אתחול מערך מגודל n ב־$O(1)$

שלושה מערכים: A המערך עצמו, Legals אינדקסים חוקיים, Positions אינדקסים ל־Legals, ו־LegalsSize כמות האיברים ב־Legals. כל אינדקס שמופיע ב־Legals עד ל־LegalsSize הוא אינדקס מאותחל. ולכל אינדקס מאותחל נשמור ב־Positions את המיקום של האינדקס ב־Legals. בדיקת אתחול של האיבר ה־i: אם האינדקס $\text{Positions}[i]$ בתוך התחום של LegalsSize, וגם האיבר שמופיע ב־Legals במקום הזה הוא האינדקס.

עץ חיפוש בינארי

תכונה: לכל צומת, המפתח של כל איבר בתת העץ השמאלי קטן, והמפתח של כל איבר בתת העץ הימני גדול מהמפתח של הצומת.

פעולת Successor: אם יש בן ימני, נלך ימינה ואז כל הדרך שמאלה. אם אין בן ימני, נלך למעלה עד לפניה הראשונה ימינה (כלומר ־ שהגענו למעלה דרך הבן השמאלי).
בנוסף: אם עוברים על כל העץ אז עוברים על כל קשת לכל היותר פעמיים לכן העלות היא $O(n)$. גם אם מתחילים מאיבר x ומתקדמים k פעמים אז $O(k+h)$.

```
if x.right ≠ null then:
    return Min (x.right)
y ← x.parent
while y ≠ null and x = y.right
do:
    x ← y
    y ← x.parent
return y
```

פעולת Delete: אם אין ילדים ־ קל. אם יש רק בן אחד ־ נחליף בהורה את האיבר בבן שלו. אם יש שני ילדים: נחליף את x ב־successor של x, שנסמנו ב־y. ל־y אין בן שמאלי (כי הולכים ימינה ואז כל הדרך שמאלה). לכן נוציא את y מהעץ ונכניס אותו חזרה במקום x.

מעבר על העץ

Pre-Order	In-Order	Post-Order
if $x \neq$ null then	if $x \neq$ null then	if $x \neq$ null then
Process (x)	In-Order $(x.\text{left})$	Post-Order $(x.\text{left})$
Pre-Order $(x.\text{left})$	Process (x)	Post-Order $(x.\text{right})$
Pre-Order $(x.\text{right})$	In-Order $(x.\text{right})$	Process (x)

עץ AVL

תכונה: נגדיר $\text{BF}(v) = h(v.\text{left}) - h(v.\text{right})$, אז $|\text{BF}(v)| \leq 1$ לכל צומת v. נובע שהגובה הוא $O(\log n)$, הוכחה עם עץ פיבונאצ'י (הבנים של F_h הם F_{h-1} ו־F_{h-2}, מספר הצמתים הוא $f_{h+3}-1$ כאשר f סדרת פיבונאצ'י הרגילה $f_n = \frac{\Phi^n-(1-\Phi(\approx -0.618))^n}{\sqrt{5}}, \Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$).

פעולת רוטציה: מתקנת אם $|\text{BF}(v)| = 2$ אבל מה שמתחת בסדר. אם $\text{BF}(v) = 2$ נפצל למקרים לפי ה־BF של הבן השמאלי ($-1 \to LR, 0/1 \to R$), ואם $\text{BF}(v) = -2$ נפצל למקרים לפי ה־BF של הבן הימני ($-1/0 \to R, 1 \to RL$). (הציור של סיבוב ימני נמצא בסוף הדפנוס).

פעולת הכנסה: נכניס כרגיל, ואז נעלה למעלה בעץ ונחפש AVL criminals (עם $|\text{BF}(v)| = 2$). אם ה־BF חוקי והגובה לא השתנה בפעולת ההכנסה אז אפשר לעצור, אחרת נמשיך לעלות למעלה ואם $|\text{BF}(v)| = 2$ נבצע רוטציה. אחרי הכנסה יש לכל היותר רוטציה אחת לכן אפשר לעצור גם במקרה הזה.

פעולת מחיקה: יכול להיות שיהיו כמה רוטציות בדרך למעלה, ונתחיל מההורה של מי שמחקנו באמת (למשל זה יכול להיות ה־successor).

