האוניברסיטה העברית בירושלים
בית הספר להנדסה ולמדעי המחשב ע״ש רחל וסלים בנין

סדנת תכנות בשפת C ו-++C (67312) תרגיל 7
העשרה

1 שיקולים לקביעת פונקציית קיבולת הווקטור

לווקטור שלנו, כמו גם ל-std::vector, יש פונקציית קיבולת, המתארת מהי כמות האיברים המקסימלית שהוא יכול להכיל בכל רגע נתון. נגדיר את $cap_C: \mathbb{N} \cup \{0\} \times \mathbb{N} \to \mathbb{N}$ להיות פונקציית הקיבולת של הווקטור, כך שבהינתן $size \in \mathbb{N} \cup \{0\}$ – כמות האיברים הנוכחית שהווקטור מכיל (לפני פעולת ההוספה), $k \in \mathbb{N}$ כמות האיברים שרוצים **להוסיף** לווקטור ו- $C \in \mathbb{N} \cup \{0\}$ קבוע הזיכרון הסטטי המקסימלי של הווקטור, הפונקציה תחזיר את הקיבולת המקסימלית של הווקטור.

כאשר מדובר בזיכרון סטטי ($size + k \leq C$), זה קל – הקיבולת של הווקטור היא C תמיד. עתה, מהי הקיבולת של הווקטור כשהוא חוצה את C ועובר להשתמש בזיכרון דינמי? האם בהכרח נרצה להגדיר: $capC(size, k) = \begin{cases} C & size + k \leq C \\ size + k & size + k > C \end{cases}$? נראה להלן שלא.

הנחת המוצא שלנו היא שאנחנו רוצים לשמור על זמני ריצה טובים ככול האפשר. המטרה שלנו, אפוא, תהיה שבמימוש אופטימלי פעולות הגישה לווקטור, ההוספה לסוף הווקטור וההסרה מסוף הווקטור יפעלו כולן ב- $O(1)$. אנו נתייחס רק לפעולת ההוספה לסוף הווקטור, כשזמן הריצה של פעולת הגישה ופעולת ההסרה מהסוף ייגזר משיקולים אלו באופן טריוויאלי.

תחילה, כאשר הווקטור עושה שימוש בזיכרון הסטטי, הוקצו עבורו מראש $C \cdot sizeof(T)$ בייטים שזמינים לו סטטית. מכאן ששמירת איבר חדש בסוף הווקטור תעשה בנקל ב- $O(1)$. השאלה העיקרית היא, כיצד נקבע את קיבולת הווקטור בהקצאות דינמיות? כדי להקל על הדיון, נסמן ב- $\phi: \mathbb{N} \cup \{0\} \times \mathbb{N} \to \mathbb{N}$ את פונקציית הקיבולת **עבור זיכרון דינמי**, כך ש-

$$cap_C(size, k) = \begin{cases} C & size + k \leq C \\ \phi(size, k) & size + k > C \end{cases}$$

1.1 ניסיון ראשון - $\phi(s, k) = s + k$

כדי לא להקשות על הקריאה עם משתנים נוספים, נסתכל על המקרה הפרטי $k = 1$. ההתאמה ל-k יחסית טריוויאלית. נגדיר $\phi(s) = s + 1$ כי כאמור $k = 1$. במקרה זה הנחנו, אפוא, **שקיבלת הווקטור (כשהוא משתמש בזיכרון דינמי) תהיה כמות האיברים הנוכחית שלו ועוד 1 (האיבר החדש שנוסף)**. דהיינו, ϕ יחזיר בדיוק את כמות האיברים החדשה שתהיה בווקטור, לאחר הוספת האיבר.

לפיכך, אם גודל הווקטור לפני הוספת האיבר הוא $C < size \in \mathbb{N}$, אז כשנוסיף איבר חדש לסוף הווקטור, נצטרך להקצות זיכרון מחדש, כך שעתה נקבל $(size + 1) \cdot sizeof(T)$ בייטים. על פניו, נשמע שזה דיי פשוט, לא? נבצע הקצאה דינמית שתוסיף לנו $sizeof(T)$ בייטים, נכתוב עליהם את האיבר החדש – ובא לציון גואל. או... שלא? בכל פעם שנגדיל את הווקטור, נצטרך להעתיק את איבריו. [1] [2] העתקת האיברים היא פעולה לינארית ולכן תבוצע, כמובן, ב- $O(n)$. מכאן שניסיון זה לא עומד בדרישות זמן הריצה.

---

[1] מאחר ש-T עשוי להיות אוביקט, שימוש ב-realloc לא יסייע לנו. מסיבות דומות, נזכיר שיש לתעדף שימוש באופרטורים של ++C על אלו של C ולכן אין להשתמש בפונקציות הקצאת הזיכרון של C, אלא בכלי הקצאת זיכרון של ++C.

