מבנ"ת 10

שחר פרץ

12 במאי 2025

מרצה: עמית ווינס

PRIORITY QUEUE ADT ..... (1)

אנחנו מחזיקים איברים עם מפתח (priority) ונרצה לתמוך בפעולות הבאות:

- הכנסת איבר x שקיימים לו $x.key,\ x.value$.
- מינימום – האיבר עם key מינימלי ("ה־priority הכי חשוב").
- הקטנת מפתח $\texttt{Decrease-Key}(x, Q, \Delta)$ כאשר Q מצביע לאיבר, ו־Δ בכמה מקטינים – זה מאפשר להגדיל חשיבות של איבר, אך דורש גישה פנימית למערך.
- למחוק את המינימום

אלו הן הפעולות הבסיסיות. יש מספר פעולות אופציונליות קצת יותר מעניינות:

- מחירת איבר $\texttt{Delete}(x, Q)$.
- מציאת איבר

ה־key דורש אך ורק קיום סדר מלא.

1.1 Implementations

נוכל לממש באמצעות AVL:

P. Queue	Insert	Retrieve-Min	Delete-Min	Decrease-Key	Delete
AVL tree	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Sorted Circular Array	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Circular Array + Min Pointer	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Binary Stack	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

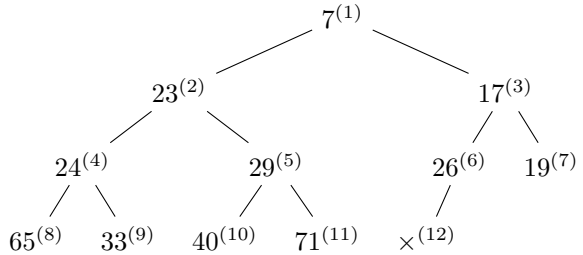
רק עץ AVL נראה נורמלי בכלל. אבל זה בא במחיר. כשמכניסים משהו ל־AVL, אין באמת טעם למיין את כל האיברים ברגע שמכניסים אותם – אולי צריך רק את 10 האיברים החשובים ביותר. זה דוקש המון מצביעים, ואפילו ב־B-trees – חבל על ה־I/O.

1.2 Faster Implementations – Binary Stack

נרצה להחליש את הדרישה של BST – המינימום יהיה למעלה בשורש, ומשני צדדיו כל האיברים יהיו יותר גדולים יותר (או שווים). כל צומת יהיה המינימום ביחס לשני תתי העצים שיוצאים ממנו. הדרישה הזו מאפשר להקטין את הסיבוכיות.

בניגוד לעץ חיפוש בינארי, בו דרשנו שכל תת־העץ יהיה יותר קטן/גדול (כתלות בצד), ולא רק הקודקוד העליון – זוהי תכונה לוקאלית. הדבר היחיד שנדרש כאן הוא לדעת מי אלו הבנים (מנגד, התכונה של BSTs גלובלית).

נרצה להוסיף קודקודים למטה, דוגמה:



להכניס איבר חדש ל־$\times$. לתהליך של ההכנסה נקראה heapify-up (כאשר heap=ערימה). ננסה להכניס ל־$\times$, אם זה לא גדול יותר, נחליף אותו עם אביו, וכן הלאה.

נגיד ורוצים למחור את השורש – היינו רוצים להעיף משהו באיזור של השורש. באופן דומה לפעם הקודמת, נעיף את ה־7 ונחליף אותו במה שנמצא ב־$\times$. במקום מה שעשינו קודם, של פעפוע כלפי מעלה, נשווה של השורש החדש למטה ונפעפע את התשובה מטה. כיצד נבחר האם מעפעפים ימינה או שמאלה? לפי הגדלים, כי חייבים לקבל את התכונה. אז נעלה למעלה את המינימלי בית הילדים עד שמסיימים. זהו heapify-down.

הגדלת מפתח עובדת באופן זהה. אן מקטינים heapify up ואם מגדילים heapify down.

הייתרון – המבנה הזה ממש פשוט ומאוזן בצורה מושלמת. אין בלגנים כמו רוטציות וכאלו. האיזון גם יותר טוב – רק ברמה הנמוכה ביותר, חסרים העלים האחרונים בלבד, כלומר הגובה הוא $\log_2 n \pm 1$ (במקום ה־$1.44 \log n$ של AVLs). נחזור לטבלה מלמעלה.

