ΑΝΑΦΟΡΑ ΕΡΓΑΣΙΑΣ
για τις ΠΟΛΥΔΙΑΣΤΑΤΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΕΡΓΑΣΙΑ: Υλοποίηση του αλγορίθμου Chord (DHT)
ΟΜΑΔΑ: Αικατερίνη Δέρβου, 1054185, up1054185@upnet.gr, 4ο έτος.
Αριάδνη Μαχιά, 1059556, up1059556@upnet.gr, 4ο έτος.
Αθηνά Φουσέκη, 1059623, up1059623@upnet.gr, 4ο έτος

ΠΕΡΙΓΡΑΦΗ:

Ο αλγόριθμος Chord είναι ένα πρωτόκολλο για ένα peer-to-peer, κατανεμημένο hash table. Ένα κατανεμημένο hash table αποθηκεύει ζεύγη key-value, αποδίδοντας κλειδιά σε υπολογιστές, ή αλλιώς "κόμβους". Κάθε κόμβος αποθηκεύει τα value για τα keys για τα οποία είναι υπεύθυνος. Ο Chord υποδεικνύει πώς τα κλειδιά αποδίδονται σε κάθε κόμβο και πώς ο κάθε κόμβος μπορεί να ανακαλύψει το value που αντιστοιχεί σε κάποιο key, βρίσκοντας πρώτα τον κόμβο που είναι υπεύθυνος για αυτό το key.

Από Wikipedia: https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

ΥΛΟΠΟΙΗΣΗ:

Η ομάδα μας υλοποίησε τον αλγόριθμο Chord, ενεργοποιώντας εικονικά κόμβους σε ένα Chord Ring, μέσω των διευθύνσεων IP τους και στη συνέχεια περνούσαμε σε αυτούς values.

Μέσω ενός προγράμματος που γράψαμε στη γλώσσα Python, δημιουργήσαμε ένα αρχείο με ψευδοτυχαίες διευθύνσεις IP, οι οποίες αντιπροσωπεύουν κόμβους που θέλουν να γίνουν μέρος του Chord Ring. Οι IPs κατακερματίζονται μέσω της hash function (1) (SHA-1 -βιβλιοθήκη της Python) και μετατρέπονται σε IDs, μέσω των οποίων ενεργοποιούνται οι αντίστοιχοι κόμβοι. Στη συνέχεια, αφού έχει δημιουργηθεί το Chord Ring, φτιάχνουμε το finger table(2), το οποίο μας βοηθάει/ είναι υπεύθυνο για τη δρομολόγηση των διαφόρων αιτημάτων μέσα στον δακτύλιο. Έπειτα, σε κάθε κόμβο αποδίδονται κάποια values. Τα values(2) είναι τίτλοι ταινιών, τους οποίους κατακερματίζουμε, και ανάλογα με το hashed value, τοποθετούνται στον αντίστοιχο κόμβο μέσω της συνάρτησης lookup()(5), στον κόμβο όπου το id του είναι ίσο με το hashed value, ή, στην περίπτωση που αυτός δεν είναι ενεργός, στο κόμβο με το αμέσως μεγαλύτερο id.

Στο Chord Ring που έχουμε δημιουργήσει μπορούν να λάβουν χώρα οι ακόλουθες λειτουργίες:

1. Εισαγωγή νέου κόμβου
2. Διαγραφή Κόμβου
3. Αναζήτηση τίτλου ταινίας μέσω της lookup()

Επίσης, μπορούμε να εκτυπώσουμε τον δακτύλιο των ενεργών κόμβων, καθώς και τα finger tables αυτών.

Hash Function: Στο hash_function.py έχουμε μία συνάρτηση την hash που έχει ως ορίσματα το κλειδί (key) για τον κόμβο ή για την τιμή κάποιας ταινίας (value) και το

μέγεθος του ring του chord. Με το κλειδί(key) καλούμε την συνάρτηση encode() που το μετατρέπει από συμβολοσειρά σε byte ώστε να γίνει αποδεκτό από τη συνάρτηση κατακερματισμού(hashlib.sha1()). Αυτή η συνάρτηση, κατακερματίζει το κατάλληλα κωδικοποιημένο κλειδί και επιστρέφει αντικείμενο κατακερματισμού κλάσης SHA-1. (Διαλέξαμε hash SHA-1 γιατί είναι η πιο συνήθης και βολική). Επίσης, η συνάρτηση hexdigest() επιστρέφει την ισοδύναμη δεκαεξαδική τιμή του κατακερματισμένου αντικειμένου και έτσι θα χρησιμοποιούμε την δεκαεξαδική τιμή της.
Αφού βάζουμε ως μέγεθος(size) μόνο δυνάμεις του 2, εάν το υπόλοιπο της διαίρεσης του μεγέθους με το 16 δεν είναι μηδέν αυτό σημαίνει ότι έχουμε για μέγεθος δυνάμεις του 2 που είναι μικρότερες του 16 (δηλαδή 2, 4 ή 8). Για αυτές χρειαζόμαστε μόνο το τελευταίο δεκαεξαδικό ψηφίο που επιστρέφεται από την συνάρτηση κατακερματισμού. (Η SHA-1 είναι 160 bits οπότε συνολικά επιστρέφει 40 δεκαεξαδικά ψηφία). Έπειτα, μετατρέπουμε την δεκαεξαδική τιμή σε δεκαδικό ακέραιο (με την int(δεκάεξαδικός_αριθμός,16)) και από αυτόν, η hash συνάρτησή μας επιστρέφει το υπόλοιπο (modulo) της διαίρεσης του ακεραίου αυτού με το μέγεθος του ring. Επιστρέφει το modulo, καθώς θέλουμε να επιστρέφει έγκυρες τιμές, δηλαδή τιμές μικρότερες του ring.
Εάν το υπόλοιπο της διαίρεσης του μεγέθους του ring με το 16 είναι μηδέν αυτό σημαίνει ότι έχουμε για μέγεθος δυνάμεις του 2 που είναι μεγαλύτερο ή ίσο του 16. Για να αποφασίσουμε πόσα δεκαεξαδικά ψηφία χρειαζόμαστε από την συνάρτηση κατακερματισμού κάνουμε την διαίρεση του μεγέθους του ring με το 16. Το αποτέλεσμα(d) της θα δείξει πόσες φορές χωράει το 16 σε αυτόν τον αριθμό, οπότε και πόσο ψηφία χρειαζόμαστε. Έτσι, παρόμοια, αφού μετατρέψουμε την δεκαεξαδική τιμή σε δεκαδικό ακέραιο, η hash συνάρτησή μας επιστρέφει το υπόλοιπο (modulo) της διαίρεσης του ακεραίου αυτού με το μέγεθος του ring.

