אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב
סמסטר אביב 2018

סיכום קצר בנושא מבני נתונים

לעיתים קרובות נרצה לשמור בזיכרון נתונים שיעזרו לנו במימוש אלגוריתמים ותכניות לצרכים שונים. לשם כך קיימים מבני נתונים שונים. כל מבנה מאחסן את הנתונים בצורה שונה ומאפשר פעולות שונות ביעילות משתנה. הפעולות שבהן התמקדנו הן חיפוש הכנסה ומחיקה של נתונים. כאשר אנחנו צריכים לבחור באיזה מבנה נתונים להשתמש, ניקח בחשבון את הפעולות שנרצה לעשות ביעילות הרבה ביותר ונבחר במבנה שמאפשר לנו למזער את סיבוכיות הזמן שלהן וכן את סיבוכיות הזיכרון הנגזרת מכך.

הנה סיכום של מימושי מבני הנתונים שלמדנו, עם המאפיינים הבולטים של כל אחד. בכל המקרים הסיבוכיות מתייחסת למקרה הגרוע, אלא אם צוין אחרת, ומספר האיברים במבנה יצויין ע״י n.

- **רשימה רציפה בזיכרון** (נקרא גם **מערך** - array). הטיפוס list של פייתון ממומש כך. היתרון הבולט הוא שמימוש זה מאפשר גישה לאיבר לפי אינדקס נתון בזמן $O(1)$. מצד שני פעולות דינאמיות כמו הכנסה ומחיקה דורשות במקרה הגרוע $O(n)$ (עם זאת, הכנסה לסוף הרשימה מתבצעת ב- $O(1)$).

- **רשימה מקושרת (linked list)**. בדומה למערך, גם כאן ישנו סדר קווי (ליניארי) – איבר ראשון, איבר שני, וכו׳. אבל הם לא רצופים בזיכרון, אלא מכל איבר יש מצביע לאיבר הבא. היתרון הבולט – בהינתן מצביע לאיבר מסויים, אפשר להכניס אחריו או למחוק את האיבר האחריו ב- $O(1)$. מצד שני גישה לאיבר באינדקס i רצה ב- $O(i)$.

- **עץ חיפוש בינארי (Binary search tree)**. כאן המבנה אינו ליניארי אלא היררכי – לצומת יש בן ימני ובן שמאלי. ראינו שפעולות חיפוש הכנסה ומחיקה רצות בסיבוכיות שתלויה במבנה העץ. עץ מאוזן הוא עץ שגובהו $O(\log n)$, אבל בעץ כללי הגובה יכול להיות $O(n)$. המבנה של עץ נקבע על פי סדר הכנסת האיברים אליו (וגם מחיקת האיברים ממנו – אבל לא ראינו מימוש לפעולה זו כי היא מעט מסובכת).

- **טבלת ערבול (hash table)**. מאפשרת פעולות חיפוש הכנסה ומחיקה בזמן $O(1)$ **בממוצע**. הרעיון הוא זה: נניח שרוצים לשמור את פרטיהם של סטודנטים הנוכחים בשיעור מסויים, ויש n=100 כאלו. המזהה של כל סטודנט הוא מס׳ ת״ז. זהו מספר בן 8 ספרות (הספרה התשיעית היא ספרת ביקורת). לכן יש 10^8 מפתחות אפשריים.

יש לנו שתי אפשרויות:

אוניברסיטת תל אביב - בית הספר למדעי המחשב
מבוא מורחב למדעי המחשב
סמסטר אביב 2018

א. לשמור טבלה בגודל מאה מיליון תאים וכל תעודת זהות תמופה ע"י ה- int שמיוצג על ידה. אולם, רק 100 תאים מתוך 10^8 יהיו בשימוש וכל השאר סתם יוותרו ריקים. אמנם זה יאפשר ביצוע פעולות ב- $O(1)$, **אבל הזיכרון יועמס בצורה בלתי סבירה, מה שעלול לפגוע בזמני הריצה בפועל, או ממש "לתקוע" את המחשב!**

ב. נתפשר על זמן הריצה כדי לקבל חיפוש והכנסה בזמן $O(1)$ **בממוצע** אך נחסוך המון זיכרון. נבנה טבלה שגודלה בערך $m=100$, כלומר בערך מספר המפתחות שרוצים לשמור. כדי לדעת היכן לשמור כל איבר (וגם לחפש אותו) מגדירים מיפוי בין קבוצת המפתחות האפשריים לקבוצת האינדקסים של טבלה $\{0,1,\ldots,m-1\}$. מיפוי כזה נקרא בהקשר שלנו פונקצית hash. בהינתן מפתח x, האינדקס בטבלה שבו הוא יישמר הוא $\text{hash}(x)$. למשל אם hash(id) = id%100 אז סטודנטית בעלת ת"ז 12345678 תישמר באינדקס 78. כאשר מחפשים בטבלה איבר בעל מפתח x מחשבים את האינדקס $i=\text{hash}(x)$ ומחפשים את האיבר באינדקס זה בטבלה. אם האיבר נמצא בטבלה – הוא חייב להיות באינדקס זה (כי הרי הוא הוכנס ע"י חישוב אותו אינדקס בדיוק). הבעיה שמתעוררת היא שמיפוי כזה ממש לא חד-חד-ערכי, כלומר בוודאי שיכולות להיות התנגשויות של מפתחות ש"רוצים" להיכנס לאותו תא בטבלה. ראינו שתי גישות להתמודדות עם התנגשויות:

1) שיטת השרשראות. שומרים בכל אינדקס i של הטבלה רשימה (אפשר גם מבנה אחר) ובו כל האיברים שהוכנסו לטבלה ומופו לאינדקס זה. הרשימות האלו נקראות שרשראות, ומכאן השם של השיטה. אם לטבלה בגודל m הוכנסו n איברים אז האורך הממוצע של שרשרת הוא n/m. מסמנים גודל זה ב- α, והוא נקרא פקטור העומס. אם פונקצית ה-hash "טובה", כלומר מפזרת את המפתחות באופן די אחיד בטבלה, הרי שכל שרשרת תהיה באורך שאינו רחוק מ- n/m. ואם דאגנו מלכתחילה שהטבלה תהיה מספיק גדולה, ובפרט ש- $n < cm$ עבור קבוע c כלשהו, הרי שכל שרשרת תהיה באורך קבוע. ואז חיפוש, הכנסה ומחיקה של איבר ירוצו בזמן קבוע. כל זה כאמור תלוי בפיזור של פונקציית ה-hash, ולכן במקרה הגרוע סיבוכיות הפעולות עדיין $O(n)$. זה קורה למשל במקרה הקיצוני בו פונקצית ה- hash היא קבועה ($\text{hash}(x) = k$ עבור קבוע k, כלומר כל האיברים ממופים לאותו אינדקס בטבלה). בפועל יש פונקציות hash טובות מאוד. למשל זו של פייתון. במקרים ספציפיים, בהם יש לנו מידע על המפתחות, אפשר "לתפור" פונקציית hash טובה אחרת.

2) Couckoo hashing. בשיטה זו האיברים נשמרים בתוך הטבלה עצמה, ולא במבנה כמו רשימה ש"יוצא" מכל תא שלה. במקום להשתמש בפונקציית hash אחת, משתמשים בכמה, למשל ארבע, h_1, h_2, h_3, h_4. כלומר לכל איבר x יש 4 מקומות בטבלה שבהם הוא יכול להישמר: $h_1(x), h_2(x), h_3(x), h_4(x)$. כאשר באים להכניס איבר x לטבלה, עוברים על אידקסים אפשריים אלו לפי הסדר, והראשון הפנוי יהיה זה שאליו ייכנס האיבר.

אבל מה עושים אם אין מקום פנוי, כלומר כל האינדקסים $h_1(x), h_2(x), h_3(x), h_4(x)$ כבר תפוסים ע״י איברים שהוכנסו קודם? מנסים להזיז את אחד האיברים הללו לאחד משלושת המקומות האחרים האפשריים עבורו. למשל, נניח שבאינדקס $h_1(x)$ שמור איבר y. המקומות האפשריים עבור y הם $h_1(y), h_2(y), h_3(y), h_4(y)$. אחד מהם הוא האינדקס בו הוא שמור כרגע. אז בודקים את שלושת האחרים לפי הסדר, ואם אחד מהם פנוי מזיזים לשם את y ומכניסים את x למקום שהתפנה. אם אין מקום פנוי אחר עבור y, עוברים לנסות את מזלנו בהזזת האיבר ששמור ב- $h_2(x)$, וכך הלאה. מכיוון שלכל איבר כזה יש 3 מקומות אלטרנטיביים, בודקים סה״כ $4 \cdot 3 = 12$ אינדקסים בשלב הזה (לא בהכרח שונים). מעין משחק ״כיסאות מוזיקליים״. אם אף אחד מהם לא פנוי, יש לנו שתי אפשרויות להמשך: להיכנס ״לעומק״ ולנסות להזיז את אחד האיברים באחד מ- 12 האינדקסים הללו, ובכך להגדיל את מספר האינדקסים שמשתתפים במשחק ל- $4 \cdot 3^2 = 36$. או לוותר, ולהכניס את x לטבלה צדדית, שבה שמורים איברים ״בעייתיים״ שלא הצלחנו למצוא להם מקום. השיטה נחשבת מוצלחת מאוד מהסיבה הבאה: אם בוחרים את פונקציות ה- hash בצורה טובה, בסיכוי גבוה מאוד לא נזדקק כמעט לטבלה הצדדית. לכן בסיכוי גבוה מאוד חיפוש עובד ב- $O(1)$.