פעולת Join ששומרת על BF: נניח ש־$T_1 < x < T_2$ (כל איבר ב־T_1, T_2). סתם Join זה קל: ניצור שורש x, את T_1 נשים משמאל ואת T_2 נשים מימין לשורש. כדי לשמור על BF: בה"כ $h(T_1) \leq h(T_2)$. נמצא את b הצומת הראשון בשושלת השמאלית (ללכת כל הזמן שמאלה) של T_2 כך ש־$h(b) \leq h(T_1)$ (ולכן $h(b) \in \{h(T_1)-1, h(T_1)\}$). נסמן ב־$c$ את ההורה של b, נצרף מתחת ל־c את x כך שהבן השמאלי שלו הוא T_1 והבן הימני שלו הוא תת העץ של b. לאחר מכן נבצע תיקונים בדרך למעלה החל מ־x. סך הכל $O(\log n)$.

פיצול עץ באמצעות Join: בהינתן צומת x בעץ T, נרצה לפצל לשני עצים: $T_1 < x < T_2$. נתחיל מ־x עצמו, נתחיל לבנות את T_1, T_2 מתת העץ השמאלי והימני של x. הסיבוכיות יוצאת $O(\log n)$, כי כל Join עולה כמו ההפרש בין הגבהים. עולים למעלה וכל פעם שעלינו מהבן הימני נוסיף את תת העץ השמאלי ל־T_1 וכל פעם שעולים מהבן השמאלי נוסיף את תת העץ הימני ל־T_2.

עצי rank

פעולות חדשות: Select מחזיר את האיבר ה־k בגודלו, Rank מחזיר את המיקום של איבר בסדר הממוין.

נשמור לכל צומת **שדה חדש** שנקרא size. אפשר לעדכן אותו בכל הפעולות ולשמור על אותו זמן הריצה כי זה <u>שדה נוסף שתלוי רק בבנים הישירים של הצומת</u>.

פעולת $\text{Tree-Select}(T,k)$:

```
r ← T.left.size + 1
if k = r: return T
if k < r: return Tree-Select (T.left, k)
if k > r: return Tree-Select (T.right, k − r)
```

פעולת $\text{Rank}(x)$: נאתחל את ה־rank עם כמות הצמתים בתת העץ השמאלי ועוד 1 (x עצמו). נלך למעלה ובכל צומת שבאנו אליו מימין נוסיף גם את כמות הצמתים בתת העץ השמאלי שלו ועוד 1.

עץ finger tree: מאפשר חיפוש של האיבר ה־k בגודלו בזמן $O(\log k)$. נשמר גם <u>מצביע לאיבר המינימלי</u> בכל הפעולות. כדי למצוא את האיבר ה־k נתחיל מהמינימום ונעלה עד שיש לפחות k צמתים ואז נשתמש ב־Tree-Select רגיל.

B-tree

פרמטר d ־ כמות הבנים המינימלית (חוץ מהשורש שיכול להיות פחות). בכל צומת יש בין $d-1$ ל־$2d-1$ מפתחות ובין d ל־$2d$ ילדים (יש ילד בין כל שני מפתחות, בקצה השמאלי ובקצה הימני). <u>כל העלים באותו העומק</u>, והגובה חסום על ידי $\log_d(n)$. חיפוש הוא $O(\log_2 d \cdot \log_d n) = O(\log_2 n)$.

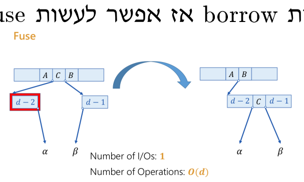
פעולת Insert מלמטה למעלה: נגיע לעלה שאליו נכניס, נוסיף את המפתח לעלה, ואז אם יש $2d$ מפתחות נפצל באופן הבא: נסמן את המפתחות בעלה ב־$a_1, \ldots, a_{2d}$. נעביר להורה את a_d, ונשים משמאל ומימין לו את $a_1, \ldots, a_{d-1}$ ואת $a_{d+1}, \ldots, a_{2d}$ עם כמות מפתחות $d-1$ ו־d. אם גם להורה יש עכשיו $2d$ מפתחות אז נפצל גם אותו ברקורסיה. פיצול של השורש הוא מיוחד: אפשר להשאיר רק איבר אחד בשורש, כך שהבן השמאלי שלו הוא $a_1, \ldots, a_{d-1}$ והימני $a_{d+1}, \ldots, a_{2d}$.

