תרגיל בית 4

מגישים: צוף בכר ועינם קסטל.

תאריך: 9.1.22

<u>שאלה 1:</u>

1.כיוון שאנו מקבלים עץ חיפוש בינארי T בגודל n, לכל צומת בT מתקיים הטענה הבאה:

א'- אם הוא אינו עלה בT, כל בן שמאלי של הצומת קטנה ממנה וכל בן ימני גדול ממנה.

ב' - בהרצאה ראינו כי האלגוריתם Inorder מדפיס גרף בינארי כלשהו בצורה הבאה: מדפיס את תת העץ השמאלי של שורש, מדפיס את השורש, ולאחר מכן מדפיס את תת העץ הימני.

לפי א' +ב' נקבל כי אם נשמור לתוך מערך בגודל n , את הפלט של Inorder(T), נקבל מערך ממוין של מפתחות T.

לכן האלגוריתם:

ניצור מערך בגודל n, (פעולה יחידה O(1))

נפעיל Inorder על T ונשמור את הפלט לתוך מערך בגודל n . (במקרה הגרוע ביותר 2n פעולות)

נסמן מיקום i=1. r=n,

כעת נדפיס את האיבר הזי במערך ולאחר מכן את האיבר במקום הראשון נקטין את r באחד ונגדיל את i באחד . נמשיך עד שנדפיס את כל המספרים (nפעולות)

סה"כ: $O(C*n) = O(n)$

.2

```
K_Nearest(A, x, k):
If k == n:
        Return A
B = new array of size(n)    (O(1))
B[i].d = | A[i] – x|             (O(n))
 B[i].value =   A[i]              (O(n))
C = Build_max_heap(B[1…k])      (O(k))
For i = k+1…. n:                       (O(nlogk)
        If C[1].d > B[i].d :    # biggest object in heap
                        C[1] = B[i]
                    Heapify(C,1)
```

לולאת forloop בזמן של $O(n*logk)$

```
For i in k:
                        C[i] = C[i].value
```

סהכ כנדרש : $O(n) + O(n) + 0(n) + O(nlogk) + O(k) = O(nlogk)$

שאלה 2:

כיוון שראינו בתרגול מספר 3 פתרון דומה למימוש. ניעזר בפתרון הנ"ל.

D יהיה מבנה נתונים אשר מבוסס על מבנה הנתונים 2-3 tree. נרחיב את תכונות כל צומת אמיתית במבנה החדש כך:

נסמן את כל הצמתים $v \in V$ בD.

לכל צומת $v \in V$ נוסיף את השדה: weight. שדה זה ישמור את סכום המפתחות שנמצאים בתת העץ של v (כולל את v).

לכל צומת $v \in V$ נוסיף את השדה: size: . שדה זה ישמור את מספר הצמתים בתת העץ השל v, כולל את v.

Init(D):

פונקציית האתחול תעשה דומה לזו של 2-3:

ראינו בהרצאה כי האתחול של 2-3 tree הינו O(1) ולכן גם עבור D הטענה מתקיימת.

Insert(D, x):

כיוון ש D מבוסס על עץ 2-3 ההוספה של צומת נוספת תיקח אותה סיבוכיות זמן$O(\log n)$

סהכ $O(\log n)$ (ראינו כי תחזוקה של שדות כאלו לא משנה את הסיבוכיות)

Delete(D, x):

באופן דומה לInsert(D, x), הפעולה תתרחש בסיבוכיות זמן של עץ 2-3

כלומר $O(\log n)$

AverageOf(D, k_1, k_2):

עבור k_1: נמצא את הצומת v_1 הקטנה ביותר אשר מקיימת כי $v.\mathrm{key}$ גדולה או שווה מהערך של k_1:

נעשה זאת באמצעות חיפוש בינארי של K_1 בעץ. פעולת חיפוש בינארי כידוע הינה בסיבוכיות של $O(\log n)$ כאשר n מספר האיברים בעץ.

