האוניברסיטה העברית בירושלים

בית הספר להנדסה ולמדעי המחשב ע"ש רחל וסלים בנין

Programming Workshop in C & C++ - 67315

תרגיל בית 4 – שפת C
סמסטר אביב, 2020/21

מועד פרסום התרגיל: תאריך – 11\5\2021
מועד הגשת התרגיל: תאריך – 26\5\2021 בשעה - 23:59

1 רקע

בתרגיל זה תדרשו לעשות שימוש בכלים שרכשתם במהלך הקורס בכדי לממש ספריה שתכיל מבנה נתונים גנרי מסוג טבלת גיבוב (hashmap). מבנה נתונים זה יעשה שימוש מאחורי הקלעים במבנה נתונים אחר אותו תממשו (וקטור בגודל דינמי). לבסוף, תממשו חבילת טסטים (test suite) על מנת לבדוק את נכונות הספרייה שכתבתם.
קראו היטב את ההוראות המופיעות לאורך המסמך ואת ההנחיות. בנוסף, ודאו כי אתם מבינים היטב את הAPIs אותם אתם נדרשים לממש (יופיעו בקבצים שיסופקו לכם עם התרגיל).

2 הגדרות

נגדיר מספר מונחים הנוגעים לטבלאות גיבוב (לקוח מקורס מבני נתונים):

1. <u>פונקציית גיבוב (hash function)</u> – פונקציה הממפה איברים מקבוצה כלשהי (קבוצת ה-"מפתחות", שגודלה אינו מוגבל) לאיברים מקבוצה אחרת בעלת גודל סופי. לשם פשטות, ניתן להניח שמדובר בפונקציה מהצורה: $h: U \to \{0, \dots, m-1\}$ כאשר U היא קבוצת איברים כלשהי (למשל: מחרוזות, מספרים וכו'),
ו – $\{0, \dots, m-1\}$ היא קבוצה סופית של תוצאות אותן ניתן לקבל מהפונקציה h – נכנה קבוצה זו ה"תאים" של פונקציית הגיבוב. שימו לב שמעקרון שובך היונים, הפונקציה h אינה חח"ע (היא ממפה קבוצה אינסופית לקבוצה סופית).
בנוסף, נרצה שכל פונקציית גיבוב h תקיים:
- h תהיה קלה לחישוב.
- h לא תמפה הרבה איברים לאותו תא, כלומר תחלק את האיברים באופן יחסית אחיד בין התאים.
2. <u>טבלת גיבוב (hashmap)</u> – מבנה נתונים המכיל מיפוי של מפתחות לערכים. המפתחות והערכים יכולים להיות מספרים, מחרוזות, או כל טיפוס נתונים נתמך אחר. טבלת גיבוב משתמשת בפונקציית גיבוב כדי להחליט כיצד לאחסן את זוגות המפתחות והערכים. היתרון של מבנה נתונים זה הוא שבהינתן פונקציית גיבוב "טובה" (כזו שעונה על התנאים לעיל), פעולות ההוספה, החיפוש וההסרה שלו מבוצעות בסיבוכיות זמן ריצה ממוצעת של $\theta(1)$.

3 מימוש HashMap

נציג מספר מושגים נוספים הנוגעים לפונקציות, טבלאות ומפות גיבוב, להם תדרשו במימוש התרגיל:

3.1 קיבולת הטבלה (capacity): מייצגת את כמות האיברים המקסימלית שניתן לשמור במבנה הנתונים. בתרגיל זה, קיבולת הטבלה תהיה חזקה של 2, כאשר הקיבולת ההתחלתית של הטבלה תהיה 16 (מוגדר כקבוע). שימו לב – בהכרח מתקיים $\text{capacity} \geq 1$. נשנה את הקיבולת בהתאם לכמות האיברים שנידרש לאכסן (ראו 3.2).

3.2 גורם עומס (Load Factor): לבד מגודל הקלט בפועל, הביצועים של מפת הגיבוב מושפעים משני פרמטרים: גורם עומס עליון וגורם עומס תחתון (upper load factor & lower load factor). גורם העומס מוגדר באופן הבא:

$$\text{LoadFactor} = \frac{\text{M}}{\text{capacity}} \quad , \quad \text{capacity} > 0 \ \wedge \text{M} \geq 0$$

כאשר M מייצג את כמות האיברים שמבנה הנתונים מכיל ברגע נתון (בפועל), ואת הקיבולת (capacity) תיארנו בסעיף 3.1. גורם העומס העליון והתחתון מגדירים כמה ריק או מלא נסכים שמבנה הנתונים יהיה. כלומר, אם גורם העומס חוצה רף מסוים, נגדיל או נקטין את קיבולת מבנה הנתונים בהתאם, כך שמחד לא נדרוש הרבה זיכרון מיותר ומאידך נוכל להכיל את כמות האיברים שנרצה.