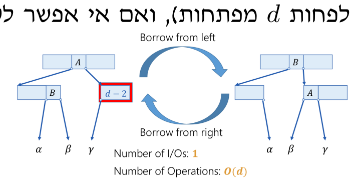
חסרונות: צריך ללכת גם למטה וגם במקרה הגרוע למעלה, צריך לזכור מי היו ההורים כשאנחנו יורדים (ב־B-tree לא שומרים את ההורים כי זה היה מייקר את הפיצול), וצריך איכשהו לייצג צומת עם $2d$ מפתחות.

פעולת Insert מלמעלה למטה: נפצל צמתים מלאים בדרך למטה (עם $2d-1$ מפתחות). ככה לא עוברים על אף צומת פעמיים, אבל עושים פיצולים מיותרים. פיצול בדרך למטה קורה מההורה: אם אנחנו רואים שאנחנו הולכים לצומת מלא נעלה את a_d שלו אלינו, נפצל אותו ל־$a_1, \ldots, a_{d-1}$ ו־$a_{d+1}, \ldots, a_{2d-1}$, ורק אז נרד.

בשני המקרים פיצול של צומת רגיל נחשב פעולת I/O אחת ופיצול של השורש נחשב 2 פעולות I/O.

פעולת Delete מלמטה למעלה: אם האיבר לא בעלה אז נחליף אותו עם ה־predecessor שלו (שנמצא בעלה "שמאל מהאיבר, הכי ימין, הכי ימין...") ונמחק את ה־predecessor. משם המחיקה היא כמו מחיקה של איבר בעלה. במקרה של underflow: נעשה borrow מהאח השמאלי או הימני (שאפשר אם יש שם לפחות d מפתחות), ואם אי אפשר לעשות borrow אז אפשר לעשות fuse.





אם עושים fuse אז יכול להיות שיש underflow ב־parent כי לוקחים ממנו איבר ואז ממשיכים לעשות אותו דבר עליו. ואם ה־fuse לוקח את האיבר היחיד שהיה בשורש אז העץ יורד רמה, ומי שהיו השני בנים שלו הם עכשיו השורש.

פעולת Delete מלמעלה למטה: בדומה ל־Insert מלמעלה למטה, נוודא שלא יהיו צמתים עם רק $d-1$ מפתחות בדרך, באמצעות אותם borrow ו־fuse. ככה לא נצטרך לעלות למעלה במחיקה, כי אם יש d מפתחות בהורה אז ה־fuse לא רקורסיבי. וגם פה אם מה שמוחקים נמצא ב־internal node אז נצטרך לרדת ל־predecessor (ולוודא בדרך שיש d מפתחות בכל צומת) ולמחוק אותו.

עצי B+: רק בעלים יש איברים, וב־internal nodes יש רק מפתחות בלי איברים.

עצי B*: כל הצמתים מלאים לפחות $2/3$, ובהכנסה במקום לעשות split נפזר את המפתחות שווה בשווה בין האחים.

עץ B rank: נשמור בכל צומת את השדה sizeSums, באינדקס ה־i יש את מספר האיברים בתתי העצים באינדקסים עד i.

ערימות

פונקציה	בינארית	בינומית	בינומית עצלה	פיבונאצ'י
	במקרה הגרוע		amortized	
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Meld/Join	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

הערה: את Delete אפשר לממש עם Decrease-key ו־Delete-min.

ערימות בינאריות

המבנה הוא **עץ כמעט שלם**, כלומר שהכל מלא חוץ מזה שברמה התחתונה חסרים צמתים מימין. לכן גם הגובה של העץ הוא $O(\log n)$. אפשר לייצג את המבנה בתור מערך, כך ש: $\text{Left}(i) = 2i, \text{Right}(i) = 2i+1, \text{Parent}(i) = \lfloor \frac{i}{2} \rfloor$ (אם המערך מתחיל מאינדקס 1).

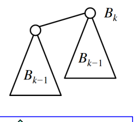
תכונת ה־min heap: המפתחות של הילדים גדולים מהמפתחות של ההורה.

