האוניברסיטה העברית בירושלים

בית הספר להנדסה ולמדעי המחשב ע"ש רחל וסלים בנין

סדנת תכנות בשפת C ו־C++ (67315)
מבחן מסכם – C++

תאריך הגשה: 7 לאוגוסט, 2020, בשעה 19:00.

1 הנחיות לפתרון מבחן הבית

נבקש לחדד את ההנחיות והנהלים לפתרון מבחן הבית, שמתווספים לנהלי הקורס:

- **אמינות ויושרה אקדמית:** זוהי אינה מטלת קורס. זהו מבחן לכל דבר ועניין. לכן, בהתחשב בעובדה שלא קיימת השגחה על מבחן זה כפי שהייתה לו היינו מקיימים את המבחן בבית הספר, נחמיר בצורה יוצאת מן הרגיל (ביחס לתרגילים בקורס) בבדיקת העתקות. עבדו לבד. **חל איסור לשוחח על המבחן ולחלוק חומר כלשהו הנוגע אליו. בין היתר חל איסור לחלוק אלגוריתמים ורעיונות למימוש; לשתף בהתלבטויות ותהיות; לצפות במסך של המחשב של סטודנט/ית אחר/ת שמחבר/ת את המבחן; בוודאי ובוודאי חל איסור חמור לשתף קוד מכל סוג שהוא, לרבות Unit Testing וכן חל איסור להעתיק קוד מהאינטרנט או מכל מקור חיצוני אחר** (בדיוק כפי שנעשה במבחן המתקיים בבית הספר) (מדובר ברשימה שאינה ממצה – הבנתם את העיקרון). כל שאלה הנוגעת לנוסח המבחן (ותו לא) ניתן להפנות לצוות הקורס בשעות הקבלה, שיפורסמו באתר הקורס.

- **לוח הזמנים**: לטובת פתרון מבחן הבית יעמדו לרשותכם כ־72 שעות. תכננו ונהלו את זמנכם היטב. שימו לב שאינכם מבזבזים זמן רב מדי על דברים שאינם מהותיים לתרגיל. כתבו קוד ראשוני ופשוט שעובד ורק לאחר מכן שפרו אותו – עדיף להגיש קוד ראשוני שעובר presubmission מאשר קוד שהושקעה בו מחשבה על חלק קטן מאוד, אך בכללותו אינו מצליח לעמוד בדרישות הבסיסיות של התרגיל. **אל תשכחו לגבות את עבודתכם.**

- **זמינות הסגל לשאלות ופורום בעיות טכניות**: במבחן הבית, בשונה מתרגילי הבית שהגשתם במהלך הקורס, לא יפתח פורום לשאלות טכניות או מהותיות. במקום זאת, נקיים מדי יום שעות קבלה מרוכזות. בשעות אלה תוכלו לשאול שאלות לגבי **עניינים טכניים בלבד** הקשורים למבחן (לדוגמה שאלות לגבי חוסר בהירות או ניסוח). **שאלות אחרות לא יענו (וחבל על הניסיון והזמן של כולנו...)**.
השאלות הנפוצות שישאלו בשעות הקבלה יפורסמו במרוכז באשכול מתעדכן, בחלק של המבחן באתר הקורס. **עקבו אחריו.**

- **פתרון בית הספר**: למבחן הבית לא יסופק פתרון בית הספר. באתר הקורס מצורפת דוגמה מינימלית של תוכנית העושה שימוש במחלקה שתדרשו לממש. הדוגמה מכילה תיעוד רב מאוד ומטרתה לסייע לכם בהבנת התוכנית. קראו אותה.

- **Pre-Submission:** במבחן זה ה־pre-submission script יהיה מנימילי במיוחד. מטרתו היא לוודא שקובץ ה־tar שהגשתם אכן מכיל את הקבצים הנחוצים וכי הקובץ שהגשתם אכן עובר הידור בהצלחה. לכן, יתכן ולא נבצע בדיקות תקינות כלל על הקובץ שהגשתם (ואם כן, הן יהיו מינימליות ביותר).

לכן, שימו לב שאינכם מסתמכים על תוצאת ה־presubmit בשום שלב, שכן איננה מהווה ביטחון לכך שהקוד תקין ולא נקבל ערעורים שבסיסם יהיה צליחת ה־presubmission בלבד.

- **הארכות זמן**: בניגוד לתרגילי הבית, **לא תינתן ארכה להגשת מבחן הבית בגין כל סיבה שהיא.** כשקבענו את רמת הקושי של המבחן מצד אחד, ואת כמות הזמן שהוקצתה לפתרונו אותו מצד שני, לקחנו בחשבון נסיבות אישיות, לחץ זמן בתקופת המבחנים וכל סיבה נוספת שעלולה לעלות. לכן, בפרט, גם זכאות לתוספת זמן במבחנים פרונטלים אינה מזכה בארכה להגשת מבחן זה. כמו כן, לא ניתן "לצבור" ארכה על חשבון המבחן שהגשתם ב־C או על חשבון בנק ימי החסד שהקצנו להגשת התרגילים בקורס.