FingerTable: Το finger table είναι ένας πίνακας που έχει κάθε κόμβος. Περιέχει πληροφορίες χρήσιμες για τη δρομολόγηση αιτημάτων μέσα στον δακτύλιο. Το i-οστό στοιχείο του finger table είναι ο κόμβος που απέχει βήμα 2^i από τον κόμβο στον οποίο ανήκει, ή στην περίπτωση που δεν είναι ενεργός αυτός, ο αμέσως επόμενος ενεργός κόμβος.

Values: Οι τίτλοι ταινιών προέρχονται από ένα αρχείο csv με πάνω από 9.000 τίτλους, καθώς και με τα είδη στα οποία ανήκει η κάθε ταινία. Από το αρχείο, διαβάζουμε μόνο τους τίτλους των ταινιών, τους περνάμε από τη hash function

ΚΩΔΙΚΑΣ:

κύριο πρόγραμμα

```
import linecache
import os
import sys
from hash_function import hash
import math
import pandas as pd
from time import perf_counter, perf_counter_ns

# Disable printing
def blockPrint():
```

```
    sys.stdout = open(os.devnull, 'w')

# Restore printing
def enablePrint():
    sys.stdout = sys.__stdout__

def stabilize(sortli, ring_size):
    #stabilizing successors predecessors
    for item in sortli:
        if sortli.index(item) + 1 >= len(sortli):  # if it's the last
element
            item.successor = sortli[0]
            item.predecessor = sortli[sortli.index(item) - 1]
        elif item == sortli[0]:  # if it's the first element
            item.successor = sortli[sortli.index(item) + 1]
            item.predecessor = sortli[-1]
        else:
            item.successor = sortli[sortli.index(item) + 1]
            item.predecessor = sortli[sortli.index(item) - 1]

    # stabilizing the finger tables
    for item in sortli:
        fingTemp = []  # create a table for each node
        for i in range(0, int(math.log(ring_size, 2))):
            cand_node = item.id_ + math.pow(2, i)  # candidate node -
might not be active
            if cand_node > ring_size:
                cand_node = item.id_ + math.pow(2, i) - ring_size
            for item2 in sortli:  # iterating the list of the sorted
nodes to find the element to add to the finger table
                search_node = getattr(item2, 'id_')
                closest_value = cand_node - search_node

                if closest_value <= 0 and cand_node <= sortli[-1].id_:
                    fingTemp.append(item2)
                    break
                elif closest_value > 0 and cand_node <= sortli[-1].id_:
                    continue
                else:
                    fingTemp.append(sortli[0])
                    break

        setattr(item, 'finger', fingTemp)


def leave(list, dlt):  # deletes a node from the ring. It transfer the
nodeValues, fixes finger_table etc
    item = lookup(list, list[0], dlt)  # getting the node with id_ == dlt
| (item gets the node)
    # Passing the values of the deleting item to its successor before
removing it, so they won't be lost
```

```
    dlt_nodeValues = item.NodeValues  # nodeValues of deleting item
    suc_nodeValues = getattr(item.successor, 'NodeValues')
    suc_nodeValues.extend(dlt_nodeValues)  # adding the values of the
deleting item to its successor
    list.remove(item)  # Removing the item

    stabilize(list, ring_size)

    #printAllValues(list)  # PRINTS

def join(list, add):  # deals when a new node joins the ring. It adds the
node, fixes finger_table etc
    print("adding: ", add)
    addnode = Node(add)
    list.append(addnode)  # adding the node to the list

    list = sorted(list, key=lambda item: item.id_)  # sorting the list
    stabilize(list, ring_size)  # stabilizing the list
    # now, the new node has joined the ring with the correct
(succ/pred)essor & finger table

    suc = getattr(addnode, 'successor')  # suc = successor node of add
(the new node which is joining)
    valueTemp = []  # list with the temporary values which will be add to
the new node

    for item in reversed(suc.NodeValues):  # for loop is reversed as not
to affect the order of the NodeValues list
        IDvalue = hash(item, ring_size)
        if IDvalue <= addnode.id_ or IDvalue > list[list.__len__() -
1].id_:
            valueTemp.append(item)  # adding the value to the temporary
list for the new node
            suc.NodeValues.remove(item)  # REMOVING the value from the
successor (old node, not right any more)
    setattr(addnode, 'NodeValues', valueTemp)  # ADDING all the suitable
values to the new node

    # PRINTS
    #printAllValues(list)

    return list



def printAllValues(list):
    for item in list:
        print("For id: ", item.id_)
        for item2 in item.NodeValues:
            IDvalue = hash(item2, ring_size)
```

```
            print("id:", item.id_, "idvalue:{}".format(IDvalue), "
values: ", item2)



def valueassign(sortli):
    # reading values from csv file and convert to list
    df = pd.read_csv('movies.csv', usecols=["title"]).head(250)
    value_list = df["title"].astype(str).tolist()

    for item in value_list: #iterating list of values
        IDvalue = hash(item, ring_size) #hash value and get id of key
        valueTemp = []
        asnode = lookup(sortli, getattr(sortli[0], 'id_'), IDvalue) #node
that value must be held
        #append value to values of node
        if not asnode.NodeValues:
            valueTemp.append(item)
            setattr(asnode, 'NodeValues', valueTemp)
        else:
            valueTemp = asnode.NodeValues
            valueTemp.append(item)
            setattr(asnode, 'NodeValues', valueTemp)


def lookup(sortli, looks, to_find):  # ------------------
Search------------------------
    # if int(looks) <= int(to_find):

        for item in sortli:
            finder = item.id_
            if int(str(finder)) == int(str(looks)):
                print("\nWorking with node {}".format(item))
                succs = getattr(item, 'successor')
                pred = getattr(item, 'predecessor')

                if int(str(item)) == int(str(sortli[-1])) and
int(to_find) > int(str(item)):
                    return sortli[0]

                elif int(str(item)) == int(str(sortli[-1])) and
int(to_find) < int(str(sortli[0])):
                    return sortli[0]

                elif int(str(item)) == int(str(sortli[0])) and
int(to_find) > int(str(sortli[-1])):
                    return sortli[0]

                elif int(str(item)) == int(str(sortli[0])) and
int(to_find) < int(str(sortli[0])):
                    return sortli[0]
```

```
                elif int(str(item)) <= int(to_find):
                    #wanted node is the one we are looking at
                    if int(str(finder)) == int(str(to_find)):  # finder
== to_find
                        print("\tKey: {} exists in node with id:
{}".format(to_find, item.id_))
                        # print("finder == to_find:
{}".format(item.id_))--------------------FIX A
PRINT------------------------
                        return item

                    # wanted node is successor
                    elif int(str(succs)) == int(to_find):
                        print("\tKey: {} exists in node with id:
{}".format(to_find, item.successor))
                        return item.successor



                    #to_find among looks and successor
                    elif int(str(succs)) >= int(to_find) and int(finder)
< int(to_find):
                        print("\tKey: {} exists in node with id:
{}".format(to_find, item.successor))
                        # print("2exist suc: {}".format(i))
                        return item.successor

                    else:

                        print("\tIt's not in {}'s succesor. Iterating
finger table...".format(item))
                        for i in range(0,len(item.finger)):
                            node = item.finger[i]
                            #wanted node is the finger we are looking at
                            if int(str(node)) == int(to_find):
                                print("\tKey: {} exists in node with id:
{}".format(to_find, node))
                                return node
                            #wanted node among fingers
                            elif i < len(item.finger)-1 and
int(str(node)) < int(to_find) and int(str(item.finger[i+1])) >
int(to_find):  # and int(str(item)) < int(str(to_find))
