מבני נתונים ואלגוריתמים

מגיש : מלאק אמיר 203764683

חלק א'

סעיף א'

בסעיף זה עברנו על כל המילים בקובץ והכנסנו אותם למערך בדרך הבאה :

עבור כל מילה עברנו על מערך המילים והשוונו את האות הראשונה של המילה הנוכחית עם האות הראשונה של המילים המערך. וזאת ע"מ למצוא את קבוצת המילים במערך שמתחילות באותה האות. לאחר מכן, מחפשים באותה קבוצת מילים האם המילה הופיעה בעבר. אם כן, מגדילים את ה counter של אותה מילה השמורה במערך ב 1. אחרת, ממשיכים עד שהגענו לאות ראשונה יותר גדולה מהאות הראשונה של המילה שלנו, ומוסיפים שם את המילה הרצויה.

סעיף ב'

עברנו על המילים במסמך, ועבור כל מילה בדקנו תחילה אם היא מופיעה ברשימת המילים שבקלט של הפונקציה. אם כן, ביצענו את מה שרשום בסעיף א'. אחרת, עברנו למילה הבאה במסמך.

סעיף ג'

עבור המסמך הראשון מבצעים את סעיף א'. מתקבלת רשימה של מילים, עבור רשימה זו ועבור המסמך הבא מבצעים את סעיף ב', מקבלים רשימת מילים חדשה, וממשיכים חלילה עבור כל המסמכים שנקלטו. בסוף תהליך זה מתקבלת רשימת מילים אשר מכילה את המילים המשותפות לכל המסמכים, עבור רשימה זו עוברים מסמך מסמך ומבצעים את סעיף ב' אך עם שינוי קטן – עבור כל מילה במסמך בודקים אם היא מופיעה ברשימה שלנו, אם כן לא מוסיפים אותה לרשימה החדשה. אחרת, מוסיפים.
לבסוף, עבור כל מסמך מתקבלת רשימת מילים שמכילה את המילים שאינם "לא מילים". כלומר, המילים בעלי הסתברות נמוכה להופעה במסמכים.
המילים ברשימה זו הם המילים שאיתם כדאי לתייג את המסמך.

סעיף ד'

עבור סעיף א' : עוברים על כל המילים במסמך (n מילים), עבור כ מילה עוברים על רשימת המילים המתעדכנת. במקרה הגרוע, שבו אין מילים החוזרות על עצמן במסמך יותר מפעם אחת, נצטרך לעבור עד לסוף הקבוצה של המילים המתאימה ברשימה. כאשר מספר המילים ברשימה הן כפונ' של מספר המילים במסמך שלנו. לכן נקבל לבסוף סיבוכיות של : $O(n^2)$.

עבור סעיף ב' : נניח כי רשימת המילים אותה מקבלים הינה באורך a מילים.
אנו עוברים על כל המילים במסמך ועבור כל מילה, במקרה הגרוע, עוברים על כל הרשימה שאותה קיבלנו. כלומר, $a \cdot n$ עד כה.
עבור המילים במסמך שמופיעות ברשימת הקלט (עד שמצאנו כי המילה אכן מופיעה עברנו, במקרה הגרוע, על כל רשימת הקלט. כלומר, a פעמים) שמספרן הינו קבוע כלשהו c (רוב הסיכוי $c \geq a$), נעבור על הרשימה המתעדכנת, במקרה הגרוע, b פעמים. כאשר b הינו מספר המילים שהופיעו עד כה במסמך וברשימה המתעדכנת (אצלנו $b \leq a$, כי בקוד שלנו הרשימה הסופית אינה מכילה מילים זהות).
לכן סיבוכיות המימוש שלנו : $a \cdot n + c \cdot a \cdot b = O(n)$.
הערה : עבור סעיף זה הנחנו ש $n \gg a$, שזו הנחה לגיטימית במנוע חיפוש.

