<u>עבודה במבני נתונים 4</u>

מגישים: ניב לוי 204416846 דין אברג'יל 308365881

<u>שאלה 1</u>

נחשב תחילה את גודל העץ B-Tree :

מספר הצמתים הוא n. כל הצמתים מלאים (2t-1 בלוקים) ולכן בכל צומת יש 2t מצביעים כאשר גודל כל מצביע הוא ביט אחד ולכן גודל כל המצביעים בעץ הוא $n * 2t$. גודל כל בלוק הוא D, מספר הבלוקים בכל צומת הוא 2t-1 לכן גודל כל צומת הוא $D * (2t - 1)$. יש n צמתים ולכן גודל כל הבלוקים הוא $D * (2t - 1) * n$. בסה"כ $D * (2t - 1) * n + n * 2t$.

נחשב את גודל העץ MBT :

מספר הצמתים הוא n. כל הצמתים מלאים (2t-1 בלוקים) ולכן בכל צומת יש 2t מצביעים כאשר גודל כל מצביע הוא ביט אחד ולכן גודל כל המצביעים בעץ הוא $n * 2t$. כל צומת מכיל פלט של SHA1 ולכן כל צומת מכיל 20 בייטים. בסה"כ $n * 2t + 20 * n$

היחס הוא:

$$\frac{D * (2t - 1) * n + n * 2t}{n * 2t + 20 * n}$$

<u>שאלה 2</u>

התשובה היא לא.

<u>אלגוריתם הכנסה:</u>

צמתים עם סיומת B מציינים צמתים ב- B-Tree ומשתנים עם סיומת M מציינים צמתים ב- MBT.

לכל צומת ב- B-Tree יש מצביע לצומת המקביל אליו ב- MBT. כמו כן, בדומה לצמתים ב- B-Tree גם לצמתים ב- MBT יש את השדות:

<u>SplitChild(x, i)</u>

```
y ← x.c(i)
z ← AllocateNode()
zM ← AllocateMBTNode()
z.leaf ← y.leaf
z.hash = zM
z.n ← t – 1
for j = 1 to t – 1
        z.key(j) ← y.key(j+t)
```

```
if not y.leaf
	for j = 1 to t
		z.c(j) ← y.c(j+t)
		zM.c(j) ← y.hash.c(j+t)
for j = x.n + 1 downto i + 1
	x.c(j+1) ← x.c(j)
	x.hash.c(j+1) ← x.hash.c(j)
x.c(i+1) ← z
x.hash.c(i+1) ← zM
for j = x.n downto i
	x.key(j+1) ← x.key(j)
x.key(i) ← y.key(t)
x.n ← x.n + 1
y.n ← t − 1
i ← y.n
s ← y.hash.c(i+1)
while i ≥ 1
	s  =  y.key(i) + y.hash.c(i) + s
	i ← i – 1
y.hash.value = sha1(s)
i ← z.n
s ← z.hash.c(i+1)
while i ≥ 1
	s  =  z.key(i) + z.hash.c(i) + s
	i ← i – 1
z.hash.value = sha1(s)
DiskWrite(y)
DiskWrite(z)
DiskWrite(x)
```

Insert(T, k)

```
r ← T.Root
if r.n = 2t − 1
	s ← AllocateNode()
	sM ← AllocateMBTNode()
	T.root ← s
	TMBT.root ← sM
	s.leaf ←FALSE
	s.n ← 0
	s.c1 ← r
	SplitChild(s, 1)
	InsertNonfull(s, k)
else
	InsertNonfull(r, k)
```

```
InsertNonfull(x, k)
i ← x.n
if x.leaf
        while i ≥ 1 AND k < x.key(i)
                x.key(i+1) ← x.key(i)
                i ← i − 1
        x.key(i+1) ← k
        x.n ← x.n + 1
        i ← x.n
        s ← empty string
        while i ≥ 1
                s  = s  + x.key(i)
                i ← i – 1
        x.hash.value = sha1(s)
        DiskWrite(x)
Else
        while i ≥ 1 AND k < x.key(i)
                i ← i − 1
        i ← i + 1
        DiskRead(x.c(i))
        if x.c(i).n = 2t − 1
                SplitChild(x, i)
                if k > x.key(i)
                        i ← i + 1
        InsertNonfull(x.c(i), k)
        i ← x.n
        s ← x.hash.c(i+1)
        while i ≥ 1
                s  =  x.key(i) + x.hash.c(i) + s
                i ← i – 1
        x.hash.value = sha1(s)
```