                                print("\tKey: {} is among
fingers".format(to_find))
                                return lookup(sortli, node, to_find)

                            #wanted node among fingers but finger+1 goes
across ring
```

```
                            elif i+1 < len(item.finger) and
int(str(item.finger[i])) > int(str(item.finger[i+1])) and int(to_find)>
int(str(item.finger[i])):
                                print("\tKey: {} is among
fingers".format(to_find))
                                return lookup(sortli, node, to_find)
                            #wanted node is greater than fingers
                            elif i+1 == len(item.finger) and int(to_find)
> int(str(item.finger[i])):
                                print("\tKey: {} is greater than last
finger".format(to_find))
                                return lookup(sortli, item.finger[i],
to_find)


                elif int(str(item)) > int(str(to_find)):

                    #case where to_find < first node
                    if int(str(item)) == int(str(sortli[0])):
                        if int(to_find) < int(str(item)):
                            return sortli[0]

                    #wanted node is predecessor
                    elif int(str(pred)) == int(str(to_find)):
                        print("\tKey: {} exists in node with id:
{}".format(to_find, item.id_))
                        return item.predecessor

                    #wanted node among pred and looks
                    elif int(str(pred)) < int(to_find) and int(finder) >=
int(to_find):
                        print("\tKey: {} exists in node with id:
{}".format(to_find, item.id_))
                        return item

                    else:
                        #iterating finger table
                        for j in range(0, len(item.finger)):
                            fing = item.finger[j]
                            #found in finger table
                            if int(str(fing)) == int(str(to_find)):
                                return fing


                            # fing+1 goes around ring and fing+1 <
to_find
                            elif j+1 < len(item.finger) and
int(str(fing)) >= int(str(item.finger[j+1])) and
int(str(item.finger[j+1])) < int(to_find):
                                print("fing+1 goes around ring and fing+1
< to_find")
```

```
                                return lookup(sortli,
int(str(item.finger[j+1])) , to_find)

                            #fing+1 goes around ring and fing+1 > to_find
                            elif j+1 < len(item.finger) and
int(str(fing)) >= int(str(item.finger[j+1])) and
int(str(item.finger[j+1])) > int(to_find):
                                print("fing+1 goes around ring and fing+1
> to_find")
                                return lookup(sortli, fing, to_find)

                            #finger doesnt go around ring
                            elif j+1 < len(item.finger) and
int(str(item.finger[j])) == int(str(item.finger[-2])) and int(str(fing))
< int(str(item.finger[j+1])):
                                print("finger doesnt go around ring")
                                return lookup(sortli,
int(str(item.finger[j + 1])), to_find)




class Node:
    nodes = []

    def __init__(self, id_):
        self.id_ = int(id_)
        Node.nodes.append(self)  # passing each node in a list
        self.successor = 0
        self.predecessor = 0
        self.NodeValues = []

    def __str__(self):
        # return '('+str(self.id_)+')'
        return str(self.id_)


ring_size = int(input("Give the size of the ring: "))  # have a variable
to pass the max size of the Chord ring
print("\n\n")


numberofips = int(ring_size * 0.65)

for i in range(numberofips):
    data = linecache.getline('randip.txt', i).strip() #read ips from file
    hashedip = hash(data, ring_size) #get hashed value of ip
    check = True  # Checking if the inserting Node already exists
    for item in Node.nodes:
        if item.id_ == hashedip:
            check = False
```

```
    if check == True:  # if it is a new Node, the add it to the ring
        Node(hashedip)

sortli = sorted(Node.nodes, key=lambda item: item.id_) #sort nodes in
ascending order to make the ring

blockPrint()
build_and_assign_start = perf_counter()
stabilize(sortli, ring_size)  # calculating the successors, predecessors
and finger tables
valueassign(sortli) #assign values on
build_and_assign_stop = perf_counter()
enablePrint()

time_of_build = build_and_assign_stop - build_and_assign_start

for item in sortli:  # printing the ring
    print(item.predecessor, end='\t')
    print(item, end='\t')
    print(item.successor)

print("\n\n")

for item in sortli: #printing finger tables
    print("Node: {}".format(item))
    for n in item.finger:
        print("\t   {}".format(n))

print("\n\nTime of build was: {}".format(time_of_build))

# menu
while True:
    print(
        "\n\nWhat would you like to do next?\n1.Add Node\n2.Delete Node\
n3.Show Finger Tables\n4.Print Chord Ring\n5.Search \n6.Print Values of
Nodes \n7.Exit")
    sel = input("Choose one: ")
    # """----------------ADD NODE----------------------------------"""
    if sel == '1':
        tempnum = input("Give the id of the node: ")
        num = hash(tempnum, ring_size)  # HASH NUM - IDnode ###########
        blockPrint()
        join_start = perf_counter()
        while num >= ring_size:
            num = int(input("Give an id smaller than {}:
".format(ring_size)))
        check = True  # Checking if the inserting Node already exists
        for item in sortli:
            if item.id_ == num:
                check = False
        if check == True:  # if it is a new Node, the add it to the ring
```

```
            sortli = join(sortli, num)
        join_stop = perf_counter()
        enablePrint()
        print(check)
        print("node added: {}".format(num))
        time_of_join = join_stop - join_start
        print("Time of join was: {}".format(time_of_join))

    # """----------------CHORD RING-----------------------------------"""
    elif sel == '4':
        # print(sortli)
        for item in sortli:
            print(item.predecessor, end='\t')
            print(item, end='\t')
            print(item.successor)

    # """---------------FINGER
TABLES-----------------------------------"""
    elif sel == '3':
        print("\n\n")
        for item in sortli:
            print("Node: {}".format(item))
            for n in item.finger:
                print("\t   {}".format(n))

    # """----------------DELETE-----------------------------------"""
    elif sel == '2':
        dlt = input("Give the id of the node you'd like to delete: ")
        leave_start = perf_counter()
        leave(sortli, dlt)
        leave_stop = perf_counter()
        time_of_leave = leave_stop - leave_start
        print("Time of leave was: {}".format(time_of_leave))

    # -------------------SEARCH-----------------------------------
    elif sel == '5':
        looks = int(input("Give the node which will execute the lookup:
"))
        to_find = input("Give the key you'd like to find: ")
        h_to_find = int(hash(to_find, ring_size))
        print(h_to_find)
        #blockPrint()
        search_start = perf_counter()
        loo = lookup(sortli, looks, h_to_find)  # -----------------
passing the return node to loo -------------------
        search_stop = perf_counter()
        #enablePrint()
        time_of_search = search_stop - search_start
        print("Time of search was: {}".format(time_of_search))
        for itemm in loo.NodeValues:
            if itemm == to_find:
```

```
                print("Movie:{} in node: {}".format(itemm, str(loo.id_)))
    # --------------------AssignValues---------------------------
    elif sel == '6':
        print("Your current ring is:\n")
        for item in sortli:
            print("Values of node {} are:".format(item.id_))
            print(item.NodeValues)
            print("")
    # """----------------EXIT----------------------------------"""
    elif sel == '7':
        break

    else:
        print("Give valid input")
        continue
```