- **אחסון הקוד במקומות חיצוניים:** אם אתם מאחסנים את פתרון המבחן שלכם בשיטת איחסון חיצונית – **מחובתכם לוודא** שאין לאף אדם אחר הרשאה לגשת למבחן. כך למשל, אם אתם משתמשים ב־git (על ידי פלטפורמות כדוגמת github, bitbuck-et וכו'), עליכם לוודא כי ה־repository שלכם פרטי כך שלאף אדם מלבדכם אין גישה אליו. באופן דומה, אם התיקיה שבה שמור המבחן מגובת אוטומטית באמצעות Dropbox, Google Drive וכו' – עליכם לוודא שלאף אדם אחר אין גישה לתיקיה זו. כך, **גם אם אתם פותרים את המבחן על מחשבי בית הספר, וודאו כי לאף משתמש אחר אין הרשאה לתיקית הפתרון שלכם.**

רשלנות בהגנה על פתרונכם מהווה עבירת משמעת על תקנון המשמעת האוניברסיטאי. דינם של המעתיק והמועתק – זהה. אנו נבצע בדיקות שגרתיות של אמצעי איחסון חיצוניים על מנת לשמור על טוהר הבחינות.

זכרו. מבחן בית הוא עניין של אמון. היה לנו חשוב לאפשר לכם לקבל ציון מספרי בקורס, כך שיוכל לשקף את מידת ההשקעה שלכם. אנא, אל תגרמו לנו להתעסק עם עבירות משמעת. חבל על הזמן של כולנו.

2 רקע

במבחן זה תדרשו לעשות שימוש בכלים שרכשתם במהלך הקורס, כדי לממש container חדש ויעיל במיוחד. ניזכר ב־container, שסביר כי הוא המוכר ביותר לכם – `std::vector<T>`. נרצה לממש container הזהה לו לחלוטין מבחינת התנהגות, אך חסכוני יותר בזמני ריצה. קראו היטב את ההוראות המופיעות לאורך המסמך, בפרט לאלו הנוגעות לטיפוסים מ־STL בהם מותר (או, אסור), להשתמש במבחן זה, כמו גם לנושאי יעילות. כמו כן, קראו בעיון רב את ההנחיות הנוגעות לאופן הגשת התרגיל, לנושאים הקשורים לטוהר המידות וכיוצא בזאת. מדובר במבחן לכל דבר ועניין.

3 זיכרון סטטי וזיכרון דינמי

3.1 היתרונות והקשים של ניהול זיכרון ב־Stack וב־Heap

במהלך הקורס למדנו מהן הדרכים בהם נוכל לשמור בזיכרון ערכים ומבני נתונים. בפרט, דיברנו על שני מקטעים רלבנטים – ה־stack וה־heap. בפרט, ראינו כי:

- **זיכרון סטטי (שימוש ב־Stack):** ראינו שהזיכרון ב־stack זמין לנו "כברירת מחדל" בכל פונקציה, כשלכל פונקציה ה־stack ששייך לה. כמו כן, ראינו כי מדובר ב־"זיכרון לזמן קצר", שכן בעת יציאה מהפונקציה זיכרון זה משוחרר באופן אוטומטי.

- **זיכרון דינמי (שימוש ב־Heap)**: ראינו שהזיכרון ב־heap עומד לרשותנו רק כשנבקש זאת במפורש, באמצעות בקשה להקצאת זיכרון דינמי. כמו כן, ראינו כי להבדיל משימוש בזיכרון הסטטי, הזיכרון הדינמי אינו "קיים לזמן קצר בלבד", אלא קיים עד אשר נבקש ממערכת ההפעלה לשחררו באופן מפורש (ולא – ניצור דליפת זיכרון).

הבחירה באיזה כלי להשתמש – בזיכרון סטטי או דינמי, תלויה בסיטואציה הניצבת לפנינו, ולכל כלי יש את יתרונותיו וחסרונותיו. נציג כמה מהם:

- **שימוש בזיכרון סטטי:** מצד אחד, הגישה ל־stack מהירה באופן משמעותי מגישה ל־heap, ולכן ניתן לו עדיפות. מצד שני, הזיכרון שמוקצה ב־stack זמין, כאמור "לזמן קצר" בלבד. כלומר, עת ששמורה (scope) כלשהיא מסיימת את פעולתה, המשתנים שהוגדרו עבורה ב־stack משוחררים אוטומטית – וגישה אליהם מעתה ואילך תחשב כקריאה בלתי חוקית. כמו כן, ל־stack גודל מקסימלי, שלא ניתן לחצות. למשל, כברירת מחדל, גודל המחסנית במחשבים המשתמשים ב־Windows כמערכת הפעלה הוא 1MB. ברורה, אם כן, המגבלה שבשימוש ב־stack לאחסון מידע רב.

- **שימוש בזיכרון:** מצד אחד, זיכרון דינמי מקנה לנו גמישות שכן אנו יכולים לבקש כמות גדולה הרבה יותר של זיכרון (בניגוד ל־stack, שכאמור מוגבל). ראינו שאפשר לנצל ייתרון זה בייחוד במקרים בהם איננו יודעים מהו גודל הקלט. אלא שמנגד, מאחר שמדובר ב־"זיכרון לזמן ארוך", על התוכנה לנהל את הזיכרון שאותו ביקשה ממערכת ההפעלה – וכידוע, אילו זו שוכחת לשחרר זיכרון שהקצתה, היא מביאה לכך שנוצרת דליפת זיכרון בתוכנית. יתרה מכך, כאמור לעיל, ניהול ה־heap, ובפרט הגישה לפריטים המאוחסנים בה, "מורכבת יותר". מכאן שגישה לזיכרון המאוחסן ב־heap תהא איטית יותר ותביא לכך שזמן הריצה של התוכנית שלנו יארך.

אנו נוכחים לראות כי לכל כלי הייתרונות והחסרונות שלו – ולנו האחריות להשתמש בכלים העומדים לרשותינו בתבונה.