אם לא מצאנו עלה מתאים בחיפוש בינארי, נשתמש בפונקציה 2-3_succssor

עבור k_2: נמצא נמצא את הצומת v_2 הגדולה ביותר אשר מקיימת כי $v.\mathrm{key}$ קטנה או שווה מהערך של k_1:

נשתמש שוב בחיפוש בינארי כדי למצוא את v_2 בסיבוכיות של $O(\log n)$

ראינו בתרגול 6 את הפונקציות הבאות:

$\mathrm{Sum\ Of\ Smaller\ Rec(D.root, k)}$ עם סיבוכיות של $O(\log n)$

Rank(x) עם סיבוכיות של $O(\log n)$

אם $v_2.key > v_1.Key$ אז אין איברים המקיימים את הדרישה ולכן נחזיר 0. (מתקיים באופן ריק)

אחרת נחזיר: $\frac{\text{Sum of smaller}(D,x2.key) - \text{Sum of smaller}(D,x1.key) + x1.key}{\text{Rank}(D,x2) - \text{Rank}(D,x1) + 1}$

סה"כ הפונקציה היא במספר סופי של $O(\log n)$ פעולות ולכן עומדת בתנאי סיבוכיות כנדרש.

3-

Our Algorithm will use the following process:

1-

We will initialize minimum Binary-Heap: A with H[2],H[3]

2-

Print H[1]

3-

We will run While loop that will stop running once K values have been printed. Stage 4-6 are part of this While loop.

4-

We will use Help_Extract_Min on A and will store the returned value in the variable: Temp

5-

Print Temp

6-

We will insert to A the son/sons of Temp. (Two/One/Zero depending on if temp is internal node or leaf)

Correctness:

Corollary: Let T be a group of the M smallest keys in H: (a1,a2,…,aM).

Thus, the Parent of A(M+1)(the element with the M+1 smallest key) belongs to T.

Assume to the contrary that this Corollary in not true. Thus, there exists aj that does not belong to T such that its son is a(M+1). By the property of Minimum-Heap: the key of aj is smaller than the key of a(M+1), in contradiction to the face that a(M+1) is the next element to be entered into T.

The sons of each element printed are inserted into H, thus the next element to be printed is always part of H.

Time Complexity:

Stage 1: The initialization of A is log(2)=O(1)

Stage 2: Printing H[1] is O(1)

Stage 3: The will loop will run for a maximum of K iterations. Thus O(k)

Stages 4-6:

Help_Extract_Min is will take log k time as shown in lecture.

Print Temp will take O(1)

Extracting the sons of Temp and Inserting them into A will take maximum O(logn*4)=O(log n) time.

Stages 4-6 are part of the While loop thus the total run time is O(k*logk)

4.-

4.1

Assume by induction that for each $k<n$

$T(K) <= C1*k$

$T(n)<=T(n/5)+T(3n/4)+C2*n<=C1*(n/5)+C1*(3n/4) + C2*n$

If $C1=(20/19)$ we will get:

$T(n)<=2*C2*n$

Thus $T(n)=O(n)$

4.2

Assume by induction that for each $k<n$

$T(k)<=C1*k$

$T(n)=2*T(n/4)+O(n^2)<= 2*(n/4)*C1 + C2*(n^2)$

If $C1=C2/2$ we will get:

$T(n)<=C2*(n/4)+C2*(n^2)<=2*C2*(n^2)$ for almost every n

Thus $T(n)=O(n^2)$

4.3

Assume by induction that for each $k<n$

$T(k)<=C1*k$

$T(n)=T(n-1)+O(2^n)<=C1*(n-1)+C2*(2^n)$

If $C1=2*C2$

$T(n)<=2*C2(n-1)+C2*(2^n)<=2*C2*2^n=C2*(2^n=1)$

The last inequation holds for all $n>N$ for some $N>0$.

Thus $T(n)=O(2^n)$

5.1

The median of X1,…..,Xn (by key value) is the element Xi of the group such that ⌈(n-1)/2⌉ elements of the group X1,…..,Xn are larger than in terms of key value, and ⌊(n-1)/2⌋ elements of the group are smaller than in terms of key value.(The assumption is that median refers to lower median, for upper-median the proof is analogous)

For each Xi it holds that it's weight is Wi=1/n.