<u>זמן ריצה</u>

SplitChild- נלמד בכתה שהפונקציה רצה ב- O(t) , כמו כן אנו הוספנו מספר פעולות קבועות ומספר קבוע של לולאות כך שכל אחת מהן מתבצעת לכל היותר 2t פעמים ולכן זמן הריצה של הפונקציה נשאר O(t).

InsertNonfull- נלמד בכתה ב- O(t logt n) . הוספנו לפונקציה מספר פעולות קבועות ומספר קבוע של לולאות כך שכל אחת מהן מתבצעת לכל היותר2t ולכן זמן הריצה נשאר O(t logt n).

Insert- מספר פעולות קבוע והרצת **InsertNonfull.**

<u>סיבוכיות מקום</u>

סיבוכיות המקום של הפונקציה O(n) : ישנם n צמתים בעץ הראשי ובכל צומת יש לכל היותר 2t-1 מפתחות, 2t מצביעים, מצביע להורה ומצביע לעץ המקביל. בעץ mbt ישנם n צמתים כך שבכל צומת

חתימה של 20 בייטים ולכל היותר 2t מצביעים לילדים. מבחינת פניות לדיסק, אנו סורקים את העץ לגובהו, כלומר סורקים סדר גודל של logt n צמתים ולכל סריקת צומת יש פנייה לדיסק.

<u>אלגוריתם מחיקה:</u>

לשם מימוש פונקציית delete נוסיף לכל BNode שימוקם בעץ ה-BTree שדה נוסף. השדה יהיה מצביע להורה שלו ויסומן x.p (עבור x BNode כלשהו). כמו כן לכל BNode יהיה מצביע לצומת המתאים בעץ ה-MBT שיכיל את החתימה המתאימה לו, נסמן מצביע זה ע"י x.mbt ועבור החתימה x.mbtHash (עבור x BNode כלשהו).

- אם בשורש יש בלוק אחד וגם השורש הוא עלה, אז root=null.
- אחרת, אם מספר הבלוקים בשורש הוא 1 וגם השורש לא עלה וגם בשני הילדים של השורש יש בדיוק t – 1 בלוקים, נבצע merge לשני הילדים, הצומת המאוחד יהיה השורש החדש ונבצע delete לשורש.
- אחרת, נבצע מחיקה של-key ע"י הפונקציה delete לשורש.

Delete

נסרוק את הבלוקים של הצומת

- אם הבלוק אותו אנו רוצים למחוק לא נמצא בצומת:
  - אם הצומת הוא עלה, הבלוק המיועד למחיקה לא נמצא ולכן הפונקציה תפסיק לרוץ.
  - אחרת, נעבור לבן המתאים(מתאים – הבלוק משמאלו קטן מ-key והבלוק מימינו גדול מ-key) לאחר שוידאנו כי מספר הבלוקים בו הוא לפחות t.
    - אם מספר הבלוקים בבן זה הוא בדיוק t – 1
      - אם יש לו אח ישיר ימני או אח ישיר שמאלי שיש לו לפחות t בלוקים נבצע shifting איתו.
      - אחרת, נבצע merge שלו עם אחד האחים הישירים.
- אחרת, כלומר הבלוק אותו אנו רוצים למחוק נמצא בצומת:
  - אם הצומת הוא עלה, נמחק את הבלוק מהצומת ונקטין את שדה מספר הבלוקים בצומת ב-1. כמו כן, נעדכן את הבלוק המקביל בעץ MBT ע"י הפונקציה hash ונבצע עדכון מהעלה לשורש ע"י הפונקציה updateHash.
  - אחרת, אם הצומת הוא צומת פנימי והבן שמשמאל לבלוק אותו אנו רוצים למחוק יש לפחות t בלוקים, נמצא את בלוק ה- predecessor של הבלוק שנמצא בתת העץ של הבן השמאלי לבלוק ע"י הפונקציה findPredecessor , נציב במקום הבלוק המיועד למחיקה את ה-predecessor ונבצע מחיקה של predecessor לבן השמאלי לבלוק באופן רקורסיבי ע"י הפונקציה delete.
  - אחרת, אם הצומת הוא צומת פנימי והבן שמימין לבלוק אותו אנו רוצים למחוק יש לפחות t בלוקים, נמצא את בלוק ה- successor של הבלוק שנמצא בתת העץ השל הבן השמאלי לבלוק ע"י הפונקציה findSuccessor , נציב במקום הבלוק המיועד למחיקה את ה-successor ונבצע מחיקה של successor לבן הימני לבלוק באופן רקורסיבי ע"י הפונקציה delete.
  - אחרת, כלומר בבן הימני לבלוק המיועד למחיקה ובבן השמאלי לו יש בדיוק t – 1 בלוקים, נבצע merge ביניהם, ונבצע מחיקה לאותו key באופן רקורסיבי ע"י הפונקציה delete.