עבור סעיף ג' : בהתחלה אנו מבצעים את סעיף א', לכן $f(n^2)$ עד כה. ונוצרת לנו רשימה שמספר המילים בה הוא כפונ' של n. כלומר $g(n)$.
לאחר מכן עבור כל מסמך (לא כולל את הראשון) מבצעים את סעיף ב', כלומר
$g(n) \cdot n + g'(n) \cdot g(n) \cdot g(n) = h(n^3)$.
כעת, חוזרים חלילה על כל המסמכים ומבצעים את סעיף ב' אך עבור המילים שלא מופיעות ברשימה (שמספרן הוא כפונ' של n, $w(n)$ למשל). ולכן בדומה לחישוב שתי שורות למעלה נקבל $w(n^3)$.
לסיכום,

$$f(n^2) + h(n^3) + w(n^3) = O(n^3)$$

הקודים של חלק א'

סעיף א' :

```
class wd:
    def __init__(self,string,counter):
        self.string=string
        self.counter=counter

def text_words_print(f):
    list_of_words=[]
    y=wd("pilot",1)
    list_of_words.insert(0,y)
    for word in f.read().split() :
        for i in range(0,len(list_of_words)) :
            if((i == len(list_of_words)-1) or (word[0] <= (list_of_words[i].string[0]))) :
                if (word == list_of_words[i].string) :
                    list_of_words[i].counter = list_of_words[i].counter + 1
                    break
                elif((i == len(list_of_words)-1) or (word[0] < (list_of_words[i].string)[0])) :
                    y=wd(word,1)
                    list_of_words.insert(i,y)
                    break
```

```
    for i in range(0,len(list_of_words)-1):
        print list_of_words[i].string ," : ", list_of_words[i].counter
    return

f=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file.txt","r")
text_words_print(f)
f.close()
```

סעיף ב' :

```
class wd:
    def __init__(self,string,counter):
        self.string=string
        self.counter=counter

def check_word_in_list(w,l):
    for i in range(0,len(l)):
        if(l[i]==w):
            return 1
    return 0

def text_words_print(f,l):
    list_of_words=[]
    y=wd("pilot",1)
    list_of_words.insert(0,y)
    for word in f.read().split() :
        if check_word_in_list(word,l):
            for i in range(0,len(list_of_words)) :
                if((i == len(list_of_words)-1) or (word[0] <= (list_of_words[i].string[0]))) :
                    if (word == list_of_words[i].string) :
                        list_of_words[i].counter = list_of_words[i].counter + 1
                        break
                    elif((i == len(list_of_words)-1) or (word[0] <
(list_of_words[i].string)[0])) :
                        y=wd(word,1)
                        list_of_words.insert(i,y)
                        break

    for i in range(0,len(list_of_words)-1):
        print list_of_words[i].string ," : ", list_of_words[i].counter
    return

f=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file.txt","r")
l=["moeen","the","amir","tall"]
text_words_print(f,l)
f.close()
```

סעיף ג' :

```
class wd:
    def __init__(self,string,counter):
        self.string=string
        self.counter=counter

#check if word "w" is in a list "l"
def check_word_in_list(w,l):
    for i in range(0,len(l)):
        if((l[i]).string==w):
            return 1
    return 0

#arrange the words of a given file in to a list
def get_list(f):
    list_of_words=[]
    y=wd("pilot",1)
    list_of_words.insert(0,y)
    for word in f.read().split() :
        for i in range(0,len(list_of_words)) :
            if((i == len(list_of_words)-1) or (word[0] <= (list_of_words[i].string[0]))) :
                if (word == list_of_words[i].string) :
                    list_of_words[i].counter = list_of_words[i].counter + 1
                    break
                elif((i == len(list_of_words)-1) or (word[0] < (list_of_words[i].string)[0])) :
                    y=wd(word,1)
                    list_of_words.insert(i,y)
                    break

    return list_of_words

#returns common words between file and list
def text_words(f,l):
    list_of_words=[]
    y=wd("pilot",1)
    list_of_words.insert(0,y)
    for word in f.read().split() :
        if check_word_in_list(word,l):
            for i in range(0,len(list_of_words)) :
                if((i == len(list_of_words)-1) or (word[0] <= (list_of_words[i].string[0]))) :
                    if (word == list_of_words[i].string) :
                        list_of_words[i].counter = list_of_words[i].counter + 1
                        break
```

```
                elif((i == len(list_of_words)-1) or (word[0] <
(list_of_words[i].string)[0])) :
                    y=wd(word,1)
                    list_of_words.insert(i,y)
                    break

    return list_of_words

#remove words given in a list from file and return the unremoved words
def remove_unwanted_words(f,l):
    list_of_words=[]
    y=wd("pilot",1)
    list_of_words.insert(0,y)
    for word in f.read().split() :
        if not check_word_in_list(word,l):
            for i in range(0,len(list_of_words)) :
                if((i == len(list_of_words)-1) or (word[0] <= (list_of_words[i].string[0]))) :
                    if (word == list_of_words[i].string) :
                        list_of_words[i].counter = list_of_words[i].counter + 1
                        break
                elif((i == len(list_of_words)-1) or (word[0] <
(list_of_words[i].string)[0])) :
                    y=wd(word,1)
                    list_of_words.insert(i,y)
                    break

    return  list_of_words

#print the list of words that the file could be maped with
def mapping_files(lf):
    l=get_list(lf[0])
    lf[0].seek(0)

    for i in range(1,len(lf)):
        l=text_words(lf[i],l)
        lf[i].seek(0)
    for i in range(0,len(lf)):
        s=remove_unwanted_words(lf[i],l)
        lf[i].seek(0)
        print "file",i+1," : "
        for i in range(0,len(s)-1):
            print s[i].string
    return

f1=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file1.txt","r")
f2=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file2.txt","r")
f3=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file3.txt","r")
f4=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file4.txt","r")
f5=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file5.txt","r")
f6=open("C:\\Users\\tanous\\Desktop\\pythonFile\\file6.txt","r")
```

```
f=[f1,f2,f3,f4,f5,f6]
mapping_files(f)

f1.close() ; f2.close() ; f3.close() ; f4.close() ; f5.close() ; f6.close()
```