3.3 פונקציית הגיבוב: כאמור לעיל, עלינו לבחור פונקציית גיבוב "טובה" כדי להגיע למבנה נתונים יעיל. אנו נשתמש בפונקציית גיבוב פשוטה מאוד:

$$\text{h(x)} = \text{x mod capacity} \ , \qquad \text{capacity} \in \mathbb{N} \ \text{s.t capacity} \geq 1$$

כאשר הקיבולת זהה לאופן בו הגדרנו אותה קודם לכן. x הוא ייצוג מספרי של הערך שנרצה לשמור בטבלה. כדי להמיר מחרוזות, מספרים, וכיוצא בזה למספר שלם, נספק לכם פונקציות מתאימות (ראו API). נשים לב שאופרטור מודולו (%) ב-C לא עובד כמו האחד המתמטי. למשל, ב-C נקבל $(-3)\%7 = -3$, אך המודולו המתמטי ייתן את הערך 4. מסיבה זו ומסיבות נוספות, נחשב את המודולו באופן הבא:

$$\mathbf{v \ mod \ capacity = v \ \& \ (capacity - 1)}$$

שימו לב ש**לא** מדובר באופרטור and לוגי, אלא ב- **bitwise and**! פתרון זה מאפשר לנו לעשות את פעולת המודולו באופן מהיר ויעיל.

3.4 התמודדות עם התנגשויות: כאמור, מאופן הגדרתה פונקציית הגיבוב אינה חח"ע, ולכן לעתים ניתקל בשני ערכים x, y שונים זה מזה המקיימים $\text{h(x)} = \text{h(y)}$ – כלומר, הם ימופו לאותו תא במפת הגיבוב. ישנן שתי שיטות נפוצות לפתרון התנגשויות:

1. <u>גיבוב פתוח (Open Hashing)</u> – שיטה המאפשרת לשמור יותר מערך אחד בכל תא במפת הגיבוב. התאים, הנקראים גם "סלים" (buckets), ייבנו ממבנה נתונים נוסף (למשל מערך דינמי או רשימה מקושרת). כך, במקרה של התנגשות, האיבר המתנגש נוסף לרשימה המקושרת או המערך המייצגים את התא אליו הוא ממופה.
2. <u>גיבוב סגור (Closed Hashing)</u> – שיטה לפיה כל תא יכול להכיל רק איבר אחד. במקרים אלו, עלינו למצוא דרך אחרת להתמודד עם התנגשויות ולמפות איברים.

בתרגיל זה, נשתמש גיבוב פתוח.

4 הספרייה hashmap

הספרייה שתכתבו תממש טבלת גיבוב המבוססת שיטת גיבוב פתוח. על מנת לממש שיטה זו, כל תא בטבלת הגיבוב צריך להיות מסוגל להחזיק ערכים ממספר מפתחות שונים שפונקציית הגיבוב מיפתה לאותו תא. לכן, בחלק הראשון של המימוש, נממש מבנה נתונים נוסף שנקרא vector (סוג של מערך דינמי) ובו נשתמש על מנת לאכסן מספר ערכים שונים עבור תא כלשהו בטבלת הגיבוב. בצורה סכמתית ניתן לחשוב על זה באופן הבא: כל תוצאה של פונקציית הגיבוב ממופה לווקטור של ערכים $\{0, \dots, m-1\} \rightarrow Vector(\{val1, val2, ..\})$.

4.1 הספרייה vector: בחלק זה תעזרו בקובץ vector.h, הכולל את חתימות הפונקציות אותן תדרשו לממש ואת הקבועים שיסייעו לכם כמו גודל התחלתי ופרמטר הגדילה. *שימו לב – מימוש נכון של חלק זה קריטי להצלחת מימוש התרגיל כולו.* ה-API של חלק זה נבדק בנפרד וגם יחד עם API של ה-hashmap. אתם נדרשים לממש את כל הפונקציות המופיעות ב-API, אך כמובן שניתן להוסיף פונקציות עזר כראות עיניכם ולפי צרככם (בקובץ ה-c).