byte hash(BNode node): הפונקצייה מקבלת קוד' בעץ ומחזירה את החתימה שלו. נניח של-node יש n מפתחות.

- אם הצומת הוא עלה נשרשר את המפתחות שלו ונשלח לפונקציית ה-hash ונחזיר את הפלט שלה.
- אם הצומת אינו עלה נשרשר את $x_{c_i}.key + x_{c_i}.mbtHash$ לכל $1 \leq i \leq n$ ולבסוף נשרשר את $x_{c_n}.mbtHash$, נשלח לפונקציית ה-hash את ונחזיר את הפלט שלה.

זמן ריצה: אם ה-node שיוכנס הוא עלה אז הפעולה תיקח O(t), כיוון ששרשור העלים לוקח O(t) ופונקציית ה-hash היא פעולה קבועה. אחרת, נשרשר את הבנים ואת המפתחות בסדר המתאים, פעולה שלוקחת גם כן O(t) מפני שכדי לקבל חתימה של הבנים נשתמש במצביע המתאים לצומת ב-MBT. לאחר מכן נשלח לפונקציית ה-hash שהיא פעולה קבועה. לכן בסה"כ עבור שני המקרים נקבל שזמן הריצה הוא O(t).

updateHash(BNode node): הפונקציה מקבלת קוד' בעץ, מעדכנת בתחילה את החתימה שלו ולאחר מכן עולה במעלה העץ עד אשר מגיעה לשורש ומעדכנת את החתימה בהתאם למיקום בעץ.

- נעדכן בתחילה את החתימה של node ע"י פונקציית hash ונכניס למקום המתאים לו ב-MBT (בעזרת המצביע לצומת הנ"ל בעץ ה-MBT).
- נגדיר node = node.p כל עוד node != root ונמשיך ככה עד שנגיע לשורש.
- עבור כל node כזה נעדכן את החתימה שלו ע"י פונקציית hash ונכניס למקום המתאים ב-MBT.

זמן ריצה: במקרה הגרוע ה-node שיוכנס יהיה עלה העץ ואז נצטרך לעלות במעלה העץ עד שנגיע לשורש, דבר שלוקח $O(\log_t n)$ (כגובה העץ). עבור כל רמה שנעלה בה אנחנו מבצעים הפעלה של פונקציית hash, דבר שלוקח $O(t)$. לכן בסה"כ נקבל $O(t \log_t n)$. (t הוא כמובן קבוע)

mergeRight(int i): במקרה בו הגענו לצומת בעץ בו לשני הילדים יש t-1 מפתחות נידרש למזג אותם לצומת אחת. לשם הנוחות יהיה x צומת האב, y הצומת השמאלי ו-z הצומת הימני.