חלק ב'

סעיף א'

את המיון בסעיף זה עשינו לפי "מיון ערימה". כל צומת בעץ שבנינו מייצגת מחלקה אשר כוללת שלושה שדות : שם האתר, תאריך רלוונטי, ושפת טקסט ייחודית לאתר.
המיון כולל שלושה חלקים עיקריים : HeapSort, HeapBuild, Heapify, אשר האלגוריתם שלהם מיוצג בקוד שלנו.
אומנם מיון זה אינו הכי מהיר מבחינת זמני ריצה : $O(n\log(n))$, לעומת "מיון מנייה" ו"בסיס", אך יש לו יתרונות אחרים לעומתם. מיון זה תומך ב"מיון במקום" וב "מיון השוואה", ובעל יתרון גדול : תור קדימויות (מבני נתונים דינמי).

סעיף ב'

כאן בחרנו באלגוריתם מיון "מיון בסיס". אין הרבה מה להסביר, השיטה נלמדה בהרצאות והקוד והתוצאות שבו מראות על "ביצועים מעולים".
ב "מיון בסיס" בשונה מ "מיון ערימה", זמן הריצה הינו $O(n)$. סיבה עיקרית להשוואה בינו לבין אלגוריתם של "מיון ערימה" (כך ניתן לראות בבירור את הפרש זמני הריצה). בנוסף, המיון הינו מיון יציב.
סיבה עיקרית נוספת לבחירת אלגוריתם מיון זה הינה שישנה מגבלה מסוימת למיון, והיא שהאלגוריתם יעבוד (בצורה טובה "יעילה") כאשר מספר דפי המיון גדולים ממספר הספרות שלפיהם אנו ממיינים. אך אצלנו, המספרים זהים (האיברים שאותם אנו ממיינים (התאריכים), ומספר הספרות של האיברים (תאריך מסוים)), דבר שמקשה על אלגוריתם זה, ובכך ניתן לראות ולהשוות את הביצועים שלו לעומת "מיון ערימה".
הדגש כי נעשו אותם שיקולים של סעיף א', רק שהמיון כאן מתבצע בהשוואת תאריכים ולא שפות טקסט, כאשר התאריכים מוכנסים כרצף של ספרות (כאילו שכל תאריך היה מספר אחד שלם).

סעיף ג'

השוואה בין שמני הריצה של שני האלגוריתמים התבצעה בעזרת הפונקציה :

```
from timeit import default_timer as timer
start = timer()
…
end = timer()
```

ובכך שמרנו את ערכי זמן ההתחלה (הזמן שבו הגענו לשורת ה timer הראשונה), וזמן הסיום (הזמן שבו הגענו לשורת ה timer השנייה).
חישוב הזמן שעבר :

```
Time Passed = end-start
```

* הערה : לא רשמנו כאן את התוצאות של זמני הריצה של האלגוריתמים, אך אם מריצים את הקוד רואים את התוצאות (זמני הריצה בנוסף כמובן למיונים) בבירור.
התוצאה שמבחינים בה הינה שהאלגוריתם של "מיון בסיס" ביצע את המיון בזמן קטן יותר מהאלגוריתם של "מיון ערימה". כלומר, קיבלנו ש "מיון בסיס" מהיר יותר מ "מיון ערימה".