4.2 הספרייה hashmap**:** גם לחלק זה מצורף קובץ API הנקרא hashmap.h וכולל את חתימות הפונקציות אותן תדרשו לממש בחלק זה (בדומה לספרייה הvector, גם כאן תוכלו לממש פונקציות עזר נוספות בקובץ ה-c). זהו החלק העיקרי של התרגיל ועליו יעשו מרבית הבדיקות. שימו לב:

1. הספרייה חייבת להיות **גנרית** (ומכאן, גם הוקטור חייב להיות גנרי). ספרייה שתיכתב באופן שאינו גנרי תאבד נקודות רבות ללא אפשרות לערעור, שכן היא לא תעבור טסטים (אין ביכולתכם לדעת אילו טיפוסי נתונים יוכנסו ל-Pair (ראו סעיף 4.3)).
2. לצורך מימוש הספרייה hashmap, אתם נדרשים להשתמש בספרייה vector מסעיף 4.1. בנוסף, בכל סיטואציה בה תוכלו להשתמש באחת הפונקציות מה-API של הווקטור ותבחרו שלא לעשות כך, תוכלו להפסיד נקודות. אחת ממטרות ה-API היא לאפשר כתיבת קוד נקי וברור.

4.3 הממשק Pair: חלק זה <u>נתון עבורכם</u> ועליכם להשתמש בו בתרגיל. האובייקט התכנותי אותו מייצג חלק זה הינו זוג מהצורה {key: value}. זוג זה ייצג את האלמנט הגנרי שתכניסו ל-hashmap . הממשק כולל מפתח וערך גנריים. כלומר, ניתן לייצר באמצעות ממשק זה כל זוג אפשרי. דוגמא: בהינתן struct Person ניתן יהיה לייצר זוג מהצורה:

```
{id (size_t *) : person (Person *)}
```

בנוסף לקבצים pair.h ,pair.c בהם תשתמשו בקוד שלכם, נתון לכם קובץ העזר pair_char_int.h, הכולל דוגמא לממשק עבור Pair מסוג ספציפי הכולל מימוש של זוג ואת כל הפונקציות הנלוות הנדרשות.

4.4 הקובץ hash_funcs.h: לשימושכם, צירפנו מספר דוגמאות פשוטות של פונקציות hash למספר טיפוסים שונים. שימו לב – כאן הכוונה לפונקציות שמחזירות **ייצוג מספרי** של מפתח, על מנת שטבלת הגיבוב תוכל להפעיל עליו את פעולת הגיבוב שתוארה בסעיף 3.3.

5 כתיבת טסטים

כדי לבדוק את התנהגות הספרייה שכתבתם, תכתבו בתרגיל זה גם unit tests עבור כל אחת מהפונקציות העיקריות ב-hash_map. לצורך כך, תעזרו בקובץ **test_suite.h,** המכיל את החתימות של הפונקציות הנדרשות ועושה include ל-assert.h. לפי ה-API המוגדר בקובץ זה, עליכם לכתוב טסטים לספריה שלכם העומדים ב-3 תנאים בסיסיים:

1. הקוד שלכם צריך לעבור את כל הטסטים שכתבתם
2. פתרונות לא תקינים צריכים **להיכשל** ולא לעבור את הטסטים הרלוונטיים – לצורך כך, חשוב שתבדקו מקרי קצה שונים שעשויים להתרחש בעת השימוש בספרייה
3. על הטסטים לעשות שימוש בפונקציה assert המוגדרת ב-assert.h

אנו מעודדות אתכם לבדוק את הספריה שלכם על מגוון סוגים של pair, מה שידרוש מכם לעשות שימוש במגוון פונקציות גיבוב המחזירות ייצוג מספרי של טיפוסים שונים. ככל שתרצו להוסיף pair-ים ופונקציות כאלו, עליכם לממש אותם בקבצים test_pairs.h, hash_funcs.h בהתאמה. בפרט, הקובץ pair_char_int.h **לא יהיה זמין** ל-include בסביבה בה אנו בודקים את ההגשות שלכם, ואם תרצו להשתמש בזוג מסוג זה, תצטרכו להעתיק את תוכנו אל test_pairs.h.

שימו לב – משום שכחלק מהתרגיל אתם נדרשים לכתוב טסטים, בניגוד לתרגילים קודמים **אין לשתף טסטים!**

6 כתיבת Makefile

על מנת שיהיה ניתן לקמפל ביעילות ובנוחות את הספרייה שיצרתם, עליכם לכתוב makefile המכיל את הוראות הקומפילציה הבאות:

1. libhashmap.a הכוללת את כל הקבצים הדרושים להרצת ה-hashmap שלכם.
2. libhashmap_tests.a הכוללת את סוויטת הטסטים שלכם, ואת הקבצים שבהם סוויטת הטסטים שלכם עושה שימוש.
3. עליכם להגדיר את פקודות הקומפילציה השונות הנובעות מעץ התלויות הנוצר בקמפול הספריות הללו.
4. בנוסף, עליכם להגדיר שתי פקודות PHONY (כלומר, שלא גורמות ליצירת/קימפול קבצים אך כן מגדירות פקודות לקומפיילר):
   a. הפקודה all המקמפלת את שתי הספריות המתוארות לעיל
   b. הפקודה clean, המוחקת את כל קבצי a, .o. בתיקיה בה אנו עובדים
5. דגלי הקומפילציה הנדרשים לקימפול הספריות: -Wall -Wextra -Wvla -Werror -g -lm -std=c99

7 הרצת התוכנית

את הספרייה שלכם נקמפל יחד עם קבצי main.c חיצוניים שלא ראיתם, שישתמשו בפונקציות שונות מתוך ה-API אותו מימשתם. קובץ main לדוגמא מופיע בנספח למסמך זה.