עתה, נדבר באופן ספציפי על בעיה אחת שנתקלנו בה לאורך כל הקורס: שימוש במערכים ב־C ו־++C. עוד בתחילת הקורס ראינו כי מערך הוא למעשה קטע זיכרון **רציף** באורך n־פעמים טיפוס הנתונים המבוקש. כמו כן, ראינו כי כדי ליצור מערך עלינו לקבוע מה יהיה גודלו **בזמן קומפילציה**. כלומר, לא נוכל, למשל, לבסס את גודל המערך על קלט שקיבלנו מהמשתמש – שכן אורך המערך חייב להיות זמין למהדר עוד בזמן קומפילציה.
במצב זה, עמדו לפנינו האפשריות הבאות:

- **אם מדובר בגודל קבוע וידוע מראש**: ניתן להקצות את המערך על ה־stack. אלא שמעבר לחסרונות שבהקצאה על ה־stack שהזכרנו לעיל, הרי החיסרון המרכזי ברור ובולט ביותר: אנו חייבים לדעת מהו הגודל המקסימלי של הקלט כדי ששיטה זו תצלח. וודאי, ניתן לבחור במספר גדול מאוד (למשל $\mathbb{N} \ni n = 10000$), אך תמיד נמצא קלט בגודל $n+1$ שאיתו התוכנית שכתבנו לא תוכל להתמודד. במילים אחרות, כל עוד הקלט לא חסום מלעיל, מדובר בשיטה בעייתית.
  נזכיר בהקשר הזה כי עוד בחלק של הקורס העוסק בשפת C, למדנו על המונח Virtual Length Array (VLA). ראינו ש־VLA הוא מערך בגודל שאינו קבוע (כלומר, לא נקבע בזמן קומפילציה, אלא בזמן ריצה) ומוקצה על ה־stack. אלא שלאור הבעיתיות המובנית שבכלי זה (ההקצאה על ה־stack, שמוגבל מאוד מבחינת זיכרון) – השימוש שלו אינו מומלץ כלל ואף אסרנו את השימוש ב־VLA במסגרת קורס זה.

- **אם הגודל אינו ידוע מראש או שאינו קבוע:** נוכל לעשות שימוש בזיכרון דינמי. אלא ששימוש זה מביא עמו את החסרונות שהזכרנו באשר לשימוש ב־heap.

שתי האופציות הללו אינן ממצות את כל סט הכלים שהיה לנו, אך אלו האפשרויות המרכזיות שבהן נתקלנו. במבחן הבית נממש מבנה נתונים המתנהג כמו ווקטור, אך מממש "מאחורי הקלעים" שיטה יעילה לניהול זיכרון, המנצלת את יתרונותיהם של ה־stack וה־heap וממזערת את חסרונותיהם – ובכך ננסה "ליהנות משני העולמות".

3.2 הגדרת טיפוס הנתונים Virtual Length Vector

נגדיר את ה־container "וקטור מאורך וירטואלי" (Virtual Length Vector, או בקצרה VLVector) להיות טיפוס נתונים גנרי, הפועל על אלמנטים מסוג T ובעל קיבולת סטטית C. ל־VLVector יהיה API[1] זהה לזה של `std::vector`, אך הייחודיות שלו ביחס לווקטור "רגיל", היא בכך שהוא עושה שימוש גם ב־stack וגם ב־heap לאחסון ערכיו.

3.3 אחסון סטטי (ב־stack) אל מול אחסון דינמי (ב־heap)

VLVector יפעל באמצעות האלגוריתם הנאיבי הבא על מנת "לתמרן" ביעילות בין שימוש ב־stack וב־heap:

- <u>הצהרה:</u> הווקטור יקבל כטיפוס **גנרי** $C \in \mathbb{N} \cup \{0\}$. C מסמן כמה איברים הוקטור יכול להכיל באופן סטטי, כלומר על ה־stack. כמו כן, כל מופע של VLVector יתפוס C ערכים **בדיוק** ב־stack.

- <u>התנהגות ראשונית:</u> כל עוד $size \leq C$, הערכים ישמרו ב־stack.

[1] תזכורת: API (Application Programming Interface) הוא מונח המתייחס, בעניינו, לרשימת הפעולות **הפומביות** של האוביקט, שאליהן ניתן לגשת. ראו: https://bit.ly/39LxnQt.

- <u>מעבר מזיכרון סטטי לזיכרון דינמי:</u> ברגע שכמות האיברים שבווקטור חוצה את C (כלומר $size > C$), הווקטור יפסיק להשתמש בזיכרון סטטי ויעבור להשתמש בזיכרון דינמי. כדי לעשות זאת, הווקטור יקצה את כמות הזיכרון הנדרשת, כמפורט בחלק הבא, **ויעתיק אליו** את כל הערכים שעד כה נשמרו על ה־stack (לא ניתן להימנע מהעתקה, ראו את הנספח למבחן לפירוט). מעתה ואילך, הווקטור ישתמש רק בזיכרון הדינמי כדי לגשת **לכל** הערכים.[2]

- <u>מעבר מזיכרון דינמי לזיכרון סטטי:</u> אם כמות הערכים ששמורים כרגע בווקטור נהייתה קטנה או שווה ל־C (כלומר $size \leq C$), למשל מאחר שערך כלשהוא נמחק מהווקטור, נעתיק חזרה את הערכים ל־stack ונחזור להשתמש בזיכרון הסטטי.