סעיף ד'

שיטת מיון זו נלמדה בהרצאה כך שאין כל כך מה להסביר בנוגע למימוש שלה. בגדול השיטה בונה ערימה ובמעבר חיפוש עליה כגובה הערימה שלנו (מכאן $O(\log(n))$), ולכל צומת בעץ מבצעים עליה שוב Heapify אחרי שמעבירים אותה לשורש העץ (כדי למצוא שוב את הערך הגדול ביותר בערימה הנוכחית, אחרי הוצאת האיבר הראשון בגודלו). ולכן n מעברים שבכל אחד עוברים על הערימה שלנו כתלות בגובהה $O(\log(n))$, לכן הסיבוכיות של כל האלגוריתם הינה $O(nlog(n))$.

הקודים של חלק ב' (סעיף א' + סעיף ב' + סעיף ג')

```
#--------------------------------Language Sort : Heap--------------------------------

class site:
    def __init__(self,name,language,date):
        self.name=name
        self.language=language
        self.date=date


def heapify(A,i,lenght):

    l=2*i+1
    r=2*i+2
    if(l<=lenght and A[l].language>A[i].language):
        largest=l
    else:
        largest=i
    if(r<=lenght and A[r].language>A[largest].language):
        largest=r
    if(largest!=i):
        temp=A[i]
        A[i]=A[largest]
        A[largest]=temp
        heapify(A,largest,lenght)

    return


def build_heap(A,lenght):

    for i in range(0,(lenght/2)):
        heapify(A,(lenght/2)-1-i,lenght-1)

    return


def heap_sort(A):

    build_heap(A,len(A))
```

```
    lenght = len(A)
    for i in range(1,lenght):
        temp = A[0]
        A[0] = A[len(A)-i]
        A[len(A)-i] = temp
        lenght = lenght-1
        heapify(A,0,lenght-1)

    return




#--------------------------------Date Sort : Radix----------------------------

def radix_sort(A):
    dp=1
    digit=0

    while(digit<8):
        ll=[]
        ll.insert(0,[])
        ll.insert(1,[])
        ll.insert(2,[])
        ll.insert(3,[])
        ll.insert(4,[])
        ll.insert(5,[])
        ll.insert(6,[])
        ll.insert(7,[])
        ll.insert(8,[])
        ll.insert(9,[])

        for i in range(0,len(A)):
            temp_date = A[i].date
            temp_date = temp_date/dp
            temp_date = temp_date%10
            (ll[temp_date]).insert(len(ll[temp_date]),A[i])

        A=[]
        counter=0
        for i in range(0,10):
            if(len(ll[i]) != 0):
                for j in range(0,len(ll[i])):
```

```
            A.insert(counter,(ll[i])[j])
            counter = counter+1

        digit = digit+1
        dp = dp*10

    return A




date1 = 19920709 #9,7,1992
date2 = 19850924 #24,9,1985
date3 = 19920708 #8,7,1992
date4 = 19920609 #9,6,1992
date5 = 19910709 #9,7,1991
date6 = 19920809 #9,8,1992
date7 = 19920710 #10,7,1992
date8 = 18920709 #9,7,1892

s1 = site("name1","eng", date1)
s2 = site("name2","arb", date2)
s3 = site("name3","heb", date3)
s4 = site("name4","eng", date4)
s5 = site("name5","span", date5)
s6 = site("name6","japan", date6)
s7 = site("name7","french", date7)
s8 = site("name8","arb", date8)

A = [s1,s2,s3,s4,s5,s6,s7,s8]


#-----------------Calculating the time for both Heap Sort and Radix Sort--------------------

from timeit import default_timer as timer
start = timer()
A = radix_sort(A)
end = timer()

for i in range(0,8):
    print A[i].date

print "Start Time : ", start
```

```
print "End Time : ", end
print "Time Passed : ", end-start
```

```
date1 = [9,7,1992]
date2 = [24,9,1985]
s1 = site("name1","eng", date1)
s2 = site("name2","arb", date2)
s3 = site("name3","heb", date1)
s4 = site("name4","eng", date1)
s5 = site("name5","span", date1)
s6 = site("name6","japan", date1)
s7 = site("name7","french", date1)
s8 = site("name8","arb", date1)
A = [s1,s2,s3,s4,s5,s6,s7,s8]


start = timer()
heap_sort(A)
end = timer()

for i in range(0,8):
    print A[i].language

print "Start Time : ", start
print "End Time : ", end
print "Time Passed : ", end-start
```

חלק ג'

כאן ממומש עץ אדום שחור לאגירת דפי האינטרנט. כאשר כל "חרוז" (צומת) בעץ מייצג מחלקה שבה ישנה אובייקט של דף האינטרנט הרלוונטי, אובייקט של רשימת מילות החיפוש אשר נבחרו לשימוש קודם לכן (בחלק א' סעיף ג'), ואובייקטים שונים אשר עונים על תכונות העץ אדום שחור.