פעולת Insert: נוסיף לסוף, ואז Heapify-up. **פעולת Delete-min:** נחליף את האיבר הראשון (המינימום) באיבר האחרון, ואז Heapify-down. **פעולת Delete:** כמו Delete-min, נחליף בין האיבר ה־i והאיבר האחרון ואז $\text{Heapify-down}(i)$. **פעולת Decrease-key:** פשוט Heapify-up.

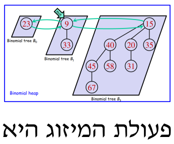
```
Heapify-up
while i > 1 and Q[i] < Q[parent(i)]
do:
    Q[i] ↔ Q[parent(i)]
    i ← parent(i)
Heapify-down
l ← left(i), r ← right(i), smlst ← i
if l < size(Q) and Q[l] < Q[smlst]
    then smlst ← l
if r < size(Q) and Q[r] < Q[smlst]
    then smlst ← r
if smlst > i then
    Q[i] ↔ Q[smlst]
    Heapify-down (Q, smlst)
```

יצירת ערימה בינארית ממערך: נפעיל על כל איבר את Heapify-down, מלמטה למעלה (נפעיל Heapify-down על צומת רק כשהתת עץ השמאלי והימני שלו הם heapים תקינים). אפשר לעשות את זה בכך שנתחיל מההורה של האינדקס האחרון (גודל ה־heap חלקי 2) ונרד באינדקסים עד ל־1, ובכך לכל רמה נעבור על $2i, 2i+1$ לפני שנעבור על i לכן זה באמת מלמטה למעלה. העלות של כל Heapify-down היא לפי הגובה, ויש לכל היותר $\frac{n}{2^h}$ צמתים בגובה h, ולכן העלות הכוללת חסומה ע"י $\sum_{h=1}^{\infty} h \cdot \frac{n}{2^h} = O(n)$.

ערימה בינומית



עץ בינומי: B_0 הוא שורש בלי כלום, B_{h+1} הוא B_h שמורכב עליו משמאל עוד B_h. לשורש יש k ילדים, יש $\binom{k}{i}$ צמתים ברמה ה־i, ובכל העץ יש 2^k צמתים.



ערימה בינומית: רשימה מקושרת של עצים בינומיים שמקיימים את תכונת ה־heap, לכל היותר אחד מכל גודל, ומצביע למי שמחזיק את המינימום כרגע.

פעולת המיזוג היא $O(\log n)$ ועובדת כמו חיבור של מספרים בינאריים, כי אפשר למזג שני עצים בינומיים מאותו הסדר בכך שנקח את האחד עם השורש היותר

גדול ואז נשים את העץ השני תחתיו. הכנסה היא מיזוג עם B_0, Delete-min מפצל עץ אחד ל־$O(\log n)$ עצים (הילדים של השורש) ומוצא מינימום חדש, ו־Decrease-key כמו בערימה בינארית עם Heapify-up.

ערימה בינומית עצלה: כמו ערימה בינומית רגילה, אבל דוחים את כל ה־linking עד ל־Delete-min. אין את הדרישה שיהיה רק עץ בינומי אחד מכל גודל. ב־Delete-min, לאחר שנמחק את השורש המינימלי ונפצל אותו לערימות נפרדות, נשמור מערך של $\log n$ מצביעים לעצים בינומיים, בהתחלה כולם ריקים, נעבור על כל הרשימה המקושרת וכל פעם שיש שני עצים מאותו הגודל נקשר אותם ונעביר אותם לגודל הבא בדומה לחיבור בינארי. תוך כדי התהליך אנחנו יודעים גם למי יש את השורש המינימלי. אחרי הפעולה נקבל ערימה בינומית רגילה, סיבוכיות $O(T_0+\log n+L)$ כאשר L כמות הlinkים שעשינו, T_0 כמות העצים ו־$\log n$ כמות העצים שנוספו עקב מחיקת השורש.