3.4 קיבולת הווקטור

כאמור, לווקטור שלנו, כמו גם ל־`std::vector`, יש פונקציית קיבולת, המתארת מהי כמות האיברים המקסימלית שהוא יכול להכיל בכל רגע נתון. נגדיר את $cap_C : \mathbb{N} \cup \{0\} \to \mathbb{N}$ להיות פונקציית הקיבולת, כך שבהינתן $s \in \mathbb{N} \cup \{0\}$ – כמות האיברים **הנוכחית** בווקטור, ו־$C \in \mathbb{N} \cup \{0\}$ – פרמטר הזיכרון הסטטי המקסימלי של הווקטור, הפונקציה תחזיר את הקיבולת המקסימלית של הווקטור.
לנגד עיננו שתי מטרות: מצד אחד, נרצה לשמור על זמני ריצה טובים ככול האפשר. כך, נרצה שפעולות הגישה לווקטור, ההוספה לסוף הווקטור והסרת האיבר שבסוף הווקטור יפעלו כולן ב־$O(1)$. מהצד השני, לא נרצה להקצות **יותר מדי** מקום, שיתבזבז לשווא.
כאשר מדובר בזיכרון סטטי ($s \leq C$), זה קל – הקיבולת של הווקטור היא C. תמיד. עם זאת, מהי הקיבולת של הווקטור כשהוא חוצה את C ועובר להשתמש בזיכרון דינמי? ניסיון נאיבי יהיה להגדיל את הווקטור כל פעם באיבר אחד. כלומר, בכל פעם שמוסיפים איבר חדש, נקצה את כל הווקטור מחדש עם $(s+1) \cdot sizeof(T)$ בייטים ולהעתיק לתוכו את איבריו של הווקטור הישן. אלא, שגישה זו פועלת בזמן ריצה של $O(n)$ ולכן אינה מתאימה.
להלן הפתרון: נגדיר את פונקצית הקיבולת $cap_C : \mathbb{N} \cup \{0\} \to \mathbb{N}$, עבור $s \in \mathbb{N} \cup \{0\}$, כמות האיברים **הנוכחית** בווקטור, ו־$C \in \mathbb{N} \cup \{0\}$, פרמטר הזיכרון הסטטי המקסימלי, כך:

$$cap_C(s) = \begin{cases} C & s+1 \leq C \\ \left\lfloor \frac{3 \cdot (s+1)}{2} \right\rfloor & otherwise \end{cases}$$

הנימוקים בבסיס הגדרה זו של cap_C מופיעים בנספח המצורף בסוף טופס המבחן. **רצוי <u>מאוד</u> שתעיינו בו ותבינו אותו. אם כן, לסיכום:**

- יש להשתמש בזיכרון סטטי כל עוד כמות האיברים בווקטור אינה חוצה את C.
- יש להשתמש בזיכרון דינמי כל עוד כמות האיברים חצתה את C.
- יש לתמוך במעבר מזיכרון סטטי לזיכרון דינמי, **ולהיפך**.
- יש לעמוד בחסם של $O(1)$ לפעולת הגישה, ההוספה/ההסרה לסוף/מסוף הווקטור.
- זיכרו שאין מנוס מהעתקת האיברים בכל הגדלה / הקטנה.

[2] לדוגמה: נניח ש־$C = 3 \wedge size = 5$. במצב זה הווקטור משתמש בזיכרון דינמי – ולכן כל איבריו, $v[0 \ldots 4]$, יאוחסו ב־heap.

- **שימו לב:** כאשר עובדים עם זיכרון דינמי, קיבולת הווקטור יכולה **רק** לגדול. לא נקטין את הווקטור כשנשתמש בזיכרון דינמי. כלומר, כשיש צורך לעשות שימוש בזיכרון דינמי, נחשב את הקיבולת מחדש רק כשמגדילים את הווקטור (כלומר רק כתוצאה מהוספת איבר/ים). למען הסר ספק, נדגיש שנגדיל את הווקטור **רק** כאשר אין לנו כבר מקום. מנגד, כאשר מתעסקים עם כיווץ הווקטור (כתוצאה מהסרת איברים) – נקטין את ווקטור אם ורק אם יש לחזור לעשות שימוש בזיכרון סטטי.
כמובן, אם שוב נחצה את C ונצטרך לעשות שוב שימוש בזיכרון דינמי, נחשב את הגודל הדינמי מהתחלה (לא נעשה שימוש בערך הקודם).

- **נדגיש**: המימוש שלכם **חייב** להשתמש בהגדרת cap_C לחישוב קיבולת הווקטור בכל רגע נתון. ציונו של מימוש שיגדיל או יקטין את הווקטור בצורה שונה, או יחזיר ערכים לא תואמים – עלול להיפגע **משמעותית ביותר.**

4 המחלקה VLVector

במבחן זה הנכם נדרשים לממש, בקובץ `VLVector.hpp`, את המחלקה הגנרית VLVector. מבנה הנתונים שלכם ישמור ערכים מסוג T ועם קיבולת סטטית `StaticCapacity` (שניהם משתנים גנרים שהמחלקה מקבלת). ל־`StaticCapacity` נגדיר ערך ברירת מחדל של 16. עליכם לתמוך ב־API הבא:

	התיאור	הערות	זמן ריצה
פעולות מחזור החיים של האוביקט			
בנאי ברירת מחדל	בנאי שמאתחל VLVector ריק.		$O(1)$
בנאי העתקה	מימוש של בנאי העתקה.		$O(n)$
בנאי 1	בנאי המקבל איטרטור (מקטע $[first, last)$) של ערכי T ושומר את הערכים בוקטור. החתימה המלאה בהמשך.		$O(n)$
Destructor	מימוש Destructor.		
פעולות			
`size`	הפעולה מחזירה את כמות איברי הווקטור.	ערך החזרה מטיפוס `size_t`.	$O(1)$
`capacity`	פעולה המחזירה את קיבולת הווקטור בהתאם להגדרת cap_C שלעיל.	ערך החזרה מטיפוס `size_t`.	$O(1)$
`empty`	פעולה הבודקת האם הווקטור ריק.	ערך החזרה `bool`.	$O(1)$
`at`	פעולה מקבלת אינדקס ומחזירה את הערך המשויך לו הווקטור.	הפעולה תזרוק חריגה במקרה שהאינדקס לא תקין.	$O(1)$ (amortized)[3]
`push_back`	פעולה מקבלת איבר ומוסיפה אותו לסוף הווקטור.	הפעולה לא מחזירה ערך.	$O(1)$ (amortized)
`insert (1)`	פעולה המקבלת איטרטור המצביע לאיבר מסוים במערך (position), ואיבר חדש. הפעולה תוסיף את האיבר החדש **לפני** ה־position (משמאל ל־position).	הפעולה תחזיר איטרטור המצביע לאיבר החדש (זה שהוסף כעת).	$O(n)$