- -

hash function

```
import hashlib


# function witch returns INT value depending on the key and the size of the ring
def hash(key, size):  # size is a power of 2
    # encoding string by using encode() then sending to SHA1()
    result = hashlib.sha1(key.encode())

    m = size % 16  # module of 16
    d = size // 16  # division of 16

    if m == 0:
        return int(result.hexdigest()[-d:], 16) % size  # return the last d hex of the equivalent hexadecimal value.
    else:  # 2, 4, 8
        return int(result.hexdigest()[-1:], 16) % size  # return the last d hex of the equivalent hexadecimal value.
```

- -

ΧΡΟΝΟΙ:

- ➔ CPU: intel core i7-9750H @2.60GHz
- ➔ RAM: 8GB
- ➔ HDD

-Όλοι οι χρόνοι είναι σε seconds.

- Ανά ποσοστό ενεργοποιημένων κόμβων στο Chord Ring.

25% - build

Ring | #values	50	250	1000	5000	9125
8	0.014152400000000398	0.017342800000001546	0.026746399999999504	0.0800541999999993	0.1282017999999998
32	0.015419500000000141	0.027249499999999927	0.06170420000000032	0.25155079999999996	0.4607790999999999
512	0.036211200000000033	0.07926020000000022	0.25478080000000025	1.1690564	2.085152499999995
1024	0.108850900000000002	0.16953700000000005	0.47262240000000001	2.1044361	3.803420799999999996
32768	98.2299195	100.58174419999999	114.8778578	178.52474030000002	256.08264180000003

25% -search

Ring | #values	50	250	1000	5000	9125
8	0.00004017499	0.00003583999	0.00002791999	0.0000338	0.000037325
32	0.00014637499	0.0001131	0.00014528	0.00012848	0.000158175
512	0.00057641999	0.00064157999	0.00068464	0.00060852	0.00090221999
1024	0.0010237	0.000938225	0.00094498	0.001374625	0.00092455
32768	0.0135559	0.02418544	0.01983704	0.02915074	0.02158521999

25% - join

Ring | #values	50	250	1000	5000	9125
8	0.00010453333	0.00098777499	0.00477335	0.0689931	0.34505113333
32	0.00021499999	0.00038767999	0.00114514	0.00499218	0.0220317
512	0.020231875	0.01994	0.01599422	0.01628772	0.01658443333
1024	0.07686284	0.07767284	0.06406708333	0.06676448333	0.06645054
32768	97.748524525	94.47907702	102.3642322	96.8663096	92.99693066

25% - leave

Ring | #values	50	250	1000	5000	9125
8	0.00008349999	0.00009194999	0.00008034999	0.0001283	0.0001107
32	0.000201125	0.00021207499	0.00022562	0.00022339999	0.000233775
512	0.016131925	0.01375335	0.01825034	0.01439336	0.01596158
1024	0.0636326	0.07959916	0.05890486	0.06411672	0.06855762
32768	92.23046275	92.66398056	99.15170668	101.11040428	92.97358505

40% build

Ring | #values	50	250	1000	5000	9125
8	0.015577200000000069	0.018780200000000136	0.037217300000000001	0.134357200000000018	0.238899100000000025
32	0.015820199999999984	0.025839099999998894	0.066416300000000002	0.269990099999999934	0.478830499999999994
512	0.066124900000000015	0.121915400000000023	0.359881900000000004	1.589310499999999998	2.8805429
1024	0.196751100000000018	0.286815799999999984	0.736210600000000002	3.190062400000000004	6.095998
32768	217.7913859	231.4143883	232.2558832	328.9758722	434.1051875

40% search

Ring | #values	50	250	1000	5000	9125
8	0.00005691999	0.00006552499	0.00004265999	0.00005405	0.00005669999
32	0.00017658	0.00018882499	0.00015366	0.00013457499	0.00011084999
512	0.00099902499	0.00081482499	0.00096645	0.00069265	0.00051285
1024	0.00081287 5	0.00128392 5	0.0013407	0.00186396	0.00080185999
32768	0.02470926	0.03792072	0.03990608	0.03532448333	0.0375357

40% join

Ring | #values	50	250	1000	5000	9125
8	0.00011792	0.00032057499	0.001879425	0.015742625	0.04164925
32	0.00025333 75	0.00029907499	0.0009086	0.006148525	0.01131945
512	0.039761	0.03728898333	0.0355784666 6	0.044302475	0.03285885
1024	0.14267122	0.13230494	0.14267146	0.17069298	0.15647116
32768	207.843820225	223.3746813	202.137162325	200.85391928	198.973925317

40% leave

Ring | #values	50	250	1000	5000	9125
8	0.00010183333	0.00011363333	0.00010103333	0.0001229	0.00015326666
32	0.000218425	0.00022144999	0.00021105	0.000265075	0.000224225
512	0.0320384	0.03688775	0.0337707	0.032043825	0.02955172
1024	0.169018	0.14082158	0.12534424	0.12241158	0.11974218
32768	204.678178225	221.835005375	199.8666436	202.892232 58	201.13724918

65% build

Ring | #values	50	250	1000	5000	9125
8	0.016885499999999	0.020680899999999989	0.040969800000000056	0.15256820000000015	0.26446789999999964
32	0.016560500000000644	0.028040800000000031	0.071382299999999979	0.30910210000000005	0.5553626
512	0.092108099999999992	0.17577979999999993	0.4909873	2.2643938000000006	4.0845041
1024	0.2957174	0.48061799999999977	1.1557264000000003	4.5255476	7.9483994
32768	446.4741406999999 6	430.4642715999999 6	477.2577928	663.1166849	823.8875636

65% search

Ring | #values	50	250	1000	5000	9125
8	0.00009272	0.0000823999	0.0000795	0.00008475999	0.00008583999
32	0.00013638333	0.00015936	0.000138483333	0.00015783333	0.0001459
512	0.00068643999	0.0008883799 9	0.00085426	0.00120662	0.00115904
1024	0.0019904 5	0.001317725	0.00187706	0.00254292	0.0034963
32768	0.0469550 4	0.05036256	0.04354652	0.0496815	0.05655296