ערימת פיבונאצ'י: ב־Decrease-key, פשוט נמחק את הקשת שלא מקיימת את תנאי ה־heap, ונקבל עץ שהוא לא בינומי כבר. אפשר היה להגיע ככה לעצים רדודים ורחבים שיקרים ב־Delete-min ולכן ברגע שצומת מאבד שני בנים, ננתק גם אותו ("cascading cuts", למרות שזה רקורסיבי ב־amortized זה בסדר אם פונקציית הפוטנציאל היא $\#\text{trees}+2\cdot\#\text{marked\_nodes}$). העץ הכי מסכן הוא עם תתי עצים מגודל $1,\bar{1},2,3,5,\ldots$ (פיבונאצ'י!) ולכן לעץ פיבונאצ'י מדרגה k יש לפחות Φ^{k-2} צמתים, ואם נסתכל על הבנים $y_1,\ldots,y_k$ לפי סדר הקישור, המעלה של y_i לפחות $i-2$. אבל העומק לא חסום, אפשר לבנות שרוך מדרגה 1 כי אם מורידים בן של השורש הוא לא מסומן אלא יורד ה־rank. וזה בסדר כי מה שעולה לנו בפעולות הוא הרוחב, למשל במחיקת מינימום כשצריך להפוך את כל הבנים שלו לעצים, אבל לא העומק, כי Decrease-key הוא לא Heapify-down.

מיונים

מיוני השוואות: כל אלגוריתם דטרמיניסטי שמבוסס על השוואות הוא עץ בינארי שכל פעם שואל האם איבר קטן מאיבר אחר ובסוף מגיע לעלה שהוא תוצאת האלגוריתם. יש לפחות $n!$ עלים ולכן גובה העץ לפחות $\log_2(n!)$ שזה $\Omega(n\log n)$. בנוסף לא ניתן למיין בפחות מ־$\Omega(n\log n)$ במקרה הממוצע.

מיון Heapsort: נהפוך את המערך ל־minheap, ואז כל פעם נפעיל Delete-min ונשים את המינימום בסוף, ולסיום נהפוך את המערך. אין שום זיכרון נוסף וזה $O(n\log n)$.

מיון Quicksort: נבחר pivot אקראי, ונחלק את כל האיברים למה שגדול ומה שקטן ממנו, ואז נעשה quicksort על כל חצי בנפרד. בממוצע $O(n\log n)$ ובמקרה הגרוע $O(n^2)$. **הוכחת המקרה ממוצע:**

1. משווים כל שני מפתחות לכל היותר פעם אחת (כי מוציאים את ה־pivot)
2. ההסתברות להשוות את האיברים ה־$i<j$ בגודלם היא $\frac{2}{j-i+1}$, כי ההסתברות שהם יושוו בהינתן שה־pivot ביניהם ($i\le k\le j$), לפניהם או אחריהם היא $\frac{2}{j-i+1}$.
3. לכן מספר ההשוואות הוא $O(n\log n)$ (טור $\sum_{i=1}^{n-1}\sum_{k=1}^{n-i+1}\frac{2}{k}<\sum_{i=1}^{n-1}\sum_{k=1}^{n}\frac{2}{k}<2n(\ln n+1)$).

פרטישן Lomuto: נקדם את j, ונשמור i שעד אליו כל האיברים הקטנים מה־pivot. כל פעם שאנחנו עוברים באיבר שקטן מה־pivot נקדם את i ונעביר את האיבר למקום ה־i. **פרטישן Hoare:** הבעיה עם Lomuto זה שהוא לא יודע להתמודד טוב עם המון הופעות של ה־pivot. נתקדם משמאל עד שיש לנו איבר גדול או שווה ל־pivot, נתקדם מימין עד שיש לנו איבר קטן או שווה ל־pivot, ואז נחליף בין האיבר מימין והאיבר משמאל (עד שהם מתנגשים).

מיון count sort: עולה $O(n+R)$ עבור n מספרים בין 1 ל־R, פשוט בודקים כמה פעמים הופיע כל מספר ואז כותבים את המערך מחדש.

מיון radix sort: כמו count sort אבל עובד לפי הספרות של המספר. כל מספר הוא d ספרות בין 0 ל־$b-1$. נמיין את המספרים כל פעם עם count sort או מיון יציב אחר לפי ה־least significant digit. זה יוצא $O(d(n+b))$. לכן אפשר למיין n מספרים ב־$0,\ldots,n^k-1$ בזמן $O(kn)$ (עם בסיס n).

בעיית הבחירה: בהינתן מערך עם n איברים, למצוא את האיבר ה־k בגודלו במערך. **QuickSelect אקראי:** נבחר בכל פעם pivot אקראי, נחלק את כל האיברים למה שגדול ומה שקטן ממנו, ואז לפי כמות האיברים לפני ואחרי ה־pivot נוכל לדעת לאיזה חצי של המערך להמשיך ברקורסיה.