[3] זמן ריצה לשיעורין. ראו: https://bit.ly/3jSVAsQ.

insert (2)	פעולה המקבלת איטרטור המצביע לאיבר מסוים במערך (position), ו־2 משתנים המייצגים Input Iterator למקטע $[first, last)$. הפעולה תוסיף את ערכי האיטרטור **לפני** ה־position.	הפעולה תחזיר איטרטור שמצביע לאיבר החדש (זה שהוסף כעת). תוכלו להסיק כיצד יש להגדיר את $first, last$ בעזרת החתימה של בנאי (1) המופיעה בהמשך.	$O(n)$
pop_back	הפעולה מסירה את האיבר האחרון מהווקטור.	הפעולה אינה מחזירה ערך.	$O(1)$ (amortized)
erase (1)	הפעולה מקבלת איטרטור של הווקטור ומסירה את האיבר שהוא מצביע עליו.	הפעולה תחזיר איטרטור לאיבר שמימין לאיבר שהוסר.	$O(n)$
erase (2)	הפעולה מקבלת 2 משתנים המייצגים איטרטור של מופע ה־VLVector, למקטע $[first, last)$. הפעולה תסיר את הערכים שבמקטע מהווקטור.	הפעולה תחזיר איטרטור לאיבר שמימין לאיברים שהוסרו.	$O(n)$
clear	פעולה המסירה את כל איברי הווקטור.		$O(n)$
data	פעולה המחזירה **מצביע** לטיפוס הנתונים שמחזיק כרגע את המידע.	הפעולה תחזיר מצביע לטיפוס נתונים המחוזק ב־stack או ב־heap, בהתאם למצב הנוכחי של ה־VLVector.	$O(1)$
iterator	תמיכה ב־iterator (**לרבות typedefs**) בהתאם לשמות הסטנדרטים של C++.	עליכם לממש Random Access Iterator (const ו־non const).	
אופרטורים			
השמה	תמיכה באופרטור ההשמה (=).		
subscript	תמיכה באופרטור [].	האופרטור יקבל אינדקס ויחזיר את הערך המשוייך לו. **אין לזרוק חריגה במקרה זה.**	$O(1)$
השוואה	תמיכה באופרטורים == , ! =	שני ווקטורים שווים אחד לשני אם ורק אם איבריהם שווים.	

דגשים, הבהרות, הנחיות והנחות כלליות:

- החתימה לבנאי 1 היא:

```
template<class InputIterator>
VLVector(InputIterator& first, InputIterator& last);
```

- **נדגיש שוב:** על המחלקה להיות **גנרית**. הערך הגנרי הראשון הוא טיפוס הנתונים שהמחלקה מאחסנת, אליו התייחסנו כ־T. הפרמטר הגנרי השני הוא הקיבולת המקסימלית שניתן לאחסן באופן סטטי, ולה קראנו StaticCapacity (בחלק "התיאורטי" היא מסומנת כ־C). **ל־StaticCapacity יהיה ערך ברירת המחדל – 16.**

- **ניתן להניח** כי מופעים מסוג T תומכים ב־operator==, operator= וכן כי יש למופעי T בנאי דיפולטיבי ובנאי העתקה.

- **בפתרונכם אינכם רשאים לעשות שימוש באף container של STL.** קרי, ניתן לעשות שימוש בכל אלגוריתם של STL, אך אינכם רשאים לעשות שימוש ב־containers. כך, בפרט, אין לעשות שימוש ב־std::array, std::vector ו־std::list. **שימוש**

ב־containers יגרור <u>בהכרח</u> ציון 0 במבחן (וממילא אינו יכול לעמוד במלוא ההגדרות של VLVector). באופן דומה, למען הסר ספק, לא ניתן להשתמש בספריות חיצוניות.

- ה־API הנ"ל מציג לכם את שמות הפונקציות המחייבות, הפרמטרים, ערכי החזרה וטיפוסיהם. בעת מימוש ה־API, עליכם ליישם את העקרונות שנלמדו בקורס באשר לערכים קבועים (constants) ומשתני ייחוס (references). **שימוש בקונבנציות אלו הוא חלק אינטגרלי מהמבחן.** עיקרון זה נכון בפרט גם לגבי מימוש ה־iterator.

- ה־API לו אתם נדרשים זהה מבחינת הפרמטרים וערכי החזרה לזה של `std::vector`. לכן המלצתנו החמה היא כי תעיינו ותכירו היטב את ממשק זה, שכן הוא ישרת אתכם בכל התלבטות הנוגעת למימוש.[4] כפועל יוצא, במקרה שאינכם בטוחים איך המחלקה צריכה להתנהג (ובפרט, למשל, כיצד "erase (2)" פועל? מהם הפרמטרים שהוא מקבל בדיוק?) – תוכלו לעיין ב־API כאמור. כך, תוכלו להיעזר ב־`std::vector` גם כדי לבדוק כי המימוש שלכם עובד כראוי (כלומר, תוכלו ליצור לעצמכם – **ולעצמכם בלבד** – Unit Testing הנסמך על `std::vector`).