We will define the Median of the group as Xk, it holds that:

$\sum_{xi<xk} Wi$=⌊(n-1)/2⌋(amount of elements smaller)*1/n(weight of each element)=

$$\begin{cases} \frac{n-1}{2} * \frac{1}{n} = \frac{1}{2} - \frac{1}{2n} < 1/2, \ n \ is \ odd \\ \frac{(n-2)}{2} * \frac{1}{n} = \frac{n-2}{2n} = \frac{1}{2} - \frac{1}{n} < 1/2, \ n \ is \ even \end{cases}$$

$\sum_{xk<xi} Wi$=⌊(n-1)/2⌋(amount of elements larger)*1/n(weight of each element)=

$$\begin{cases} \frac{n-1}{2} * \frac{1}{n} = \frac{1}{2} - \frac{1}{2n} < 1/2, \ n \ is \ odd \\ \frac{n}{2} * \frac{1}{n} = \frac{n}{2n} = \frac{1}{2} = 1/2, \ n \ is \ even \end{cases}$$

Thus by definition, the Median of X1,…..,Xn by key value is also the weighted median of the group

5.2

First we will use the merge-sort algorithm upon the n keys (O(nlogn)) to sort them. For each Xi in the sorted array there exists weight Wi.

We will run on the sorted array from the end to the beginning. In each iteration we will sum the weights of each element. This sum will be defined as W_sum.

We will start from Xn and will sum each element until we add the Xk element such that after addition of this element, W_sum>=1/2.

First Case: W_sum=1/2. In this case we will return Xk-1.

Explanation:

$\sum_{xi<xk-1} Wi < 1/2$ $\sum_{xk-1<xi} Wi$=1/2

The second sum is such because only when adding Xk we get that W_sum is ½ and the first sum is such because the sum of weights of elements k to n is ½ and the weight of Xk-1 is positive.

Second Case: W_sum>1/2 after adding Xk. In this case we will return Xk.

Explanation:

$$\sum_{xi<xk} Wi < 1/2 \ \sum_{xk<xi} Wi < 1/2$$

The first sum is such because the sum of weights of Xk to Xn is larger than ½ so the sum of X1 to Xk-1 must be smaller than ½. The second sum is such because only after adding Xk we get that the sum is equal/greater than ½. So the sum of Xk+1 to Xn is smaller than 1/2 .

Time complexity: In each iteration adding to sum and checking If W_sum is >=1/2 will take O(1). We will run this iteration a maximum of n times(n elements) thus the time complexity of this part will be O(n).

Merge-Sort will take O(nlogn) time so in total we will get:

O(nlogn)+O(n)=O(nlogn) for the whole algorithm

5.3

We will define the array as A with p=1 and r=n.

1-First we will use the Find_Median method on the keys of the array and will get the Median with respect to key values. Then we will use Partition function in order to Partition A[p . . .r] into subarrays A[p . . . q] and A[q + 1 . . .r] with respect to the pivot x(Median) so that A[i] < x < A[j] for every p ≤ i < q and q + 1 ≤ j ≤ r. We will set: A[q] = x.

2-Next, we will sum the weights of the array from index q+1(including) to r(including). If this sum is equal to ½ then we return A[q] and are done. (A[p…q-1] is smaller than 1/2, A[q + 1 . . .r] is equal to ½)

3-If the sum of A[q+1 . . .r] weights is larger than ½ we will repeat stage 1 with A[q+1. . .r].

If the sum of A[q+1 . . .r] is smaller than ½ we will check if the sum of weights of A[p…q-1] is smaller than 1/2. If so return A[q] and we are done. Else, repeat stage 1 with A[p,…,q-1].

We will repeat these stages iteratively until getting the Weighted_Medium.

As seen in Tutorial 9, the Find_Median method and Partition method time complexity is equal to the length of the array these function receive as input.

In the first iteration:

We call Partition and Find_Median on arrays of n length. Thus in the first iteration stage one is of time complexity O(n+n) = 0(n).

Stage 2+3: we will sum half of the array in each of these stages, time complexity: O(n/2 + n/2)=O(n).

Each iteration will be with an array half the length of the previous iteration, thus the Recurrence is:

$$T(n) <= T(n/2) + cn < cn\sum_{i=0}^{\infty}(1/2)^i = 2cn = O(n)$$