ניתוח סיבוכיות בתוחלת: נגדיר $n_0=n$, ו־n_i מ"מ גודל המערך (כמות ההשוואות) אחרי האיטרציה ה־i, אז $E[n_i]\le\frac{3}{4}n_{i-1}$ ולכן $\sum_i E[n_i]\le n\cdot\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i=4n=O(n)$.

חציון החציונים: דרך להפוך את QuickSelect ל־$O(n)$ במקרה הגרוע. מחלקים את המערך לחמישיות, ומוצאים חציון לכל חמישיה בנפרד. לאחר מכן לוקחים את החציונים של כל החמישיות ומוצאים להם ברקורסיה חציון עם QuickSelect-MedOfMed. אז יש לפחות $0.3n-4$ איברים שגדולים/קטנים ממנו, מלבנים...

מיון finger tree: נסמן ב־I את מספר ההיפוכים, שזה אינדקסים $i<j$ כך ש־$A[i]>A[j]$. אם נכניס את כל האיברים ל־finger tree בסדר הפוך נקבל סיבוכיות $O\left(n\log\left(\frac{I}{n}+2\right)\right)$, משתמשים בסיבוכיות amortized של הכנסות ל־AVL ועלות החיפוש שהיא $O(\log(d_i+2))$ כאשר d_i מספר האיברים אחרי האינדקס i שקטנים ממנו.

סיבוכיות amortized

הגדרה: נניח שיש לנו מבנה נתונים שתומך ב־k סוגים של פעולות $T_1,\ldots,T_k$, ונגדיר עבור פעולה op את $T(\text{op})$ להיות הסוג שלה. נאמר שסיבוכיות הזמן $\text{amort}(T_i)$ היא חסם amortized לעלות של הפעולות אם לכל סדרה חוקית של פעולות $\text{op}_1,\ldots,\text{op}_n$ מתקיים $\text{time}(\text{op}_1,\ldots,\text{op}_n)\le\sum_{i=1}^{n}\text{amort}(T(\text{op}_i))$. **שיטת ה־aggregation:** אם כולם מאותו הסוג T_j אז $\text{amort}(T_j)=\frac{1}{m}\sum_{i=1}^{m}\text{time}(\text{op}_i)$.

שיטת הבנק: כל פעולה שמפעילים יכולה לשמור מטבעות בבנק (בדרך כלל הפעולות הזולות), או לנצל מטבעות שהכניסו כבר לבנק, והסיבוכיות ה־amortized היא כמה מטבעות כל פעולה קונה.

שיטת הפוטנציאל: דומה לשיטת הבנק, אבל מתארים את כמות המטבעות בבנק עם פוטנציאל Φ שהוא המאזן של המבנה הנתונים בכל מצב נתון, כך ש־$\Phi_0=0$ ו־$\text{amort}(\text{op}_i)=\text{time}(\text{op}_i)+\Phi_i-\Phi_{i-1}$. למשל עבור מערך שמגדיל את עצמו, $\Phi(M,n)=\begin{cases}2n-M & n\ge\frac{M}{2}\\ 0 & \text{else}\end{cases}$ כאשר M גודל המערך הפיזי ו־n כמות האיברים שיש בו כרגע.

ניתוח amortized של AVL: למצוא לאן להכניס זה עדיין $O(\log n)$ אבל התיקונים בסדרה שיש בה רק הכנסות לעץ הם $O(1)$ amortized. זה כי אנחנו עולים למעלה בכל צומת ששינה את הגובה שלו, כלומר שה־BF שלו הפך מ־0 ל־±1, כמו כן אם זה ±2 אז אנחנו מפסיקים לעלות כי אחרי הכנסה יש לכל היותר תיקון אחד. בכל הכנסה יוצרים לכל היותר 3 צמתים עם $\text{BF}=0$, ולכן נשמור מטבע בבנק על כל צומת כזה... התיקונים עולים $O(1)$ amortized גם אם יש רק הכנסות ואז רק מחיקות אבל לא הוכחנו.

ניתוח amortized של B-tree: הכנסה מלמטה למעלה עולה $O(1)$ amortized לתיקונים, אם נשלם מטבע על כל צומת עם $2d-1$ מפתחות. זה נכון גם אם מערבבים הכנסות ומחיקות, התיקונים עדיין $O(1)$ amortized.