65% join

Ring | #values	50	250	1000	5000	9125
8	0.0001505	0.00025813333	0.0008986	0.00675653333	0.02022803333
32	0.0003838	0.00049226	0.00072259999	0.0030972 4	0.00657298
512	0.07473678	0.07018708	0.066217575	0.0680745	0.06673406
1024	0.2613835	0.34275008	0.27261744	0.29000778	0.29360481666
32768	429.35403345	426.5239403	432.173905225	471.6623396	461.0446691

65% leave

Ring | #values	50	250	1000	5000	9125
8	0.00013354	0.00017552	0.00012468	0.0001554	0.00014892
32	0.00035524	0.000363919999	0.00037774	0.00036578	0.0003975
512	0.07005893333	0.073945216666	0.06661635	0.069747233333	0.06478051666
1024	0.25869776666	0.29442014	0.29103453333	0.2905504	0.27114621666
32768	427.5408095	432.534048	431.8694684	468.397710733	474.002136

80% build

Ring | #values	50	250	1000	5000	9125
8	0.01513819999999999	0.020651899999999834	0.043536000000000002	0.161327800000000002	0.2685731999999996
32	0.019577200000000072	0.030699999999999995	0.073339800000000029	0.313521299999999967	0.565650900000000001
512	0.109648599999999987	0.216436599999999976	0.5612461	2.4992983000000004	4.4484832
1024	0.409249599999999999	0.5736055000000002	1.2980523999999996	5.3869472	8.8217383
32768	572.6839104999999	587.6632328000001	665.3605927	840.3832313	1042.878437

80% search

Ring | #values	50	250	1000	5000	9125
8	0.000122825	0.00010924999	0.00010976	0.00013549999	0.00009078
32	0.00016021999	0.00014468	0.00018635999	0.00023153999	0.00017398
512	0.00105632	0.0012482	0.00156574	0.00144412	0.00137552
1024	0.003541111999	0.00213328	0.0024006	0.0028595	0.00293556
32768	0.074250775	0.06250034	0.08327616	0.07407626	0.09361558

80% join

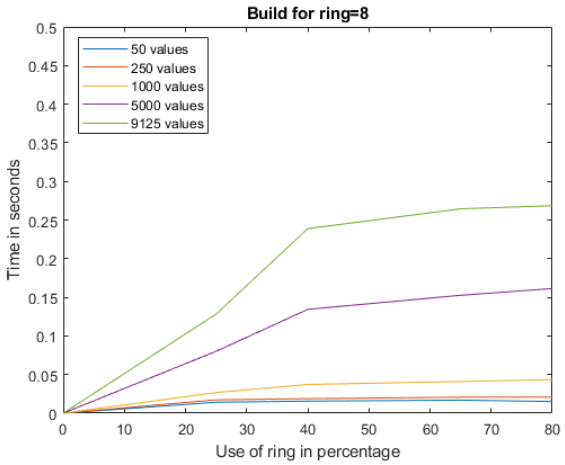
Ring | #values	50	250	1000	5000	9125
8	0.0001594499	0.0003032	0.00105665	0.0084779	0.0209136
32	0.00041724	0.0004739799	0.000729979999	0.00300481999	0.00574696
512	0.0871642	0.08604954	0.0939581	0.09516958333	0.08494946
1024	0.3603296633	0.3487356333	0.36951783333	0.35087235	0.36989412
32768	558.465510133	557.858041333	617.132023667	592.3268625	566.151720167

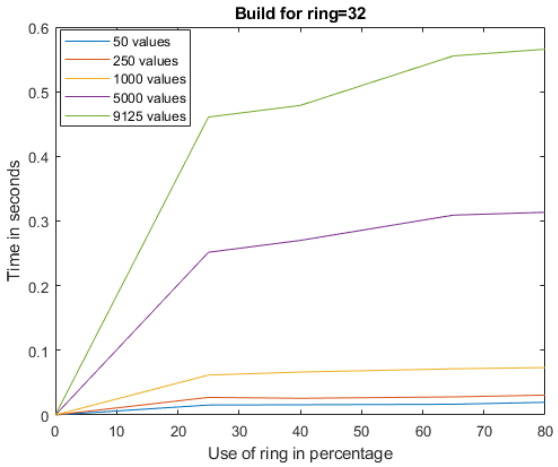
80% leave

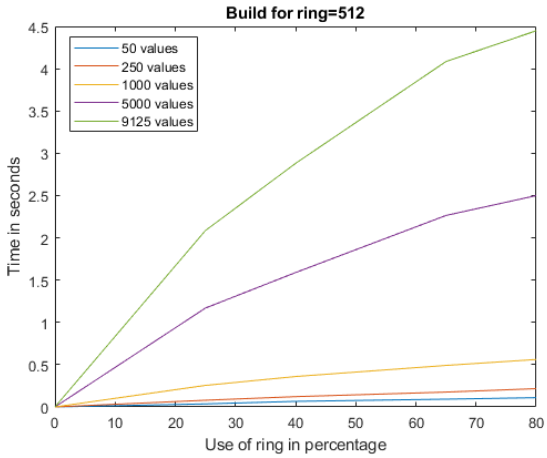
Ring | #values	50	250	1000	5000	9125
8	0.0001529	0.00016232	0.00014086	0.000171079999	0.00020294
32	0.00045058	0.00050278	0.00040538333	0.000412439999	0.00040666666
512	0.08360433333	0.08223501666	0.091989	0.08320015	0.08538468333
1024	0.3527412	0.35562596	0.3586482	0.35997078	0.36140268
32768	555.676566733	559.6123287	621.996813267	591.565491933	600.197295533