בנוסף, נריץ את הטסטים שלכם על מימושים שונים של הספרייה HashMap, ונבדוק שמימושים נכונים עוברים את כל הטסטים, ואילו מימושים לא-תקינים נכשלים בטסטים המתאימים.

7 הגשת התרגיל

עליכם להעלות ל-git את הקבצים הבאים ואותם בלבד:

vector.c
hashmap.c
test_suite.c
test_pairs.h
hash_funcs.h
Makefile

8 הערות ודגשים

- אנא וודאו כי התרגיל שלכם עובר את סקריפט ה-pre-submission ללא שגיאות או אזהרות. הסקריפט זמין בנתיב:

~labcc/presubmit/ex4/run <path_to_submission>

- עליכם לבדוק כי התוכנית מתקמפלת על מחשבי ביה"ס בעזרת ה-Makefile שיצרתם, יחד עם הדגלים הרלוונטיים, ללא הערות.

- כחלק מהבדיקה האוטומטית תבדקו על סגנון כתיבת קוד. אנא ודאו כי הנכם מבינים את דרישות ה coding style של הקורס במלואן.

- הקפידו על בדיקות של דליפות זיכרון ושאר השגיאות אותן בודק ה-**valgrind**. המשקל אשר יינתן לבדיקות אלו יהיה רב ולכן וודאו זאת היטב.
  למען הסר ספק, הvalgrind אמור לרוץ באופן תקין מלא-מלא, כלומר שום שגיאה או הערה מכל סוג שהוא. אם מופיעה הערה כלשהי, גם אם אינה קשורה לדליפת זיכרון באופן ישיר, היא נחשבת כשגיאה ותהיה על כך הורדת נקודות.

- תיעוד – הקפידו על תיעוד הולם של הקוד אותו אתם כותבים.

- נראות הקוד – הקפידו על כל הדגשים התכנותיים – אורך פונקציות, שמות משתנים ושמות פונקציות אינפורמטיביים וכו'. תהיה בדיקה ידנית ואנחנו נתייחס לדגשים אלה בקפדנות.

- אנו מדגישות שוב – אין לשתף טסטים!

נספח – קובץ Main לדוגמא

```c
// Includes //

void *elem_cpy (const void *elem)
{
  int *a = malloc (sizeof (int));
  *a = *((int *) elem);
  return a;
}

int elem_cmp (const void *elem_1, const void *elem_2)
{
  return *((const int *) elem_1) == *((const int *) elem_2);
}

void elem_free (void **elem)
{
  free (*elem);
}
/////

int main ()
{
  // Insert elements to vec.
  vector *vec = vector_alloc (elem_cpy, elem_cmp, elem_free);
  for (int i = 0; i < 8; ++i)
    {
      vector_push_back (vec, &i);
    }
  vector_free (&vec);

  // Create Pairs.
  pair *pairs[8];
  for (int j = 0; j < 8; ++j)
    {
      char key = (char) (j + 48);
      //even keys are capital letters, odd keys are digits
      if (key % 2)
        {
          key += 17;
        }
      int value = j;
      pairs[j] = pair_alloc (&key, &value, char_key_cpy, int_value_cpy, char_key_cmp,
                             int_value_cmp, char_key_free, int_value_free);
    }

  // Create hash-map and inserts elements into it, using pair_char_int.h
  hashmap *map = hashmap_alloc (hash_char);
  for (int k = 0; k < 8; ++k)
    {
      hashmap_insert (map, pairs[k]);
    }

  //apply double_value on values where key is a digit
  char key_dig = '2', key_letter = 'D';
  printf ("map['2'] before apply if: %d, map['D'] before apply if: %d\n",
          *(int *) hashmap_at (map, &key_dig), *(int *) hashmap_at (map, &key_letter));
  int apply_if_res = hashmap_apply_if (map, is_digit, double_value);
  printf ("Number of changed values: %d\n", apply_if_res);
  printf ("map['2'] after apply if: %d, map['D'] after apply if: %d\n",
          *(int *) hashmap_at (map, &key_dig), *(int *) hashmap_at (map, &key_letter));

  // Free the pairs.
  for (int k_i = 0; k_i < 8; ++k_i)
    {
      pair_free ((void **) &pairs[k_i]);
    }

  // Free the hash-map.
  hashmap_free (&map);

  return 0;
}
```