מבני נתונים: גרף

מגישים: יובל נוסוביצקי ועומר בלס
תז: 323986398 ,322450883

תוכן העניינים

חלק 1: הקוד

מחלקת Node:

מייצגת צומת בגרף.

שדות המלחקה הם:

תפקיד	שם
שומר את המספר המזהה של הצומת	int id
שומר את המשקל של הצומת	int weight
שומר את שכניו של הצומת (פרטים בהמשך)	Neighborhood neighborhood
שומר את האינדקס שהצומת נמצא בו בערימה (פרטים בהמשך)	int indexInHeap

Node(int id, int weight)

יוצר צומת חדש עם שדות int ו- weight, שאר השדות מאותחלים ל `0\null`. זמן הריצה ב- $O(1)$.

Neighborhood neighbors()

מחזיר את הייצוג של השכנים של הצומת, זמן הריצה $O(1)$

int getNeighborhoodWeight()

מחזיר את סכום המשקלים של הצומת ושל שכניו. מתבצעת רק קריאה ל `Neighborhood :: getHeighborhoodSum` וזה כפי שנראה בהמשך רץ ב- $O(1)$, ולכן זמן הריצה של הפונ' הוא $O(1)$.

Neighborhood.Edge addEdge(Node other)

מוסיף אובייקט של קשת מהצומת לצומת `other`, ומחזיר את האובייקט הזה. קורא ל- `Neighborhood :: insert` שהיא רצה ב- $O(1)$ כפי שנראה בהמשך.

int compareTo(Node o)

משווה בין הצומת הזו לצומת o לפי סכום שכניהם, רץ ב- $O(1)$.

מחלקת Heap:

מחלקה זו היא ערימת מקסימום אשר שומרת את הצמתים בגף, כאשר הסגר הואכפי שהוגדר בפונקציית ה `compareTo`, כלומר לפי סכום המשקלים שלהם ושל שכניהם.

שדות המחלקה הם:

שם	תפקיד
int size	שומר על כמות באיברים בערימה, כיוון שלא ניתן להוסיף צמתים המזפר יכול רק לקטון
Graph.Node[] heap	המערך שעליו נשמרת הערימה

Heap(Graph.Node[] heap)

פעולה זו יוצרת תחילה מעתיקה את תוכן המערך heap למערך פנימי this.heap, זה כמובן לוקח $O(n)$ זמן. ניתן לחשוב על האיברים במערך כאיברים בעץ, כפי שראינו בכיתה. לאחר מכן, היא תעבור על העץ מהצמתים מגובה 1 עד לשורש, ותקרא עבור כל אחד בפונקציה heapifyDown. נראה בקרוב כי פונקציה זו רצה בזמן $O(h)$, כאשר h זה הכובה של הצומת. לכן זמן הריצה של חלק זה של הקוד הוא:

$$
\begin{aligned}
1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \ldots + k \cdot \underbrace{\frac{n}{2^k}}_{\text{no. of nodes at hight } k} + \ldots + n \cdot 1 &= \sum_{i=1}^{\log_2 n} i \cdot \frac{n}{2^i} \\
&= n \sum_{i=1}^{\log_2 n} i \cdot \left(\frac{1}{2}\right)^i \\
&\leq n \sum_{i=1}^{\infty} i \left(\frac{1}{2}\right)^i \\
&= n \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} \\
&= 2n \\
&= O(n)
\end{aligned}
$$

ולכן זמן הריצה של הפונקציה הוא $O(n)$.

Graph.Node[] copy(Graph.Node[] heap)

מעתיק את האיברים של מערך למערך חדש שייצג את הערימה, ומעדכן את שדות ה indexInHeap בהתאם. זמן הריצה הוא $O(n)$.

void update(int i)

פעולה זאת נקראת לאחר שמעדכנים את סכום שכניו של הצומת הנמצא באינדקס i, וצריך לעדכן את מיקומו על מנת לשמר את תכונת הערימה. זמן הריצה של heapifyUp ושל heapifyDown הם שניהם $O(\log n)$, ולכן זה זמן הריצה של הפונ'.

void indexSwitch(int i, int j)

מחליף בין שני האיברים הנמצאים במקומות הi וה- j במערך, מעדכן את שדה ה `indexInHeap` שלהם בהתאם. זמן ריצה: $O(1)$.

void heapifyUp(int i)

מבצע את אלגוריתם ה heapifyUp כפי שראינו בכיתה. מכיוון שהוא עובר לכל היותר פעם אחת על כל גובה מעל הצומת, זמן הריצה של הפונקציה היא $O(d)$ כאשר d זה עומק הצומת בעץ, בפרט $O(\log n)$.

void heapifyDown(int i)

מבצע את אלגוריתם ה heapifyDown כפי שראינו בכיתה. מכיוון שהוא עובר לכל היותר פעם אחת על כל גובה מתחת הצומת, זמן הריצה של הפונקציה היא $O(h)$ כאשר h זה גובה הצומת בעץ, בפרט $O(\log n)$.

void delete(int i)