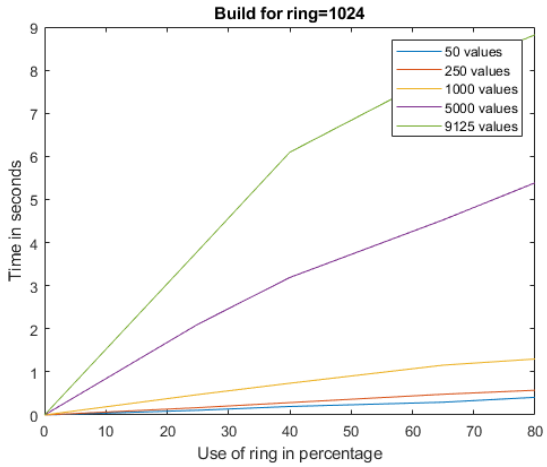
ΔΙΑΓΡΑΜΜΑΤΑ

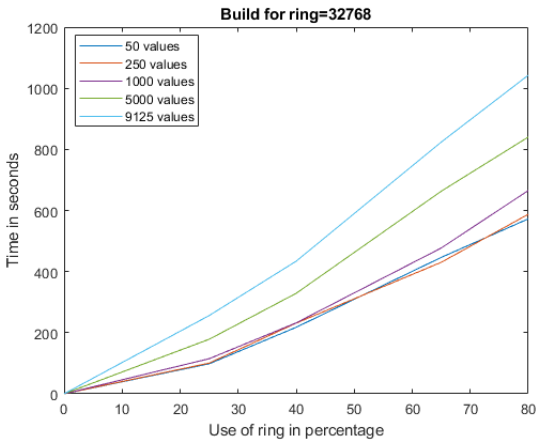
BUILD





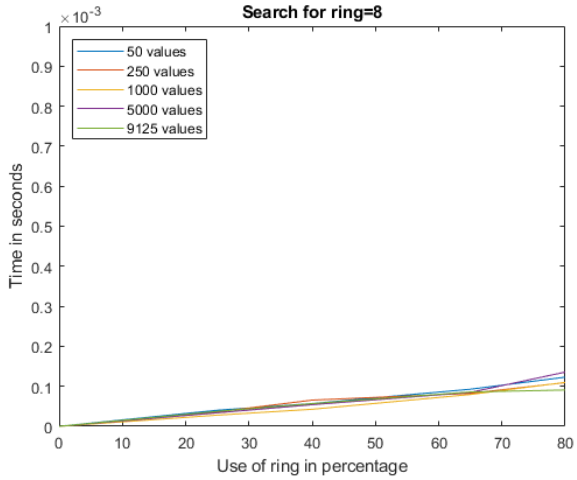


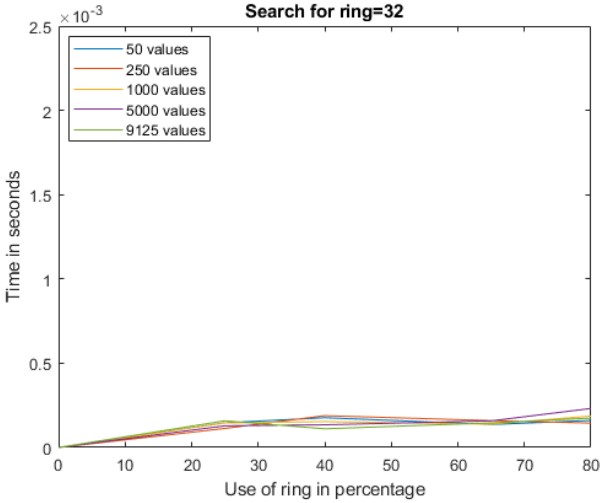


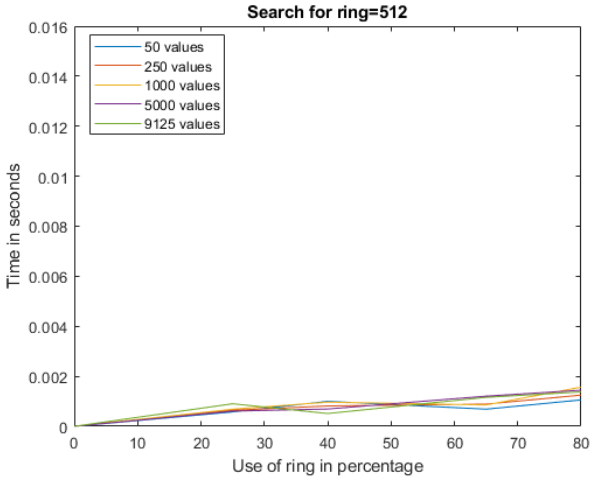


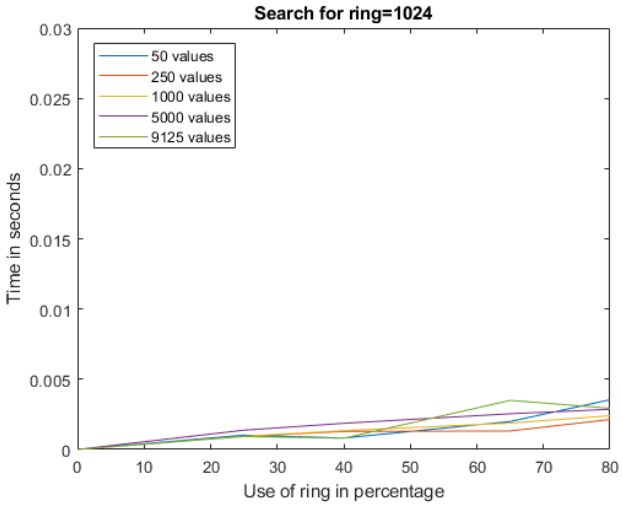
Η πολυπλοκότητα του **build** είναι το άθροισμα των πολυπλοκοτήτων των συναρτήσεων **stabilize** και **valueassign**. Η **stabilize** περιέχει ένα **for loop** που διατρέχει όλους τους ενεργούς κόμβους, οπότε έχει πολυπλοκότητα **O(n)** και ένα άλλο που διατρέχει πάλι τους ενεργούς κόμβους δημιουργώντας το **finger table** κάθε κόμβου με πολυπλοκότητα **O(nlogn)**. Όπου **n** το πλήθος των ενεργών κόμβων. Η **valueassign** περιέχει ένα **for loop** που διατρέχει όλα τα **values** όπου για κάθε **value** καλείται η συνάρτηση **lookup** για να εντοπιστεί ο σωστός κόμβος στον οποίον πρέπει να τοποθετηθεί. Η συνολική της πολυπλοκότητα είναι **O(mlogn)**, όπου **m** το πλήθος των **values**.
Επομένως, O_{build} = O(n) + O(nlogn) + O(mlogn) = O(nlogn).