- **זמני ריצה**: זמני הריצה המפורטים לעיל הם **חסמים מלעיל.**

- <u>**שימו**</u> ♡: לפני שתיגשו לחיבור הפתרון, חישבו על **כל** הכלים שרכשתם בקורס. בפרט, כשאתם שוקלים האם האופציה X מתאימה למימוש – חישבו בין היתר איזה תכונות יש לה? היכן היא מוקצת? מה הייתרונות שלה? מה היא דורשת מכם מבחינת מימוש. חשוב לנו להדגיש: תרגיל זה מתוכנן כך שהוא נוגע במרבית החומר של הקורס. שימוש נכון בכלים שונים שלמדנו לא רק שיקצר את מרבית הפונקציות לאורך של כמה שורות בלבד, אלא יאפשר לכם לקבל "במתנה" חלק נכבד מהמימוש.

5 דוגמה – Highest Student Grade

למבחן הבית לא זמין פתרון בית ספר. במקום זאת, יצרנו עבורכם תוכנית לדוגמה,עו העושה שימוש בכמה מהתכונות הבסיסיות של הווקטור. כך, אם זו מומשה נכון, תוכלו לקמפל ולהריץ בהצלחה את התוכנית. תוכנית זו, השמורה תחת הקובץ HighestStudentGrade.cpp, מצורפת כחלק מקובצי המבחן. תוכלו לעשות בה שינויים כרצונכם, ואין להגישה עם המבחן. תוכנית זו קולטת רשימה של סטודנטים מהמשתמש, דרך ה־CLI, ולאחר מכן מדפיסה את הסטודנט עם ממוצע הציונים הגבוה ביותר. לשם כך, תוכנית זו מגדירה מחלקה בשם Student, שלה 2 שדות – "שם פרטי" ו־"ממוצע ציונים". כמו כן, לשם שמירת הסטודנטים שנקלטו על ידי המשתמש, התוכנית עושה שימוש ב־VLVector. נביט בדוגמת הרצה:

```
$ ./HighestStudentGrade
Enter a student in the format "<name> <average>" or an empty string to stop:
Mozart 70.5
Enter a student in the format "<name> <average>" or an empty string to stop:
Beethoven 95
Enter a student in the format "<name> <average>" or an empty string to stop:
Liszt 83.0
<< Note: This is an empty line >>
-----------------------------------------------------------------
```

[4] ראו: https://en.cppreference.com/w/cpp/container/vector

```
Total Students: 3
Student with highest grade: Beethoven (average: 95)
```

שימו לב שהקלט שצבוע בירוק הוא קלט שהזין המשתמש. כמו כן, השורה לפני שורת הפסים ריקה מאחר שהמשתמש הזין קלט ריק. אכן. התוכנית מזכירה "ברוחה" את התרגיל הראשון שהגשתם (Closure, huh? ☺). על כל פנים, להלן מספר דגשים:

- התוכנית מבצעת בדיקות קלט בסיסיות בלבד. תוכנית זו אינה מתיימרת להיות פתרון מלא ומקיף, אלא להציג שימוש בסיסי ב־VLVector שיצרתם.
- אנו ממליצים כי תעיינו בקפידה בתוכנית, הכוללת הערות המסבירות את הנעשה שלב שלב. תוכנית זו תוכל לסייע לכם בהבנת המשימה.
- **שימו לב:** הנכם רשאים לערוך את קובץ זה כראות עיניכם. עם זאת, נזכיר שוב שלהבדיל מהנעשה בתרגילי הבית, **במבחן בית זה אין לשתף בדיקות אוטומטיות.** הבדיקות אותם הנכם כותבים נחשבות חלק מהמבחן שלכם ("טיוטות") ושיתופן אסור בהחלט ויוביל בהכרח לפסילת המבחן ולנקיטת הליכים משמעתיים.

6 נהלי הגשת מבחן הבית

- קראו בקפידה את נוסח המבחן וכן את דף ההנחיות להגשת מבחני בית. **מדובר במבחן לכל דבר ועניין – חריגה מההנחיות תגרור מתן ציון 0. נזכיר שוב שאיננו סובלנים להעתקות. העתקות יטופלו בחומרה בהתאם לדין האוניברסיטאי.**
- עליכם ליצור קובץ tar הכולל את הקובץ VLVector.hpp ואת קובץ טוהר הבחינה (בשם STATEMENT.pdf) בלבד. ניתן ליצור קובץ tar כדרוש על ידי הפקודה:

```
$ tar -cvf exam.tar STATEMENT.pdf VLVector.hpp
```

  היצמדו לשמות הללו. **לא יבדקו כלל** מבחנים שלא יופיעו בהם כל הקבצים שלעיל או שאלו יהיו בשמות שאינם תואמים **במדויק** (למשל "exam.zip", "Statement.pdf").
- זיכרו לוודא שתרגילכם עובר קומפילציה **במחשבי בית הספר** ללא שגיאות **ואזהרות**, וכנגד מהדר בתקינה שנקבעה בקורס (C++14). אזהרות יביאו בהכח לגריעת ניקוד (בהתאם לחומרת האזהרות). מבחן שאינו עובר הידור, ינוקד בציון "נכשל". בנוסף, נזכיר שיש **לתעדף** פונקציות ותכונות של C++ על פני אלו של C. למשל, נעדיף להשתמש ב־new ו־delete על פני malloc ו־free.
- כאמור בהנחיות להגשת תרגילים – הקצאת זיכרון דינמית מחייבת את שחרור הזיכרון. **במבחן הבית, עליכם למנוע בכל מחיר דליפות זיכרון מה־container שלכם.** תוכלו להיעזר ב־valgrind כדי לאתר דליפות זיכרון. דליפות זיכרון יאבדו ניקוד **משמעותי**.
- לתרגיל זה **לא ניתן פתרון בית ספר.** כחלופה לכך, ציידנו אתכם בקובץ HighestStudentGrade.cpp, המדגים את השימוש ב־VLVector. **אין להגיש את קובץ זה.**
- אנא וודאו כי התרגיל שלכם עובר את ה־Pre-submission Script **ללא שגיאות או אזהרות**. קובץ ה־Pre-submission Script זמין בנתיב.

```
~labcc2/www/cpp_exam/presubmit_cpp_exam
```

בהצלחה!!