בכל פעם שישנה כפילות ברשימה הרלוונטית שלנו אנו רושמים את אותו דף אינטרנט פעם אחת בלבד ודואגים שמחלקה של אותה צומת האוגרת את הדף הרלוונטי בעץ, לעדכן את המפתח שלה כך בהתחשבות ברשימת המילים המופיעים בעמודי האינטרנט. במצב כזה, בשליפה או סידור של אובייקט במחלקה, דף אינטרנט לדוגמה, ניתן לבצע את הפעולות הנדרשות כולל טיפול בכפילויות וסידורם של שאר האובייקטים בעזרת הרשימה שייצרנו.

בקוד יש את כל הפונקציות העונות על תכונות עץ אדום שחור. מתאפשרת כמובן מחיקה של דף ממבני הנתונים, וגם כן טיפול בכל מקרי הקצה (מחיקת דף שלא קיים וכו').

סיבוכיות זמן הריצה תלוי באיזה דף אינטרנט אנו מעוניינים למצוא בעץ. ניקח את המקרה הקיצוני שבו נצטרך "לשלוף" אובייקט אשר מסודר בעץ בסוף שלו (בשורש), כלומר $O(\log(n))$. נצטרך לבצע חיפוש בכל פעם במפתח הצמתים עד אשר נגיע לדף האינטרנט הרלוונטי, בכל מחלקה כזו, נצטרך לרוץ על רשימת המילים שלנו ובמקביל גם כן על המילים המצויות בעמוד האינטרנט וסידורם כנדרש, ולכן ברישום בצורה כללית (כתלות בכמה פעמים נרוץ על כל עמוד ורשימה רלוונטית) :
$O(an + b) \cdot O(cn + d) = O(n^2)$. מכאן שניתן לסכם את שתי התוצאות ל $O(n^2 \cdot \log(n))$. וכל פעולה נוספת הדרושה מעץ אדום שחור מוסיפה לנו סה"כ זמן ריצה של $O(1)$. לכן, לסיכום $O(n^2 \cdot \log(n))$.

```
class Site:
    def __init__(self,name,keylist):
        self.name=name
        self.keylist=keylist

    def checkpercent(self,list1):
        counter = 0.0
        for i in range(0,len(list1)):
            for j in range(0,len(self.keylist)):
                if(list1[i] == self.keylist[j]):
                    counter = counter + 1
        percent = counter / len(self.keylist)

        return percent
```

```
class Node:
    def __init__(self, val, color = 'B'):
        self.val = val
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

    def grandparent(self):
        if self.parent != None:
          return self.parent.parent

    def summary(self):
        if self.val is None:
            return self.color + ":"

        return self.color + ":" + str(self.val)

    def delete(self, val):
        """
        Recurses if val != self.val

        Finds min if left exists, or max if
        right exists, and replaces current value
        with that

        Else, deletes current node, handles balance,
        and returns the deleted node.
        """

        if self.val is None:
            # deleting key that doesn't exist
            return None

        if val < self.val:
            self.left.delete(val)
        elif val > self.val:
            self.right.delete(val)
        else:
            if self.left.val is not None:
                deleted = self.left.deleteMax()
                self.val = deleted.val
                return deleted
            elif self.right.val is not None:
```

```
            deleted = self.right.deleteMin()
            self.val = deleted.val
            return deleted
        else:
            return self.deleteEnd()

    def find(self, val):
        if self.val is None:
            return None
        elif val < self.val:
            return self.left.find(val)
        elif val > self.val:
            return self.right.find(val)
        else:
            return self

    def deleteEnd(self):
        """
        Assumes this node has a value and two black leaf children.

        self.left takes this node's place after deletion.
        self.right is assumed to be discarded later.

        If this is a non-trivial delete, then we call rebalanceBegin
        on the replacement node before returning.
        """
        c = self.left if self.left.val is not None else self.right

        if self.parent is None:
            return self

        c.parent = self.parent
        if self.parent.left == self:
            self.parent.left = c
        else:
            self.parent.right = c

        if self.color == 'R':
            return self

        if c.color == 'R':
            c.color = 'B'
            return self
```

```
        c.rebalanceBegin()
        return self

    def deleteMax(self):
        if self.right.val is not None:
            return self.right.deleteMax()
        else:
            return self.deleteEnd()

    def deleteMin(self):
        if self.left.val is not None:
            return self.left.deleteMin()
        else:
            return self.deleteEnd()

    def uncle(self):
        g = self.grandparent()
        if g != None and g.left == self.parent:
            return g.right
        else:
            return g.left

    def insert(self, val):
        if self.val is None:
            self.val = val
            self.color = 'R'
            self.left = Node(None)
            self.left.parent = self
            self.right = Node(None)