- נוסיף את המפתח ה- x.keyi לרשימת המפתחות של y.
- נעדכן את מספר המפתחות של y ושל x (x.n = x.n-1 ,y.n = y.n+1)
- נוסיף את המפתחות של z לרשימת המפתחות של y.
- נעדכן את מספר המפתחות של y (y.n = y.n + z.n).
- נעדכן את מצביעי הבנים של x: יהיה j=x.n כל עוד j != I נבצע x.cj = x.cj-1
- אם z ו-y אינם נעדכן את מצביעי הבנים של y (כולל המפתח שהגיע מ-x) כך שהמפתחות שנוספו יצביעו לבנים של z.

זמן ריצה: העברת המפתחות ועדכון הבנים לוקח O(t) כאשר t קבוע, כל שאר הפעולות לוקחות זמן קבוע ולכן בסה"כ O(t).

mergeLeft(int i): בדומה ל-mergeRight(int i) רק שהפעם נמזג את הבן השמאלי לבן הימני.

- נוסיף את המפתח ה- x.keyi לרשימת המפתחות של z.
- נעדכן את מספר המפתחות של z ושל x (x.n = x.n-1 ,z.n = z.n+1)
- נוסיף את המפתחות של y לרשימת המפתחות של z.
- נעדכן את מספר המפתחות של y (y.n = y.n + z.n).
- נעדכן את מצביעי הבנים של x: יהיה j=x.n כל עוד j != I נבצע x.cj = x.cj-1

- אם z ו-y אינם עלים נעדכן את מצביעי הבנים של z (כולל המפתח שהגיע מ-x) כך שהמפתחות שנוספו יצביעו לבנים של y.

זמן ריצה: העברת המפתחות ועדכון הבנים לוקח O(t) כאשר t קבוע, כל שאר הפעולות לוקחות זמן קבוע ולכן בסה"כ O(t).

<u>shiftRight(int i):</u> במקרה בו הגענו לצומת בעץ ומתקיים שלאח השמאלי של אותו הצומת יש לפחות t מפתחות נידרש לעשות פעולת "שיפט". לשם הנוחות יהיה x צומת האב, הצומת הימני יקרא v והצומת השמאלי יקרא u.

- נוסיף את x.keyi כמפתח המינימלי השייך ל-v.
- נעדכן את מספר המפתחות של v ושל x (v.n = v.n+1, x.n = x.n-1)
- נעביר את המפתח המקסימלי של u אל x.keyi.
- נעדכן את מספר המפתחות של u ושל x (x.xn = x.n+1, u.n = u.n-1)
- אם u ו-v אינם עלים, נעדכן את מצביעי הבנים של v ושל u כך שהבן הכי ימני של צומת u יהיה הבן הכי ימני של צומת v.
- כעת קיבלנו ש-u מכיל מספר שונה של בלוקים, לכן נעדכן את החתימה שלו ב-MBT.
- אם הוא עלה, נשרשר את המפתחות שלו ונשלח לפונקציית ה-hash, את הפלט נציב בצומת המתאים ל-u ב-MBT.
- אם הוא לא עלה נשרשר את $x_{c_i}.mbtHash + x_{c_i}.key$ לכל $1 \le i \le n$ ולבסוף נשרשר את $x_{c_n}.mbtHash$, נשלח לפונקציית ה-hash ואת הפלט נציב בצומת המתאים ל-u ב-MBT.

זמן ריצה: עדכון הבנים לוקח O(t) כאשר t קבוע, כל שאר הפעולות (עדכון החתימה ב-MBT, פעולת ה"שיפט") לוקחות זמן קבוע ולכן בסה"כ O(t).

<u>shiftLeft(int i):</u> במקרה בו הגענו לצומת בעץ ומתקיים שלאח הימני של אותו הצומת יש לפחות t מפתחות נידרש לעשות פעולת "שיפט". לשם הנוחות יהיה x צומת האב, הצומת הימני יקרא u והצומת השמאלי יקרא v. (סימטרי ל-shiftRight עד כדי שמות הצמתים).

זמן ריצה: עדכון הבנים לוקח O(t) כאשר t קבוע, כל שאר הפעולות (עדכון החתימה ב-MBT, פעולת ה"שיפט") לוקחות זמן קבוע ולכן בסה"כ O(t).