טבלאות hash

פונקציית hash: פונקציה $h:U\to[m]$ כאשר U קבוצת המפתחות האפשריים.

טבלת hash עם chaining: בכל תא נשמור רשימה מקושרת. בכל תא יש בתוחלת $\alpha=\frac{n}{m}$ איברים (פקטור העומס) בהנחה שפונקציית ה־hash מושלמת, ולכן חיפוש הוא $O(1+\alpha)$. הוספה או הכנסה הם $O(1)$ לא כולל חיפוש.

טבלת hash עם open addressing: פונקציית ה־hash מקבלת עוד קלט, $h:U\times[m]\to[m]$. במחיקה נוסיף איבר מיוחד deleted שהחיפוש מתעלם ממנו אבל הכנסה יכולה להכניס אליו. עבור פונקציית hash אידאלית חיפוש לא מוצלח הוא $O\left(\frac{1}{1-\alpha}\right)$ (כי ההסתברות ש־$\text{probes}\ge i$ היא $\frac{n}{m}\cdot\ldots\cdot\frac{n-(i-2)}{m-(i-2)}\le\alpha^{i-1}$) וחיפוש מוצלח הוא $O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$ (כמו חיפוש לא מוצלח אבל עם $\alpha'=\frac{i-1}{m}$).

דרכים מוכרות להגדיר את הפונקציה:

linear probing - $h(k,i)=(h'(k)+i)\mod m$. חיפוש לא מוצלח $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$ חיפוש מוצלח $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$.

quadratic probing - $h(k,i)=\left(h'(k)+c_1i+c_2i^2\right)\mod m$

double hashing - $h(k,i)=(h'(k)+ih_2(k))\mod m$ כאשר h_2 עם m' שזר ל־m.

פונקציות hash אוניברסליות: נאמר שמשפחת פונקציות H של $h:U\to[m]$ היא אוניברסלית אם $\Pr_{h\in H}[h(k_1)=h(k_2)]\le\frac{1}{m}$ לכל $k_1\ne k_2\in U$. עבור כל p ראשוני ו־$U=[p]$, $h_{a,b}(x)=((ax+b)\mod p)\mod m$ ו־$H_{p,m}=\{h_{a,b}\mid 1\le a<p,0\le b<p\}$ משפחה אוניברסלית.

פונקציות hash מושלמות: אם כל המילון ידוע מראש, נגריל כל פעם פונקציית hash מהמשפחה $H_{p,n}$ עד שיש פחות מ־n התנגשויות (עם 2 הגרלות בתוחלת), ואז לכל תא בטבלה עם $n_i>1$ איברים נבנה עוד פונקציית hash, $h_i\in H\left(p,n_i^2\right)$, עד שאין בה התנגשויות (עם 2 הגרלות בתוחלת). הבניה היא $O(n)$ והחיפוש הוא $O(1)$ והזיכרון הוא $O(n)$ כי $\sum\binom{n_i}{2}=|\text{Col}|<n$ לכן $\sum n_i^2=O(n)$.

$m=n$	$E[\lvert\text{Col}\rvert]<\frac{n}{2}$	$\Pr[\lvert\text{Col}\rvert<n]\ge\frac{1}{2}$
$m=n^2$	$E[\lvert\text{Col}\rvert]<\frac{1}{2}$	$\Pr[\lvert\text{Col}\rvert<1]\ge\frac{1}{2}$

Union-Find

הפעולות הן: Make-Set, Union ו־Find (שמחזיר איזשהו מאפיין יחודי של הקבוצה שאפשר להשוות בין איברים).