            self.right.parent = self
            self.rebalanceForConsecutiveRedsBegin()
            return

        # No double inserts?
        # if val == self.val:
        #   return

        if val < self.val:
            self.left.insert(val)
        else:
            self.right.insert(val)

    def rebalanceForConsecutiveRedsBegin(self):
        """
```

```
        Assumes self is red. Parent may or
        may not be red.

        May recurse on grandparent.
    """
    if self.parent == None:
        self.color = 'B'
        return

    if self.parent.color == 'B':
        return

    g = self.grandparent()
    u = self.uncle()

    if g is None:
        self.parent.color = 'B'
        return

    if u != None and u.color == 'R':
        u.color = 'B'
        self.parent.color = 'B'
        g.color = 'R'
        g.rebalanceForConsecutiveRedsBegin()
        return

    if self.parent.left == self and \
       g.right == self.parent:
        self.parent.rotateRight()
        self.right.reblanceForConsecutiveRedsFinish()
        return
    elif self.parent.right == self and \
       g.left == self.parent:
        self.parent.rotateLeft()
        self.left.reblanceForConsecutiveRedsFinish()
        return

    self.reblanceForConsecutiveRedsFinish()

def reblanceForConsecutiveRedsFinish(self):
    """
        Assumes grandparent exists

        Assumes parent and self are red
```

```
        Assumes self is left of parent and parent is left grandparent
        or, self is right of parent and parent is right of grandparent

        Does not recurse.
    """
    g = self.grandparent()

    g.color = 'R'
    self.parent.color = 'B'
    if self.parent.left == self:
        g.rotateRight()
    else:
        g.rotateLeft()

def sibling(self):
    if self.parent == None: raise "Calling sibling on root node"
    if(self.parent.left == self): return self.parent.right;
    if(self.parent.right == self): return self.parent.left;

def rotateLeft(self):
    oldRight = self.right

    self.right = oldRight.left
    self.right.parent = self

    oldRight.left = self
    oldRight.parent = self.parent
    self.parent = oldRight

    if oldRight.parent is not None:
        if oldRight.parent.left == self:
            oldRight.parent.left = oldRight
        elif oldRight.parent.right == self:
            oldRight.parent.right = oldRight

def rotateRight(self):
    oldLeft = self.left

    self.left = oldLeft.right
    self.left.parent = self

    oldLeft.right = self
    oldLeft.parent = self.parent
```

```
    self.parent = oldLeft

    if oldLeft.parent is not None:
        if oldLeft.parent.left == self:
            oldLeft.parent.left = oldLeft
        elif oldLeft.parent.right == self:
            oldLeft.parent.right= oldLeft

def rebalanceBegin(self):
    """
      May tail-recurse upwards in the tree by calling
         self.parent.rebalanceBegin()

      May confine balance to a specific subtree by calling any of:
         self.left.rebalanceFinish()
         self.right.rebalanceFinish()
         self.rebalanceFinish()
    """
    if self.parent == None:
        return

    s = self.sibling()
    if s.color == 'R':
        s.color = 'B'
        self.parent.color = 'R'
        if self.parent.left == self:
            self.parent.rotateLeft()
            self.rebalanceFinish()
        else:
            self.parent.rotateRight()
            self.rebalanceFinish()
        return

    # All black? This side of the tree is really dense.
    # We need to contemplate a rotation toward
    # the other side. We recurse upward.
    if self.parent.color == 'B' and \
      s.color == 'B' and \
      s.left.color == 'B' and \
      s.right.color == 'B':
       s.color = 'R'
       self.parent.rebalanceBegin()
       return
```

```
    self.rebalanceFinish()

def rebalanceFinish(self):
    """
        Assumes that paths that go through self have one less black
        node than paths that go through sibling, and this needs to
        be fixed.

        Ensures that imbalances are fixed without examining any
        nodes higher than self.parent.

        Does not recurse.
    """
    s = self.sibling()
    if self.parent.color == 'R' and \
        s.color == 'B' and \
        s.left.color == 'B' and \
        s.right.color == 'B':
         self.parent.color = 'B'
         s.color = 'R'
         return

    if self.parent.left == self and \
        s.color == 'B' and \
        s.left.color == 'R' and \
        s.right.color == 'B':
         s.left.color = 'B'
         s.color = 'R'
         s.rotateRight()
    elif self.parent.right == self and \
        s.color == 'B' and \