7 נספח – שיקולים לקביעת פונקצית קיבולת הווקטור

כאמור, לווקטור שלנו, כמו גם ל־`std::vector`, יש פונקציית קיבולת, המתארת מהי כמות האיברים המקסימלית שהוא יכול להכיל בכל רגע נתון. נגדיר את $cap_C : \mathbb{N} \cup \{0\} \to \mathbb{N}$ להיות פונקציית הקיבולת של הווקטור, כך שבהינתן $size \in \mathbb{N} \cup \{0\}$ – כמות האיברים הנוכחית שהווקטור מכיל, ו־$C \in \mathbb{N} \cup \{0\}$ – פרמטר הזיכרון הסטטי המקסימלי של הווקטור, הפונקציה תחזיר את הקיבולת המקסימלית של הווקטור.
כאשר מדובר בזיכרון סטטי ($size \leq C$), זה קל – הקיבולת של הווקטור היא C. תמיד. עם זאת, מהי הקיבולת של הווקטור כשהוא חוצה את C ועובר להשתמש בזיכרון דינמי? האם בהכרח נרצה להגדיר $cap_C\left(size\right) = \begin{cases} C & size \leq C \\ size & size > C \end{cases}$? 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 יגזר 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 יכולה להיעשות בנקל ב־$O(1)$.
השאלה העיקרית היא, כיצד נקבע את קיבולת הווקטור בהקצאות דינמיות? נסמן $\phi : \mathbb{N} \cup \{0\} \to \mathbb{N}$ את פונקציית הקיבולת עבור זיכרון דינמי, כך ש־$cap_C\left(size\right) = \begin{cases} C & size \leq C \\ \phi(size) & size > C \end{cases}$.

7.1 ניסיון ראשון – $\phi(s) = s + 1$

נגדיר $\phi(s) = s + 1$. 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 כעת הוא $C < s \in \mathbb{N}$, אזי כאשר נוסיף איבר חדש לסוף הווקטור, נצטרך להקצות זיכרון מחדש, כך שעתה נקבל $(s+1) \cdot sizeof(T)$ 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?
כל פעם שנגדיל את הווקטור, נצטרך להעתיק את איבריו.[5] [6] העתקת האיברים היא פעולה לינארית ולכן תבוצע, כמובן, ב־$O(n)$. מכאן שניסיון זה לא עומד בדרישות זמן הריצה.

7.2 ניסיון שני – $\phi(s) = (s+1) \cdot 2$

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) – אם פרמטר הגדילה הוא $m \in \mathbb{R}$ כך ש־$m > 1$ (ובענייננו $m = 2$) אזי הפעולה תבוצע ב־$O(1)$ לשיעורין.

[5] 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 יש להשתמש בכלי הקצאת זיכרון של C++.

[6] 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־$O(1)$?

הוכחה: יהי וקטור עם זיכרון דינמי[7] 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־

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

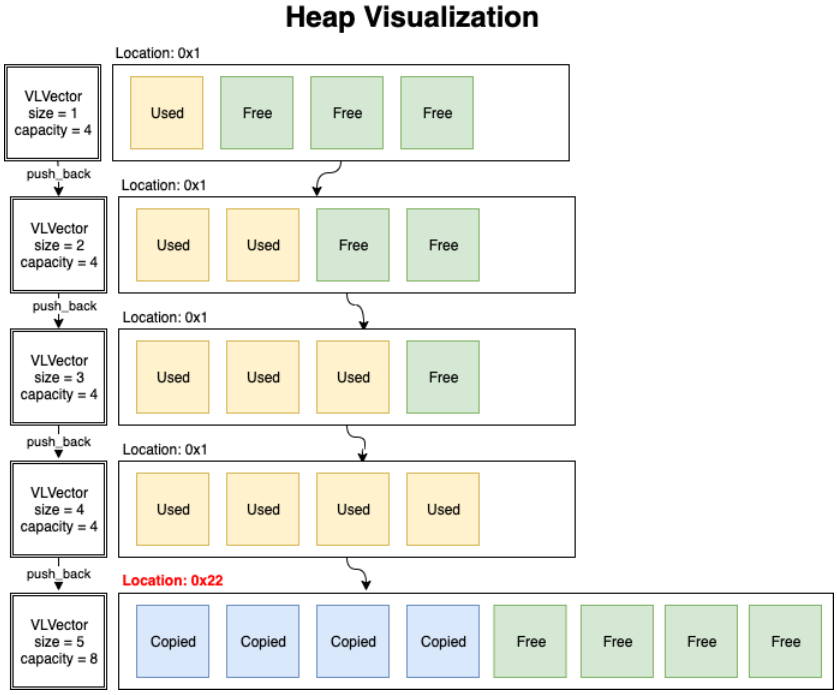
כן, בסך הכל, הוספת n איברים פועלת בסיבוכיות זמן הריצה של:

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

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

המסקנה מהטענה לעיל היא שבעניינו, כאשר $m = 2$, נקבל $\forall n \in \mathbb{N} \ \ \frac{3n-1}{n} < 3$ ולכן הצלחנו להגדיר את ϕ כך שתעבוד בזמן ריצה לשיעורין החסום על ידי $O(1)$.