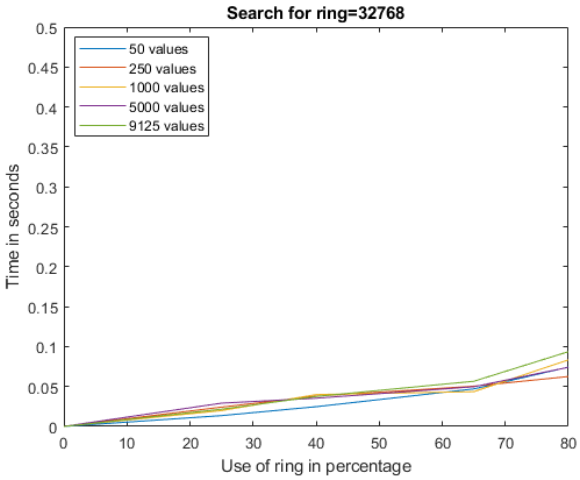
SEARCH





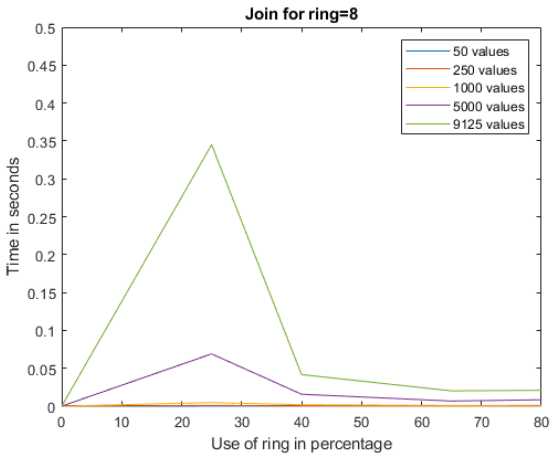


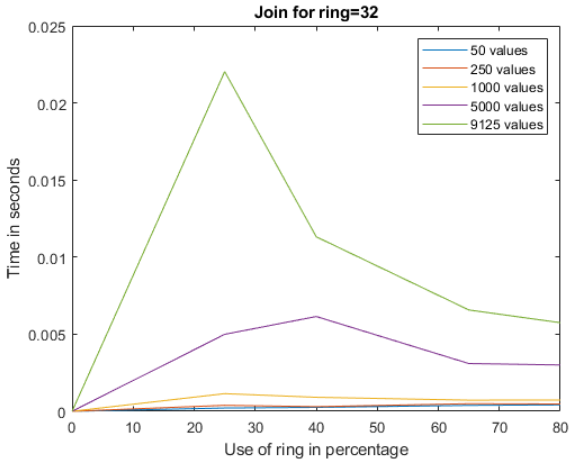


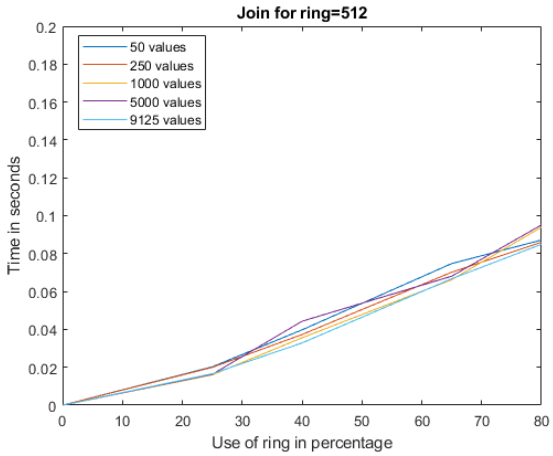


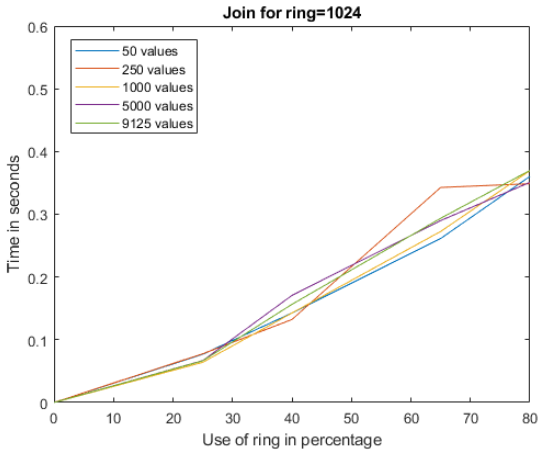
Στην χειρότερη περίπτωση, η συνάρτηση lookup θα εξετάσει έναν αριθμό κόμβων c, ο καθένας εκ των οποίων θα εξετάσει το πολύ logn άλλους κόμβους μέσω των finger table. Άρα η συνολική πολυπλοκότητα είναι $O_{search} = O(clogn) = O(logn)$, $c < n$ και c μία σταθερά.