        s.right.color == 'R' and \
        s.left.color == 'B':
         s.right.color = 'B'
         s.color = 'R'
         s.rotateLeft()

    s = self.sibling()
    if self.parent.left == self and \
        s.color == 'B' and \
        s.right.color == 'R':
         s.color = self.parent.color
         s.right.color = 'B'
         self.parent.color = 'B'
```

```
            self.parent.rotateLeft()
        elif self.parent.right == self and \
           s.color == 'B' and \
           s.left.color == 'R':
            s.color = self.parent.color
            s.left.color = 'B'
            self.parent.color = 'B'
            self.parent.rotateRight()

    def __str__(self):

        def recurse(node):
            if node is None: return [], 0, 0
            label = node.summary()
            left_lines, left_pos, left_width = recurse(node.left)
            right_lines, right_pos, right_width = recurse(node.right)
            middle = max(right_pos + left_width - left_pos + 1, len(label), 2)
            pos = left_pos + middle // 2
            width = left_pos + middle + right_width - right_pos
            while len(left_lines) < len(right_lines):
                left_lines.append(' ' * left_width)
            while len(right_lines) < len(left_lines):
                right_lines.append(' ' * right_width)
            if (middle - len(label)) % 2 == 1 and node.parent is not None and \
               node is node.parent.left and len(label) < middle:
                label += '.'
            label = label.center(middle, '.')
            if label[0] == '.': label = ' ' + label[1:]
            if label[-1] == '.': label = label[:-1] + ' '
            lines = [' ' * left_pos + label + ' ' * (right_width - right_pos),
                     ' ' * left_pos + '/' + ' ' * (middle-2) +
                     '\\' + ' ' * (right_width - right_pos)] + \
              [left_line + ' ' * (width - left_width - right_width) +
               right_line
               for left_line, right_line in zip(left_lines, right_lines)]
            return lines, pos, width
        return '\n'.join(recurse(self) [0])

class RedBlackTree:
    def __init__(self):
        self.root = None

    def randInit(self, n, max):
        """
```

```
        Returns list of elements that were inserted.
        """
        self.root = None
        for i in xrange(0, n):
            self.insert(random.randint(0,max))

    def delete(self, val):
        if self.root is None:
            return

        deleted = self.root.delete(val)
        if deleted == self.root:
            self.root = None
            return
        else:
            self.checkIfRootRotated()

    def find(self, val):
        if self.root is None:
            return None

        return self.root.find( val )

    def __str__(self):
        if self.root is None:
            return "<empty tree>"

        return str(self.root)

    def insert(self, val):
        if self.root == None:
            self.root = Node(val)
            self.root.left = Node(None)
            self.root.left.parent = self.root
            self.root.right = Node(None)
            self.root.right.parent = self.root
            return

        self.root.insert( val )
        self.checkIfRootRotated()

    def checkIfRootRotated(self):
        while self.root.parent is not None:
            self.root = self.root.parent
```

```
#########################################################
## Console interface
#########################################################

import sys, random, time, os

def performance_test():
    for n in xrange(1,15): # 1..14 inclusive
      nodes = 2**n

      if os.name == 'nt':
          timer = time.clock
      else:
          timer = time.time

      begin = timer()
      t = RedBlackTree()
      t.randInit( nodes, 30000 )
      runtime = timer() - begin
      print "n: %d, logb2: %d, runtime: %s (s)" % (nodes, n, runtime)

def main():
    """
    This whole thing is a hack.
    """
    t = RedBlackTree()

    if len( sys.argv ) == 1:
        print "Give one positive integer, or several integers. Inserts and random deletes
will occur for explicit inputs."
        return

    if len( sys.argv ) == 2:
        toInsert = t.randInit( int( sys.argv[1] ), 100 )
        print t
        return

    toInsert = []
    for i in sys.argv[1:]:
        toInsert.append( int(i) )

    for n in toInsert:
```

```
        t.insert( n )
        print
        print t

    random.shuffle(toInsert)
    for n in toInsert:
        t.delete(n)
        print
        print t

if __name__ == '__main__':
    #performance_test()
    main()
```

