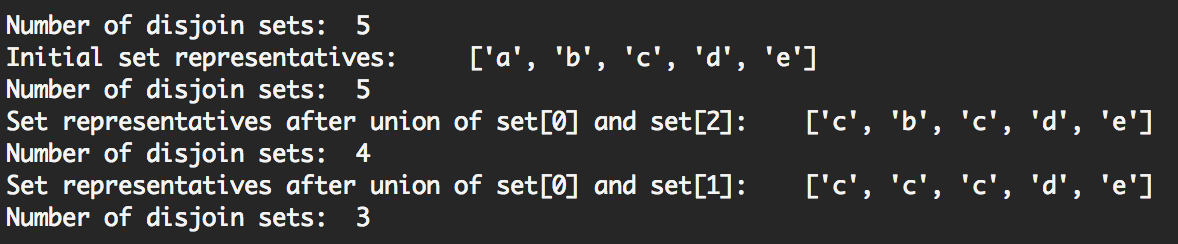
print("number of disjoint sets:\t", len(set(sets)))

a.union(a.set[0],a.set[1])

sets = [str(a.find\_set(x)) for x in a.set]

print("set representatives:\t\t", sets)

print("number of disjoint sets:\t", len(set(sets)))



The class initializer is doing the function of make-set during initialization (setting the parent, value and rank). So, it can just be included in the initialization.

Question 5

def time\_forests():

time = []

m = range(300,6000,300)

for i in m:

n = int(i/3)

start = timer()

f = forests()

for j in range(n):

f.make\_set(j)

left = i - n

while left > 0:

a = random.randint(0,n-1)

r1 = f.find\_set(f.set[a])

left = left-1

r2 = r1

while r1 == r2:

b = random.randint(0,n-1)

r2 = f.find\_set(f.set[b])

left = left-1

f.union(r1,r2)

left = left-1

end = timer()

print(i, end-start)

|  |  |
| --- | --- |
| m | Time |
|  |  |
| 3000 | 0.005694153 |
| 6000 | 0.01159682 |
| 9000 | 0.020357534 |
| 12000 | 0.032103187 |
| 15000 | 0.032596103 |
| 18000 | 0.035646319 |
| 21000 | 0.046231532 |
| 24000 | 0.049675917 |
| 27000 | 0.070627222 |
| 30000 | 0.06153777 |
| 33000 | 0.069074448 |
| 36000 | 0.073570288 |
| 39000 | 0.091271058 |
| 42000 | 0.083930505 |
| 45000 | 0.090872624 |
| 48000 | 0.100716254 |
| 51000 | 0.116978867 |
| 54000 | 0.107622541 |
| 57000 | 0.134907157 |
| 60000 | 0.13334815 |

The increase is almost linear.

Question 6

class forests\_2():

def \_\_init\_\_(self,values=[]):

self.set = [Node(value) for value in values]

def make\_set(self,x):

# x.parent = x

# x.rank = 0

self.set.append(Node(x))

def union(self,x,y):

x\_n = self.find\_set(x)

y\_n = self.find\_set(y)

x\_n.parent = y\_n

def find\_set(self,x):

if x == x.parent:

return x

else:

return self.find\_set(x.parent)

Question 7

def time\_forests(ver):

time = []

m = range(3000,63000,3000)

for i in m:

n = int(i/3)

start = timer()

if ver == 1:

f = forests()

else:

f = forests\_2()

for j in range(n):

f.make\_set(j)

left = i - n

while left > 0:

a = random.randint(0,n-1)

r1 = f.find\_set(f.set[a])

left = left-1

r2 = r1

while r1 == r2:

b = random.randint(0,n-1)

r2 = f.find\_set(f.set[b])

left = left-1

f.union(r1,r2)

left = left-1

end = timer()

print(i, end-start)

|  |  |
| --- | --- |
| m | Time |
|  |  |
| 3000 | 0.005218191 |
| 6000 | 0.017321978 |
| 9000 | 0.020208567 |
| 12000 | 0.032624985 |
| 15000 | 0.046544657 |
| 18000 | 0.048623987 |
| 21000 | 0.078163001 |
| 24000 | 0.082651261 |
| 27000 | 0.12400717 |
| 30000 | 0.11630151 |
| 33000 | 0.120294916 |
| 36000 | 0.149968778 |
| 39000 | 0.149292141 |
| 42000 | 0.169068866 |
| 45000 | 0.215476719 |
| 48000 | 0.259552196 |
| 51000 | 0.265630249 |
| 54000 | 0.254370726 |
| 57000 | 0.303942229 |
| 60000 | 0.306161941 |

The time taken for m = 60000 in the previous case is only 0.13 seconds, while here it is 0.30 seconds. Union by rank and path compression heuristics make sure the trees are not long. This makes the find operation faster. Without these heuristics, find has an average performance of O(log n) and worst case performance of O(n). So the time increases at a greater rate without the heuristics. The other two operations union and make set take constant time.