הסיבוכיות amortized של Union-Find עם up-tree היא $O(1)/O(\log^* n)$ ליצור קבוצה, $O(1)$ ל־union (לא כולל חיפוש) ו־$O(\alpha(n))/O(\log^* n)$ ל־Find. הסיבוכיות לרשימה מקושרת שבה לכל איבר יש מצביע לראש היא $O(1)$ ל־Make-Set, Find אבל $O(\log n)$ בשביל Union.

```
find(x)
if x.p ≠ x then:
    x.p ← find(x.p)
return x.p

Union(x,y)
link(find(x), find(y))

link(x, y)
if x.rank > y.rank then:
    y.p ← x
else
    x.p ← y
    if x.rank = y.rank then
        y.rank ← y.rank + 1

Make-Set(i)
x ← new-item()
x.info ← i, x.p ← x, x.rank ← 0
return x
```

תכונות: 1. rank של קבוצה רק עולה. 2. ה־Find ה־worst case הוא $O(\log n)$ בגלל union by rank. 3. לכל היותר $\frac{n}{2^r}$ צמתים עם $\text{rank}=r$. 4. ברגע שצומת מפסיק להיות שורש הוא לא יחזור להיות שורש. 5. ברגע שצומת הפסיק להיות שורש הוא ישאר באותה דרגה לתמיד. **הגדרה:** אם נגדיר $Tower(1)=2, Tower(n)=2^{T(n-1)}$ אז $\log^*$ הפונקציה ההופכית של $Tower$.

הוכחת סיבוכיות: נגדיר $\text{level}(x)=\log^*(x.\text{rank})$, במעמד ה־$i$ יש צמתים שדרגתם $T(i-1)+1,\ldots,T(i)$, יש לכל היותר n צמתים ממעמד 1 ולכל היותר $\frac{n}{T(i)}$ צמתים ממעמד i. ב־find משלמים רק על הפעולות שבהן $\text{level}(x)\ne\text{level}(x.p)$ ואחרת Make-Set משלם, והוא יחויב לכל היותר $T(l)$ פעמים עבור $\text{level}(x)=l$, לכן החיוב הכולל הוא $2n+\sum_{i=2}^{\log^* n-1}\frac{n}{T(i)}\cdot T(i)=n\log^* n$, כי אפשר לשנות הורה $T(i)-(T(i-1)+1)<T(i)$ פעמים, ויש $\frac{n}{T(i)}$ צמתים ברמה i, ועבור $i=1$ כל אחד יכול לשנות הורה פעמיים.

שונות

נוסחת סטירלינג: $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n<n!<\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}\right)$ ולכן $\log_2 n!>n\log_2 n-(\log_2 e)n$

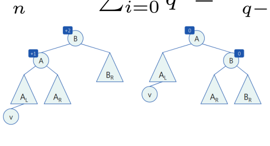
סימן (או $g=\Omega(f)$)	הגדרה	גבולי
$f=O(g)$	$\exists c.\exists n_0.\forall n\ge n_0.f(n)\le c\cdot g(n)$	$\limsup\frac{f(n)}{g(n)}<\infty$
$f=o(g)$	$\forall c.\exists n_0.\forall n\ge n_0.f(n)\le c\cdot g(n)$	$\lim\frac{f(n)}{g(n)}=0$

שיטת המאסטר: $T(n)=a\cdot T\left(\frac{n}{b}\right)+f(n)$. 1: $f(n)=O\left(n^{\log_b a-\varepsilon}\right)$ אז $T(n)=\Theta\left(n^{\log_b a}\right)$ 2: $f(n)=\Theta\left(n^{\log_b a}\right)$ אז $T(n)=\Theta\left(n^{\log_b a}\cdot\log n\right)$ 3: $a\cdot f\left(\frac{n}{b}\right)\le c\cdot f(n)$ עבור $c<1$ כלשהו אז $T(n)=\Theta(f(n))$

מאסטר 2: $T(n)=cn+T(\alpha_1n+\beta_1)+T(\alpha_2n+\beta_2)$, $0<\alpha_1,\alpha_2<1$, 1: $\alpha_1+\alpha_2<1\implies T(n)=\Theta(n)$ 2: $\alpha_1+\alpha_2=1\implies T(n)=\Theta(n\log n)$ 3: $\alpha_1+\alpha_2>1\implies T(n)=\Theta(n^p)$ כאשר $\alpha_1^p+\alpha_2^p=1$ וחסום על ידי $O(n^2)$.

$$\sum_{i=0}^{\infty}q^i=\frac{1}{1-q}\qquad\sum_{i=0}^{n}q^i=\frac{q^{n+1}-1}{q-1}\qquad\sqrt[n]{a_1\cdot\dots\cdot a_n}\le\frac{a_1+\cdots+a_n}{n}$$

רוטציה ימנית ב־AVL:



(הסבר ב"עץ AVL")