[2] ניתן לתהות האם אין אלגוריתם שלא מצריך העתקה. כשאתם שוקלים זאת, כדאי לחשוב האם הוא עונה על שאר הדרישות שהצבנו. למשל, האם פעולת הגישה שלו ממומשת ב-$O(1)$?

1.2 ניסיון שני - $\phi(s) = (s+k)\cdot 2$

שוב, לצורכי הנוחות נסתכל על המקרה הפרטי $k = 1$. נגדיר $\phi(s) = (s+1)\cdot 2$. כלומר הגדרנו את "אסטרטגית הגדילה" של הווקטור באופן שבו נגדיל את קיבולת הווקטור כל פעם פי 2, ולכן לא נבצע הקצאה מחדש בכל פעם שהמשתמש יבקש להוסיף איבר חדש. אנו טוענים כי הגדרה זו תביא לכך שפעולת ההוספה תפעל ב-$O(1)$ לשיעורין. נגדיל ונטען טענה חזקה יותר (שתשמש אותנו עוד רגע) – יהי $m \in \mathbb{R}$ פרמטר הגדילה, כך ש- $m > 1$ (ובענייננו $m = 2$). נטען כי פעולת ההוספה תבוצע ב-$O(1)$ לשיעורין.

הוכחה: יהי וקטור עם זיכרון דינמי[3] לו פרמטר גדילה $1 < m \in \mathbb{R}$. יהי $n \in \mathbb{N}$ כמות האיברים שנרצה להוסיף לסוף הווקטור. הוספת n האיברים תדרוש $\lfloor \log_m(n) \rfloor$ הקצאות מחדש, כאשר ההקצאה ה-i תהיה פרופורציונלית ל-m^i. לכן, כל פעולת הוספה מבוצעת ב-

$$c_i = \begin{cases} m^i + 1 & \exists p \in \mathbb{N} \;\; s.t. \;\; i - 1 = m^p \\ 1 & otherwise \end{cases}$$

כן, בסך הכול, הוספת nאיברים פועלת בסיבוכיות זמן הריצה של:

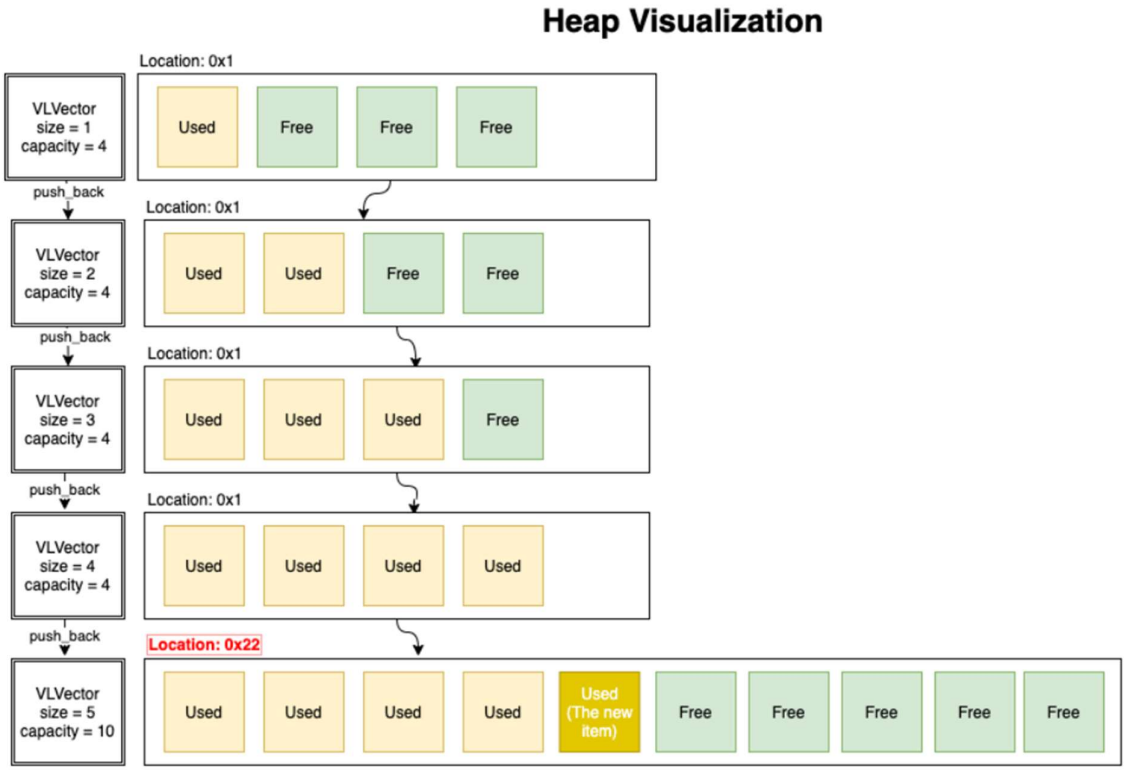
$$T(n) = \sum_{k=1}^{n} c_k \le n + \sum_{k=1}^{\lfloor \log_m(n) \rfloor} m^k \le n + \frac{m \cdot n - 1}{m - 1}$$