מבצע את אלוגריתם המחיקה מערימה כפי שראינו בכיתה, מעביק את האיבר במקום ה $\text{size} - 1$ לאינדקס i, ואז מבצע heapifyDown, לכן זמן הריצה יהיה $O(\log n)$

Graph.Node max()

מחזיר את הצומת עם סכום השכנים המקסימלי, זמן הריצה הוא $O(1)$ שכן צומת זה הוא פשוט הצומת באינדקס 0.

int leftChild(int i)

מחזיר את האינדקס של הבן הימני של הצומת במקום ה i, זמן הריצה הוא $O\left(1\right)$.

int rightChild(int i)

מחזיר את האינדקס של הבן הימני של הצומת במקום ה i, זמן הריצה הוא $O\left(1\right)$.

int parent(int i)

מחזיר את האינדקס של האב של הצומת במקום ה i, זמן הריצה הוא $O\left(1\right)$.

מחלקת HashTable:

מחלקה זאת מתחזקת טבלאת האש, אשר שומרת לכל מספר מזהה של צומת מצביע לצומת עצמו.

שדות המחלקה הם:

שם	תפקיד
int size	שומר על גודל הבטלא, פרופורציונלי לכמות הצמתים ההתחלתית.
HashTableNode[] table	מערך של רשימות מקושרות, כאשר כל חוליה בכל רשימה מקושרת מחזיקה צומת ומספר מזהה שלו.
int a, b, P	פרמטרים של פונקציית ההאש, P קבוע מראש, a, b נקבעים באקראיות בזמן הריצה.
static double scaleFactor	פקטור הפרופורציה בין כמות הצמתים ההתחלתית לגודל הטבלא

בנוסף יש את המחלקה הפנימית HashTableNode אשר מייצגת רשימה מקושרת כאשר בכל חולייה שמור צומת ומספר מזהה שלו.

HashTable(int size)

פעולה זו מאתחלת את טבלאת ההאש, ויוצרת את הפרמטרים לפונקציית ההאש זמן הריצה שלה הוא $O\left(n\right)$ שכן היא יוצרת מערך בגודל $n \cdot$ scaleFactor, ושאר הפעולות לוקחות $O\left(1\right)$.

int hash(int i)

פעולה זו מפעילה את פונקציית ההאש שנקבעה על מספר i. זמן הריצה הוא $O\left(1\right)$.

void insert(int id, Graph.Node node)

מכניס את הצומת והמספר המזהה לטבלאת ההאש. פעולה זו לוקחת $O\left(1\right)$, שכן למצוא את ההאש זה $O\left(1\right)$, ולהכניס איבר בתחילת רשימה מקושרת זה $O\left(1\right)$

Graph.Node get(int id)

הפעולה מחפשת את הצומת המתאים ל idמסויים. הפועלה מבצעת hash על id, ואז עוברת על הרשימה המקושרת שיש באינדקס הזה במערך, עד שהיא מוצאת את הid המתאים, אם בכלל. בתוחלת יש $O\left(\alpha\right) = O\left(\frac{1}{\text{scaleFactor}}\right) = O\left(1\right)$ איברים ברשימה המקושרת הזו, ולכן סיבוגיות הזמן של הפועלה התוחלת היא $O\left(1\right)$.

void delete(int id)

הפעולה מוחקת את הצומת המתאים ל- id מהטבלא. תחילה היא תבצע hash ל- id, ואז תעבור על הרשימה המקושרת שיש באינדקס זה בטלא ותמחק את הצומת הרלוונטי. יש $O\left(\alpha\right) = O\left(\frac{1}{\text{scaleFactor}}\right) = O\left(1\right)$ איברים ברשימה זו בתוחלת, ולכן סיבוכיו הזמן של הפכולה בתוחלת היא $O\left(1\right)$.

מחלקת Neighborhood:

מחלקה זו מייצגת את שכניו של צומת מסויים בעזרת רשימה מקושרת דו כיוונית של הצמתים.

במחלקה יש מחלקה פנימית בשם Edge אשר מייצגת קשת אחת ברשימה המקושרת, להלן שדות במחלקה:

שם	תפקיד
Graph.Node node	הצומת השני שאליו מחוברת הקשת
Edge next	הקשת הבאה הרשימה
Edge prev	הקשת הקודמת ברשימה
Edge otherEdge	מצביע לייצוג השני של הקשת

המלחה Edge ישנה פעולה אחת (חוץ מגטרים וסטרים) void delet(), שהיא פעולה המוחקת את הקשת עצמה מהרשימה הדו כיוונית שבא היא נמצא, ומעדכנת את neighborhhodSum בהתאם, והיא בסיבוגיות $O\left(1\right)$.

נתאר כעט את מחלקת Neighborhood.

שדות המחלקה הם:

שם	תפקיד
Edge first	מצביע לקשת הראשונה ברשימה
int neighborhoodSum	מחזיק את סכום המשקלים של הצמתים בשכונה

Edge insert(Graph.Node node)

הפעולה יוצאת קשת חדשה מהצומת של השכונה הזו, ל-node, הפעולה פשוט מכניה איבר לתחילה של רשימה מקושרת דו כיוונית, ומעדכנת פוינטרים ואת neighborhoodSum, ולכן זמן הריצה שלה הוא $O\left(1\right)$. הפועלה מחזירה את הקשת שיצרה.