אומנם לכאורה הטבלה מלמעלה מראה כאילו הסיבוכיות זהה לזו של AVLs, אבל המימוש הזה עם קבועים קטנים יותר, ואפשר ליצור ערימה כזו ב־$O(n)$. עתה גם נראה שהדבר הזה מיוצג בזכרון בצורה אלגנטית. נוכל למספר בצורה די טבעית את האיברים – ראו סימונים ב־$^{(n)}$ בעץ למעלה.

יש לנו תובנה לגבי המספור הזה – מספור הבנים הוא למעשה $2n$ ו־$2n+1$ כאשר n המספור שלי, וכאשר מתחילים למספר ב־1. זה מדהים. זה אומר שאפשר לממש ערימה באמצעות מערך, ואפשר לתחזק אותה בצורה של מערך רציף בזכרון – כדי להגיע לבן הימני, נלך לאינדקס ה־$2n+1$, וכדי להגיע לאבא, נחלק את האינדקס שלנו ונעגל למטה. ואם נרצה להוסיף איבר חדש? ניגש לאיבר האחרון במערך. המשמעות – 0 פעולות I/O (עד לכדי מקסימום דפים) כי הכל במערך רציף בזכרון, במקום עם הפניות. המערך מייצג באופן מלא את העץ והמעבר בו ב־$O(1)$.

ניסוח של המרצה: נשים לב שאם ממספרים את הצמתים לפי שכבות ולפי סדר, עבור קודקוד i, הבן הסמאלי הוא $2i$, הימני $2i+1$ והאבא $\lfloor \frac{i}{2} \rfloor$. השורש הוא הראשון, העלה האחרון הוא האינדקס המקסימלי. בהינתן זה, ניתן לממש בצורה יעילה במערך.

אנחנו נרצה לעשות heapfiy למערך כלשהו – כלומר, להפוך אותו ל־heap (ערימה). כלומר בהינתן מערך, נרצה לגרום לו לקיים את תכונת הערימה הבינארית.

העצה: נבצע heapify-down לכל הקודקודים שאינם עלים ,מהנמוך, לפי עומק, עד השורש. בסוף באמת תתקבל ערימה בינארית.

סיבוכיות? $O(h)$ כאשר h הגובה של הקודקוד. לכן לכל הצמתים שצמודים לעלים – זה יהיה $O(1)$, וכן הלאה. סה"כ הסיבוכיות:

ניסוח אחר של המרצה: עבור $\frac{n}{2}$ הקודקודים מגובה 1, ולכל היותר $2 \cdot \frac{n}{4}$ קודקודים מגובה 2, וכו':

$$\texttt{heapify-down} \leq 1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \cdots + \log n \cdot \frac{n}{n} = \sum_{h=1}^{\lfloor \log n \rfloor} h \cdot \frac{n}{2^h} \leq n \sum_{h=1}^{\inf} \frac{h}{2^h} = O(n)$$

בכל שלב, הנחנו שתת־העץ של הקודקוד שמטפלים בו הוא תקיון פרט לשורש שלו.

נמנענו מלמיין את הרשימה, ולכן זמן לינארי. נוכל להשתמש ב־heap הזה בשביל למיין, נראה את זה בראשון. ניתן לממש ערימה לא בינארית.

ניתן לממש ערימה לא בינארית, באופן דומה ל־B-trees, והסיבוכיות של רוב הפעולות תהיה $O(k \log_k n)$ וחלק $O(\log_k n)$.

אם נרצה לחבר שני heap־ים, הדבר הדי הכי טוב שאפשר לעשות הוא להקצות מערך חדש ולעשות heapification – כמו שמתואר קודם – בסיבוכיות לינארית.

המבנה הזה מאוד יעיל בפרקטיקה, למרות שפרט ליצירתו, הוא כמו AVL. יש אלגוריתמים שמצפים ש־Decrease-Key יותר מהר. לא יכול להיות שאפשר לשפר לפחות מ־$O(\log n)$ כי זה יאפשר מיון סופר מהיר, אבל לפחות לשפר את Decrease-Key אפשר, וכן איחוד ערימות יכול להיות יותר מהיר מזמן לינארי. כל זאת ועוד בשיעור הבא. נוכל גם לבנות ערימות כך ש־amoritzed דברים הם יותר טובים. בפרקטיקה ערימה בינארית הוא המבנה הכי פשוט ומהיר כדי לעשות את כל זה.

.....

שחר פרץ, 2025

קומפל ב־LaTeX ונוצר באמצעות תוכנה חופשית בלבד