<u>**f**indPredecessor</u>

- אם הצומת הוא עלה נחזיר את הבלוק הימני ביותר.
- אחרת, נפעיל את findPredecessor באופן רקורסיבי על הבן הימני ביותר.

זמן ריצה: נשים לב שנמצא את הקודם בעלה ולכן נצטרך לרדת כגובה העץ, כלומר $O(\log_t n)$.

<u>Findsuccessor</u>

- אם הצומת הוא עלה נחזיר את הבלוק השמאלי ביותר.
- אחרת, נפעיל את findsuccessor באופן רקורסיבי על הבן השמאלי ביותר.

זמן ריצה: נשים לב שנמצא את העוקב בעלה ולכן נצטרך לרדת כגובה העץ, כלומר $O(\log_t n)$.

<u>**הערה חשובה:**</u> בזמני הריצה התייחסנו ל-t כדי להיות הדוקים ככל הניתן בחישובם. כמובן ש-t הוא מספר קבוע הנקבע ביצירת העץ ולכן למעשה אפשר גם להתייחס ל-O(t) כאל O(1).

<u>ניתוח זמן ריצה</u>

האלגוריתם רץ על העץ ובכל שלב מבצע מיזוגים או "שיפטים" בהתאם לכמות המפתחות בכל תא, פעולות שלוקחות זמן קבוע. כאשר נמצא את הקוד' אשר מכיל את המפתח אותו נרצה למחוק קיימות כמה אפשרויות:

1. אם הוא עלה, נמחק אותו. O(1)
2. אם הוא לא עלה, נחפש את העוקב/הקודם שלו - $O(\log_t n)$ נחליף ביניהם, ולבסוף באופן רקורסיבי נבצע מחיקה של העוקב/קודם.

נשים לב, בקריאות הרקורסיביות הבאות (במקרה 2) בהן נרצה למחוק את העוקב/קודם שמצאנו נבצע רק מיזוגים ו"שיפטים" ולא נבצע חיפוש עוקב/קודם פעמים נוספות (כי המחיקה עכשיו לא תהיה מצומת פנימי בעץ אלא מעלה) ולכן כל הפעולות O(t). כמו כן, המחיקה תתבצע בעלה ממנו הגיע העוקב/הקודם. לכן בסה"כ כל התהליך במקרה הזה יקח $O(\log_t n)$.

לבסוף (לאחר שמחקנו את הבלוק) נבצע את חידוש החתימה בעץ ה-MBT ע"י פונקציית updateHash. פעולה זו לוקחת גם $O(\log_t n)$.

לסיכום, אנחנו מבצעים ריצה על העץ ומבצעים בכל שלב כזה פעולות קבועות דבר שלוקח $O(\log_t n)$. מלבד שני מקרים ספציפיים (חיפוש העוקב/קודם ועדכון החתימה ב-*MBT*) בהם נבצע פעולות שלוקחות $O(\log_t n)$. לכן, עבור כל האלגוריתם נקבל זמן ריצה של $O(\log_t n)$.

<u>סיבוכיות מקום</u>

סיבוכיות המקום של הפונקציה O(n) : ישנם n צמתים בעץ הראשי ובכל צומת יש לכל היותר 2t-1 מפתחות, 2t מצביעים, מצביע להורה ומצביע לעץ המקביל. בעץ mbt ישנם n צמתים כך שבכל צומת חתימה של 20 בייטים ולכל היותר 2t מצביעים לילדים. מבחינת פניות לדיסק, אנו סורקים את העץ לגובהו, כלומר סורקים סדר גודל של logt n צמתים ולכל סריקת צומת יש פנייה לדיסק.

<u>שאלה 3</u>

מכיוון שאחרת עלולים להיות מקרים של false negative, כלומר לא בטוח שבמספר הרצות של הפונקציה על אותו קלט נקבל אותו פלט. כמו כן, הערכים מפוסמים על מנת להבטיח למשתמש כי מי שכתה את SHA-1 לא השאיר דלת אחורית לפונקציה.