לפיכך, כשנחלק את $T(n)$ ב-n, עבור n פעולות הוספה, נקבל שכל פעולה מבוצעת בזמן ריצה לשיעורין של $\frac{T(n)}{n} \le \frac{n-1}{n\cdot(m-1)} + 2 \in O(1)$. ■

המסקנה מהטענה לעיל היא שבענייננו, כאשר $m = 2$, נקבל $\forall n \in \mathbb{N}: \frac{3n-1}{n} < 3$. לכן הצלחנו להגדיר את ϕ כך שתעבוד בזמן ריצה לשיעורין החסום על ידי $O(1)$. נזכיר שוב ש-ϕ הוגדרה כאן עבור $k = 1$, אך החישובים שהראנו רלוונטיים גם עבור $1 < k \in \mathbb{N}$ אחר.

הבעיה עם פרמטר הגדילה 2 היא לא זמני ריצה – אלא שימוש לא יעיל בזיכרון. נקצר ונסביר את העיקרון הכללי, מבלי להעמיק בחישוב שעומד מאחוריו. נניח שמדובר בווקטור "רגיל" (כמו std::vector, דהיינו ווקטור ללא זיכרון סטטי[4]) המחזיק ב- capacity התחלתי $C \in \mathbb{N}$. כשנידרש להגדיל את הווקטור לראשונה, כחלק מפעולת "הוספה לסוף הווקטור", הוא יצטרך לבקש ממערכת ההפעלה $2C$ בייטים חדשים לאחסון הנתונים. שימו לב לאילוסטרציה הבאה (ובפרט לכתובת בכל שלב):

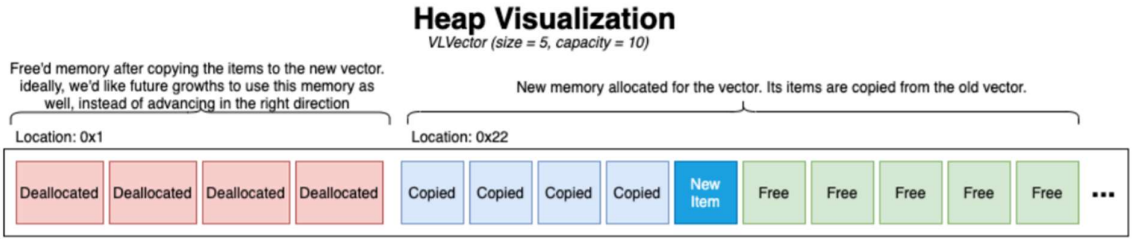
[3] לאו דווקא כזה המצויד גם בזיכרון סטטי. הוכחה זו יפה גם עבור std::vector.
[4] ההוכחה עבור וקטור שיש לו גם זיכרון סטטי, כמו המימוש שהגדרנו ל-VLVector, זהה.



במקרה הזה, נקבל שהווקטור החדש שהקצנו תופס $2C$ בייטים (כי $m = 2$), **אך <u>לפניו</u> – וזו הנקודה – ישנם C בייטים, שאותם תפס הווקטור הקודם, ושאותם נרצה לשחרר**. לכן בסוף פעולת ההכנסה יש $2C$ בייטים בשימוש על ידי הווקטור החדש, ו-C בייטים שהיו בשימוש הווקטור הישן וכעת הם deallocated.

אם כך, היכן "הבעיה" – הרי אותם C שוחררו, אזי הם זמינים לשימוש חוזר, לא? התשובה היא שכדי לעשות שימוש אפקטיבי בזיכרון, **נרצה "למחזר" זיכרון**. כלומר, נרצה להגיע מתישהו למצב שבו "צברנו" מספיק deallocated memory **רציף**, באמצעות שחרורי וקטורים קודמים, כך שביחד יוכלו להכיל מופע של וקטור גדול יותר. אם נגיע למצב כזה, נוכל "למחזר" את אותו זיכרון שעבר deallocation ולהקצות שם את הווקטור החדש, הגדול יותר. וזכרו: לא נוכל לעולם "לצרף" את אותו deallocated memory לווקטור הנוכחי, כי אנחנו רוצים להעתיק את הערכים, אז בשעת ההקצאה של הווקטור החדש, והגדול יותר, הווקטור הישן **עדיין קיים בזיכרון** ולכן לא ניתן למזג בין קטעי הזיכרון לכדי וקטור אחד.