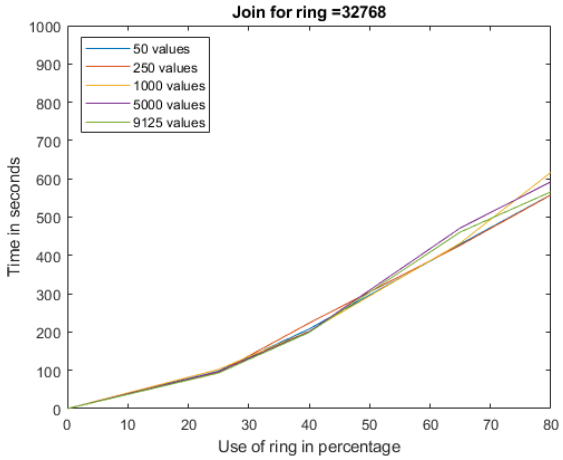
JOIN





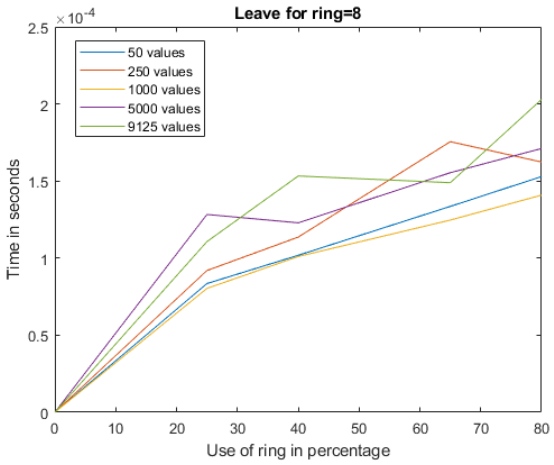


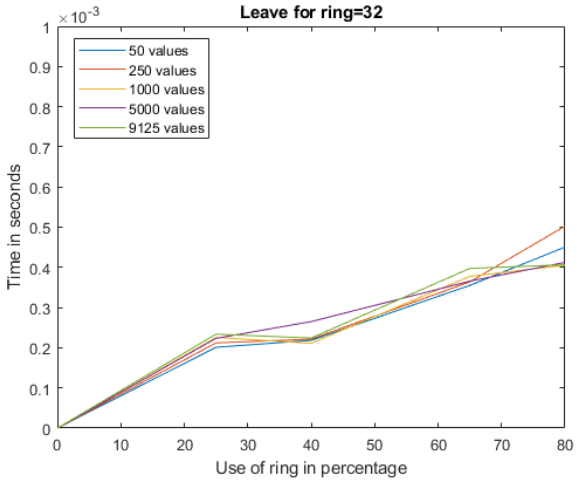


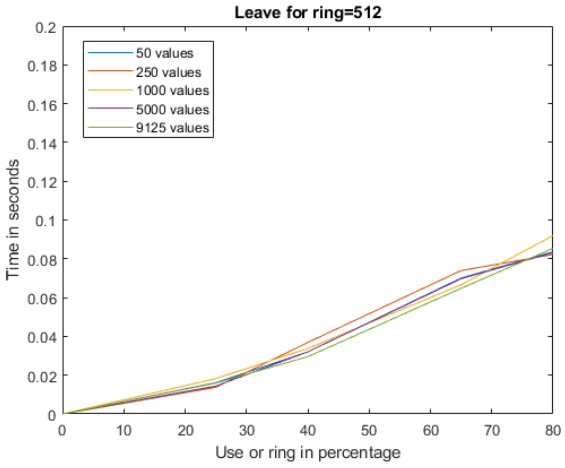


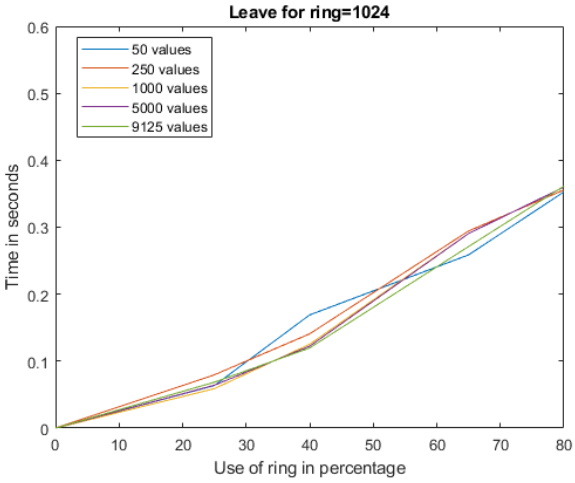
Η πολυπλοκότητα του join είναι το άθροισμα των πολυπλοκοτήτων των συναρτήσεων stabilize, sorted και ένα for loop m επαναλήψεων, όπου m το πλήθος των values. Η πολυπλοκότητα της stabilize όπως έχει προαναφερθεί είναι O(nlogn), η build-in συνάρτηση sorted έχει πολυπλοκότητα O(nlogn) και το for loop έχει O(m).
Επομένως, O_{join} = O(nlogn) + O(nlogn) + O(m) = O(nlogn).
Οι ανωμαλίες που παρατηρούμε στην συμπεριφορά του προγράμματος στα γραφήματα για τους δακτυλίους 8 και 32, για τα περισσότερα values (5000 και 9125), οφείλονται στο ότι μεγάλο πλήθος values διαμοιράζεται σε πολύ λίγους κόμβους. Αυτό έχει ως αποτέλεσμα με την εισαγωγή νέου κόμβου να είναι αναγκαία η μεταφορά πολλών values.

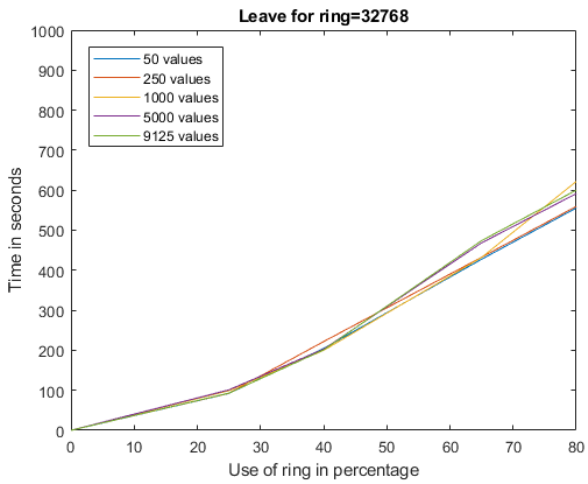
LEAVE







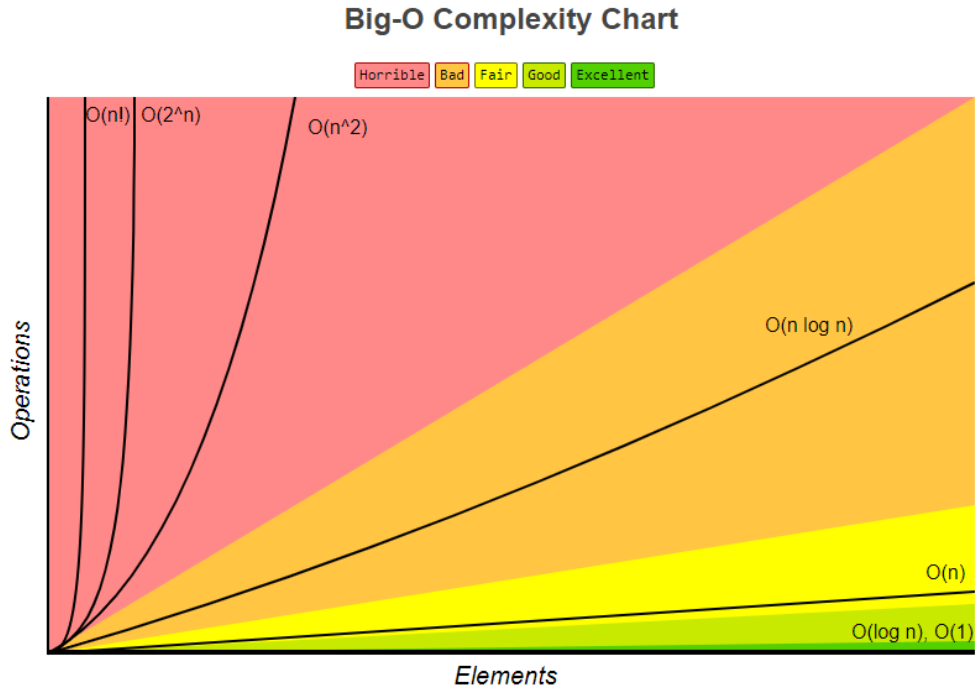




Η πολυπλοκότητα του leave είναι το άθροισμα των πολυπλοκοτήτων των συναρτήσεων lookup και stabilize. Η πολυπλοκότητα της lookup είναι O(logn) και της stabilize είναι O(nlogn). Συνολικά, O_{leave} = O(logn) + O(nlogn) = O(nlogn).

ΣΥΜΠΕΡΑΣΜΑΤΑ & ΒΕΛΤΙΩΣΕΙΣ

- Στον αλγόριθμο που φτιάξαμε παρατηρήσαμε ότι οι πράξεις join και leave είναι ιδιαίτερα κοστοβόρες και είναι προτιμότερη η αποφυγή μεταβολών στην δομή του δακτυλίου.



Από: https://www.bigocheatsheet.com/

- Η πράξη της αναζήτησης, από τα παραπάνω διαγράμματα και την πολυπλοκότητα της, φαίνεται ότι δεν επηρεάζεται σημαντικά από το πλήθος των ενεργών κόμβων και των values. Αποτέλεσμα αυτού είναι να μην υπάρχουν μεγάλες χρονικές διακυμάνσεις.
- Από τις μετρήσεις φαίνεται ότι το πλήθος των values δεν επηρεάζει τόσο την τελική πολυπλοκότητα όσο το πλήθος των ενεργών κόμβων.
- Μπορούμε να βελτιώσουμε την απόδοση του join, εάν αντί να καλούμε την συνάρτηση stabilize να αρχικοποιήσουμε το finger table του νέου κόμβου βάσει των finger tables των γειτονικών του κόμβων και να κάνουμε κάποια updates. Αυτό θα έχει ως αποτέλεσμα, η πολυπλοκότητα να είναι O(logn) αντί για O(nlogn).
- Εφόσον το finger table ενός κόμβου περιέχει ως πληροφορία ποιος θα είναι ο successor του θα μπορούσαμε να μην το έχουμε ως επιπλέον γνώρισμα του κόμβου.
- Μία άλλη έκδοση DHT είναι η Pastry η οποία μπορεί να κινηθεί αμφίδρομα.