הבעיה עם הערך 2 היא לא זמני ריצה – אלא שימוש לא יעיל בזיכרון. נקצר ונסביר את העיקרון הכללי, מבלי להעמיק בחישוב שעומד מאחוריו. נניח שמדובר בווקטור "רגיל" (ללא זיכרון סטטי[8]) המחזיק בקיבולת התחלתית $C \in \mathbb{N}$. 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):

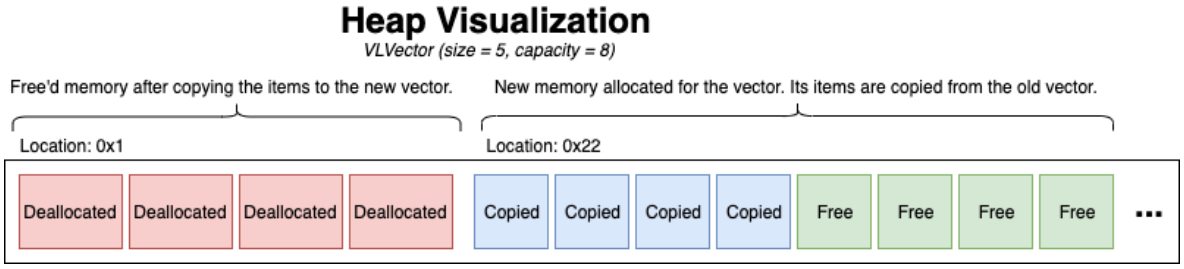


[7] לאו דווקא כזה המצוייד גם בזיכרון סטטי. הוכחה זו יפה גם עבור `std::vector`.
[8] ההוכחה עבור וקטור שיש לו גם זיכרון סטטי, כמו המימוש שהגדרנו ל־VLVector, זהה.

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 לנו $2C$ בייטים בשימוש על ידי הווקטור החדש, ו־C בייטים שהיו בשימוש על ידי הווקטור הישן וכעת הם deallocated.

אם כך, היכן "הבעיה" – הרי אותם אותם C שוחררו, אזי הם זמינים לשימוש חוזר, לא? התשובה היא שכדי לעשות שימוש אפקטיבי בזיכרון, **נרצה "למחזר" זיכרון**. כלומר, נרצה להגיע מתישהוא 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. וזיכרו: 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).[9] ראו את האילוסטרציה הבאה של ה־heap, לאחר הגדלת קיבולת הווקטור:



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 שכל 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 ששיחררנו 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.

[9] 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).

7.3 ניסיון שלישי – $\phi(s) = \left\lfloor \frac{3\cdot(s+1)}{2} \right\rfloor$

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) שהערך האידאלי לבחירה 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 מבחינת פונקציונליות). נגדיר, אפוא, את ϕ בתור: $\phi(s) = \left\lfloor \frac{3\cdot(s+1)}{2} \right\rfloor$.

7.4 מסקנות

נגדיר את פונקצית הקיבולת $cap_C : \mathbb{N} \cup \{0\} \to \mathbb{N}$, עבור $s \in \mathbb{N} \cup \{0\}$, כמות האיברים **הנוכחית** בווקטור, ו־$C \in \mathbb{N} \cup \{0\}$, פרמטר הזיכרון הסטטי המקסימלי שזמין לווקטור, בתור:

$$cap_C(s) = \begin{cases} C & s+1 \leq C \\ \left\lfloor \frac{3\cdot(s+1)}{2} \right\rfloor & otherwise \end{cases}$$

נזכיר את הנקודות הבאות הנוגעות לקיבולת הווקטור, שהובאו גם בגוף המבחן עצמו:

- יש להשתמש בזיכרון סטטי כל עוד כמות האיברים בווקטור אינה חוצה את C.
- יש להשתמש בזיכרון דינמי כל עוד כמות האיברים חצתה את C.
- יש לתמוך במעבר מזיכרון סטטי לזיכרון דינמי, **ולהיפך**.
- כאשר עובדים עם זיכרון דינמי, קיבולת הווקטור יכולה **רק** לגדול. לא נקטין את הווקטור כשנשתמש בזיכרון דינמי (ומהנימוקים שלעיל ניתן לראות את הסיבה). כלומר, כשיש צורך לעשות שימוש בזיכרון דינמי, נחשב את ϕ רק כשמגדילים את הווקטור (שנעשה כתוצאה מהוספת איבר/ים). כאשר מתעסקים עם כיווץ הווקטור (כתוצאה מהסרת איברים) – נקטין את ווקטור אם ורק אם יש לחזור לעשות שימוש בזיכרון סטטי.
  כמובן, אם שוב נחצה את C ונצטרך לעשות שוב שימוש בזיכרון דינמי, נחשב את ϕ מחדש ולא נעשה שימוש בערך הקודם.
- יש לעמוד בזמן ריצה של $O(1)$ לפעולת הגישה, ההוספה לסוף הווקטור וההסרה מסוף הווקטור.
- **נדגיש**: המימוש שלכם **חייב** להשתמש באלגוריתם הגדילה וההקטנה המתואר לעיל, ולהחזיר ערכי קיבולת זהים לאלו שמתקבלים מ־cap_C. ציונו של מימוש שיגדיל או יקטין את הווקטור בצורה שונה, או יחזיר ערכים לא תואמים – עלול להיפגע **משמעותית ביותר.**