במילים אחרות, אידאלית, היינו רוצים שהווקטור יוכל לא רק לגדול "ימינה" (כלפי זיכרון חדש, שהוא עוד לא קיבל), אלא גם "שמאלה" (כלפי זיכרון שכבר היה בשימוש בעבר, ועבר deallocation).[5]
ראו את האילוסטרציה הבאה של ה-heap, לאחר הגדלת קיבולת הווקטור:



[5] שימו לב שאנחנו דנים במצב "האידיאלי", בו בקשת הזיכרון לא "הכריחה" את מערכת ההפעלה להעביר את כל הווקטור לבלוק אחר בזיכרון (ואז כלל אין מה לשקול מקרה זה, שכן אנו מסתמכים על רציפות הזיכרון).

שימו לב לתאים המופיעים כ-deallocated. אנו נרצה לאפשר לווקטור ב-"גדילות" עתידיות להשתמש בשטח זה, שהצטבר עם הזמן, במקום לבקש זיכרון חדש ממערכת ההפעלה.

למרבה הצער, נראה שעם פרמטר גדילה של $m = 2$ זה לא יתאפשר: כאשר נחשב את הערך של C במקרה הכללי, בהינתן פרמטר הגדילה $m = 2$ נקבל:

$$\sum_{k=0}^{n} 2^k = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$$

משמעות הדבר היא כי כל הקצאת זיכרון חדשה לווקטור שנבקש ממערכת ההפעלה תהיה גדולה **ממש** מכל יתר פיסות הזיכרון שהקצנו לווקטור בעבר **ביחד**. מכאן שמערכת ההפעלה לא תוכל לעולם "למחזר" את ה-deallocated memory ששחררנו בעבר, שהרי גם כולו **יחדיו** לא מספיק לגודל החדש שנבקש. לכן, בלית ברירה, מערכת ההפעלה תצטרך "לזחול" קדימה בזיכרון ולבקש זיכרון חדש. מערכת ההפעלה לא תוכל לנצל את פיסות הזיכרון שעברו deallocation בשלבים קודמים, "לחזור אחורה" ולנצל אותן.

החישוב המלא מוביל לכך שבחירת $m < 2$ תבטיח שנוכל בשלב **כלשהו** לעשות שימוש חוזר בזיכרון ששחררנו. לדוגמה, אם נבחר $m = 1.5$ כפרמטר הגדילה נוכל להשתמש שוב בזיכרון שעבר deallocation לאחר 4 "הגדלות"; בעוד אם נבחר $m = 1.3$ נוכל לעשות שימוש חוזר בזיכרון ששוחרר בעבר לאחר 2 "הגדלות" בלבד.

1.3 ניסיון שלישי - $\phi(s) = \left\lfloor \frac{3\cdot(s+k)}{2} \right\rfloor$

המסקנה של שתי הטענות לעיל היא שנרצה לבחור פרמטר גדילה בטווח $1 < m < 2$. מצד אחד, ככול ש-m קרוב ל-1 מספר הפעמים שנוכל "למחזר" זיכרון ישן תגדל; אך כמות הפעמים שנאלץ לבצע הקצאות מחדש תגדל ולכן זמן הריצה יארך. מנגד, בחירת m שקרוב יותר ל-2 תשפר את זמני הריצה אך תמזער את כמות הפעמים שנוכל "למחזר" זיכרון ישן.

ניתן להוכיח מתמטית (נימנע מלעשות זאת כאן) שפרמטר הגדילה האופטימלי קרוב לערך של יחס הזהב, קרי $\frac{1+\sqrt{5}}{2} \approx 1.618$. מטעמים אלו, במימוש שלנו נבחר בערך 1.5 שהוא יחסית קרוב ליחס הזהב ופשוט לחישוב. בערך זה עושים שימוש במימושים רבים (למשל במימוש של ArrayList<T> ב-Java, המקביל לווקטור). נגדיר, אפוא, את ϕ בתור: $\phi(s) = \left\lfloor \frac{3\cdot(s+k)}{2} \right\rfloor$.

1.4 מסקנות

נגדיר את פונקציית הקיבולת $cap_C: \mathbb{N} \cup \{0\} \times \mathbb{N} \to \mathbb{N}$ עבור $size \in \mathbb{N} \cup \{0\}$ כמות האיברים **הנוכחית** בווקטור, $k \in \mathbb{N}$ כמות האיברים שנרצה להוסיף לווקטור בפעולת ההוספה ו-$C \in \mathbb{N} \cup \{0\}$ פרמטר (קבוע) הזיכרון הסטטי המקסימלי שזמין לווקטור, בתור:

$$cap_C(size, k) = \begin{cases} C & size + k \le C \\ \left\lfloor \frac{3 \cdot (s+k)}{2} \right\rfloor & otherwise \end{cases}$$