חלק ד'

בסעיף זה הגדרנו שתי מחלקות : מחלקה ראשונה עבור האתרים. ומחלקה שנייה עבור האיברים ברשימה של ה Hash Table.
המחלקה של האתר כללה שני שדות : שדה של רשימת מילים אשר נבחרו לשימוש, ושדה של שם האתר.
המחלקה של ה Hash Table כללה גם כן שני שדות : שדה של "מפתח" ושדה של "נתונים", כאשר השדה "מפתח" בכל תא בטבלה הינו הרשימה הראשונה שאליה משווים את שאר הרשימות שבשדה "רשימת המילים שנבחרו לשימוש" שבמחלקה הראשונה. והשדה "נתונים" הינו רשימה של מחלקות, שכל מחלקה בו הינה מסוג המחלקה הראשון (זאת שייצרנו עבור האתרים).
באלגוריתם שלנו אנו עוברים על רשימה שאבריה הינם משתנים מסוג המחלקה הראשונה. עבור כל איבר ברשימה זו (חוץ מהאיבר הראשון שאותו מכניסים ישר לטבלה בתא האפס), אנו משווים את המילים שנבחרו לשימוש ברשימה המקורים, עם המילים שנבחרו לשימוש ברשימה שבטבלה. אם אחוז המילים המשותפות הינו מעל 50% אזי מכניסים את משתנה זה (שהוא מסוג המחלקה המקורית-הראשונה) לתא שבטבלה שאליו השוונו. אחרת (פחות מ 50%) ממשיכים לתא הבא.
אם עברנו על כל התאים בטבלה ולא הייתה הכנסה לתא כלשהו, אזי מכניסים את המשתנה לתא האחרון (הלא מאוחסן) בטבלה.

הסיבוכיות של המימוש שלנו, מאחר ואנו עוברים בכל פעם (לשם השוואה) על מילות החיפוש שנבחרו בשתי הרשימות שאותם אנו משווים, ואנו עושים זאת לכל האתרים שברשותנו. כאשר בכל פעם ה"מפתח" שבתא שבטבלה מושווה עם הרשימה אותה אנו רוצים להכניס נכלל בבדיקה.
לכן אם כל מילות החיפוש שנבחרו הינם n אזי אנו בסה"כ עוברים $O(n) + O(n-1)$ על המילים שחיפוש שלנו ($O(n)$ – עבור ההשוואה של אותן רשימות שרוצים להכניס לטבלה. ו $O(n-1)$ - עבור הרשימה הנכללת בכל השוואה בתא המתאים בטבלה).
לכן בסה"כ מתקבלת סיבוכיות של $O(n)$.

הקוד של חלק ד'

```
class Site:
    def __init__(self,name,keylist):
        self.name=name
        self.keylist=keylist

class HashT:
    def __init__(self,key,sitelist):
        self.key=key
        self.sitelist=sitelist


def checkpercent(hashkey,site2):
    counter = 0.0
    for i in range(0,len(hashkey)):
        for j in range(0,len(site2.keylist)):
            if(hashkey[i] == site2.keylist[j]):
                counter = counter + 1
    percent = counter / len(site2.keylist)

    return percent


def hashtable(site,hashT,firstTime):
    flage = 0
    if(firstTime):
        lst = [site]
        class1 = HashT(site.keylist,lst)
        #class1.key = site.keylist
        #class1.sitelist = site
        hashT.insert(0,class1)
```

```
    else:
        for i in range(0,len(hashT)):
            if(checkpercent(hashT[i].key,site) >= 0.5):
                hashT[i].sitelist.insert(0,site)
                flage = 1
                break
        if(flage == 0):
            lst = [site]
            class1 = HashT(site.keylist,lst)
            hashT.insert(len(hashT),class1)

    return




hashT = []

s1 = Site("site1",["tree","red","black",7])
s2 = Site("site2",["tree","black","eagle",7])
s3 = Site("site3",["black","new","one",8])
s4 = Site("site4",["one","high","black",10])
s5 = Site("site5",["for","int","you",9])
s6 = Site("site6",["for",9,"int","black"])
s7 = Site("site7",["new",8,"one","black"])
s8 = Site("site8",[8,"new","black","one"])

l = [s1,s2,s3,s4,s5,s6,s7,s8]

hashtable(l[0],hashT,1)
"""print hashT[0].sitelist[0].name
hashT[0].sitelist.insert(0,s2)
print hashT[0].sitelist[0].name
print hashT[0].sitelist[1].name"""
for i in range(1,len(l)):
    hashtable(l[i],hashT,0)

for i in range(0,len(hashT)):
    print "hashT[",i,"] :"
    for j in range(0,len(hashT[i].sitelist)):
        print hashT[i].sitelist[j].name
```