מחלקת Graph:

מחלקה זו מייצגת את הגרף שרצינו. היא מתחזקת טבלאת האש שמעבירה מספרים מזהים ומצביעים למצמתים בגרף, וערימת מקסימום כפי שראינו קודם. שדות המחלקה הם כאלה:

שם	תפקיד
HashTable idToNodeTable	טבלאת האש ממספרים מזהים לצמתים
Heap heap	ערימת מקסימום המסודרת לפי סכום המקשלים של השכונות של הצמתים
int nodeCount	מספר הצמתים שכרגע בגרף
int edgeCount	מספר הקשתות שכרגע בגרף

Graph(Node[] nodes)

הפעולה הבונה של הגרף. מאתחלת את השדות שסופרים את הקשתות ואת הצמתים, יוצרת ערימה מהצמתים כפי שראינו ב-$O\left(n\right)$, ויוצרת טבלאת האש חדשה לצמתים, גם כן ב- $O\left(n\right)$, ואז עוברת על הצמתים ומוסיפה אותם לטבלה, כל הוספה זה $O\left(1\right)$ ולכן גם זה לוקח $O\left(n\right)$. לכן סיבוכיות הזמן של הפעולה היא $O\left(n\right)$.

Node maxNeghborhoodWeight()

מוצא את משקל השכונה הכי גבוה בעזרת ערימת המקסימום, ב- $O\left(1\right)$.

int getNeighborhoodWeight(int node_id)

תחילה מוצאת את הצומת עם המספר המזהה הזה, בעזרת טבלאת ההאש, ב$O\left(1\right)$ בתוחלת, ואז קוראת ל- Node::getNeighborhoodWeight שכפי שראינו לוקחת $O\left(1\right)$. לכן סך הכל, סיבוכיות הזמן של הפעולה הזו היא $O\left(1\right)$ בתוחלת.

boolean addEdge(int node1_id, int node2_id)

תחילה הפעולה תמצא את הצמתים עם המספרים המזהיים הללו, ב $O\left(1\right)$ בתוחלת. אם אחד מהם לא קיים, או שהם שווים, הפעולה תחזיר false. אחרת, הפעולה תוסיף את node1 לשכנים של node2, והפוך, ואז תדאג לכך שהקשתות שנוצרו מצביעות אחת על השניה, ואז יועדכנו המקומות שלהם בערימה, וזה לוקח $O\left(\log n\right)$ זמן כפי שראינו. לכן סיבוכיות הפעולה הכוללת היא $O\left(\log n\right)$ בתוחלת.

boolean deleteNode(int node_id)

תחילה הפעולה תמצא את הצומת עם המספר המזהה הזה בעזרת הטבלה ב$O\left(1\right)$ בתוחלת. אם הוא לא קיים תחזיר false. אם קיים, תעבור על כל הצמתים עם קשת שמצביע על הצומת, ותמחק מהם את הצומת הנ"ל, ואז תעדכן את המיקום בערימה. עדכון המיקום בערימה לוקח

$O(\log n)$, ויש d_v שכנים, לכן חלק זה יקח $O((d_v+1)\log n)$. לאחר מכן הפעולה תמחק את הצומת מטבלה, ב$O(1)$ בתוחלת, ואז תמחק את הצומת מהערימה ב $O(\log n)$. סה"כ זמן הריצה של הפעולה הוא $O((d_v+1)\log n)$ בתוחלת.

getNumNodes()

מחזיר את השדה, $O(1)$

getNumEdges()

מחזיר את השדה, $O(1)$

חלק 2: מדידות

הנה תוצאות במדידות:

i	n	דרגה מקסימלית	$3 \cdot \frac{\log_2 n}{\log_2 \log_2 n}$
6	64	6	6.96335053
7	128	6	7.480350929
8	256	8	8
9	512	8	8.517551673
10	1024	8	9.03089987
11	2048	8	9.539139268
12	4096	10	10.04194604
13	8192	9	10.53928802
14	16384	10	11.03128047
15	32768	11	11.51811112
16	65536	10	12
17	131072	11	12.47717765
18	262144	14	12.94987319
19	524288	11	13.41830806
20	1048576	12	13.88269279
21	2097152	13	14.34322567

הבעיה שלנו היא ווריאציה על הבעיה המוצגת בלינק. כאן ניתן לחשוב על כל צומת כעל סל, וכל קשת היא כדור. אבל במקום שכל כדור יכנס לסל אחד, כאן כל כדור נכנס לשני סלים, כלומר מגדיל את הדרגה של שני צמתים. בדף הויקיפדיה מוצגת נוסחה לתוחלת כמות הכדורים בכל סל, וניתן לראות כי בווריאציה הזו של הבעיה, אנחנו מקבלים בערך כפולה סקלרית של הנובחה המוצגת שם.