שפת C – תרגיל בית 3

הקצאת זיכרון דינמית, מצביעים לפונקציות, מבני נתונים גנריים

תאריך הגשה: יום חמישי, 14.8.2014 עד שעה 23:55
תאריך הגשה מאוחרת (בהפחתה של 10 נקודות): יום שישי, 15.8.2014 עד שעה 14:00

שאלה 1 (10 נקודות)

השלימו את הקוד של הפונקציה generalSwap שלהלן.
הפונקציה מקבלת שני אלמנטים ומחליפה את תוכנם:

```
/**
 * A swap functions that works with everything
 * Need to get also the size of the elements that are being swapped
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef void* Element;

void generalSwap(Element a, Element b, int sizeOfElement);

int main()
{
      double a = 4., b = 5.;
      int c = 44, d = 55;
      generalSwap(&a, &b, sizeof(double));
      generalSwap(&c, &d, sizeof(int));
      printf("a=%f b=%f\n", a, b); /* should print a=5 b=4 */
      printf("c=%d d=%d\n", c, d); /* should print c=55 d=44 */
      return 0;
}

/**
 * The swap function
 * Swap the two given elements of the given size
 */
void generalSwap(Element a, Element b, int sizeOfElement)
{
  /* Your code here */
}
```

הגישו את הפתרון בקובץ generalSwap.c המצורף לתרגיל (כולל פונקצית ה-main שלמעלה).
ניתן להניח כי הקריאות לפונקציה מתבצעות עם קלטים תקינים.
במקרה של שגיאה בפונקציה (הקצאות זיכרון למשל), עליכם להשאיר את המשתנים כפי שהיו קודם ולצאת מהפונקציה באופן מסודר.

רמז: ניתן להשתמש בפונקציות malloc, free ו-memcpy או שניתן להעתיק byte-byte (ניתן להשתמש גם בכל פונקציה סטנדרטית אחרת).

שאלה 2 (90 נקודות)

רקע למשימה

בעבודתכם החדשה בחברת IT מפורסמת קיבלתם אחריות על הקבצים SimpleRangeTree.c ו-SimpleRangeTree.h. קבצים אלו מממשים עץ בינארי לא מאוזן הממוין לפי המשכורות של העובדים (ראו בהמשך). כל קודקוד בעץ מכיל את הנתונים הבאים לכל עובד (Worker): תעודת זהות, שם העובד, משכורת העובד ודירוג מקום העבודה שלו. דירוג מקום העבודה הינו מספר **ממשי** בין 1 ל-10 המייצג עד כמה מקום העבודה אטרקטיבי, כאשר ערך גבוה מייצג מקום עבודה אטרקטיבי מאוד, וערכים נמוכים מייצגים מקומות עבודה לא אטרקטיביים.
הקבצים Worker.h ו- Worker.o מממשים את ה-struct והפונקציות הקשורות בעובד, כמובן שאסור לכם לשנות קבצים אלה שכן יש להם שימושים רבים אחרים בחברה בה אתם עובדים.
בחברה שלכם משתמשים ב- SimpleRangeTree כדי לבצע חיפושים אחרי טווחי משכורות של עובדים (מיהם העובדים שמרוויחים בין 1000 ל-2000 שקלים בחודש וכו') – התוכנה SimpleSalary.c מדגימה שימוש כזה בעץ (יש כמובן שימושים נוספים שאינם באחריותכם).
העובד שקדם לכם בתפקיד (שעכשיו הוא המנהל שלכם) עשה עבודה בינונית, כלומר איכות כתיבת הקוד היא לא משהו אך נראה שהקוד אינו מכיל באגים פונקציונאליים ו-"עושה את העבודה".

תיאור האלגוריתם המטפל בשאילתת טווח (הפונקציה printRange): הפונקציה מבצעת חיפוש בינארי רגיל (O(log(n))) למציאת הקודקוד הראשון בטווח, ולאחר מכן מבצעת את פעולת ה-Successor (מעבר לקודקוד הבא בגודלו בעץ) עד לכיסוי כל האיברים בטווח. סה"כ סיבוכיות הפעולה: O(log(n)+k) כאשר k מספר הקודקודים בטווח. אצלנו מדובר בסיבוכיות ממוצעת בלבד שכן אנחנו לא טורחים לאזן את העץ.

עם גדילת מספר מנהלי הצוותים העובדים תחת החברה, הנהלת החברה מעוניינת לאפשר לבצע חיפוש לפי טווחים גם עבור משכורות של מנהלים. המנהל שלכם מבקש מכם להכין עץ חיפוש דומה עבור מסד הנתונים של המנהלים. המידע ופונקציות הטיפול במנהלים מופיע בקבצים Manager.h ו- Manager.o המסופקים לכם עם התרגיל, גם קבצים אלו בשימוש נרחב בחברה ואסור לכם לשנותם[1].

בתגובה לבקשת המנהל אתם מציעים להפוך את עץ החיפוש לעץ גנרי, כך שבעתיד במקום להכין עץ חדש לכל סוג נתונים שנרצה לחפש אחריו, ניתן יהיה פשוט לשנות במעט את התוכנית שמשתמשת בעץ.
המנהל כמובן מתלהב מהרעיון. הוא נותן לכם את הקובץ GenRangeTree.h שמכיל את ה-interface שהוא רוצה שתממשו, ואת הקובץ GenSalary.c שמדגים שימוש ב-Range Tree החדש.
הקובץ GenSalary.out מייצג את הפלט הרצוי של GenSalary.c כאשר הוא מורץ עם העץ הגנרי שלכם.
מנהל המשרד מבקש מכם ליצור עותק חדש של SimpleRangeTree.c בשם GenRangeTree.c, שכן העותק הקודם נמצא בשימושים שונים ברחבי המשרד ושינויו יפר תלויות נסתרות רבות.
כלומר עליכם לבצע:

```
> cp SimpleRangeTree.c GenRangeTree.c
```

ולהכניס שינויים בקובץ GenRangeTree.c כך שהקוד הגנרי יעבוד.
שימו לב שעליכם להתבסס על הקוד ב-SimpleRangeTree.c שהמנהל סיפק לכם ולהפוך את הקוד להיות גנרי. עם זאת, מותר לכם לשנות את הקוד כראות עיניכם <u>כל עוד אתם לא משנים את ה-Interface הפומבי הכלול בקוד</u>.

---

[1] במציאות סביר להניח כי קבצי הטיפול במנהלים כלל לא יהיו דומים לקבצים המטפלים בעובדים. לצורך התרגיל, כדי לשמור על דברים פשוטים, שכפלנו את הקוד שבקבצי הטיפול בעובדים ורק הוספנו שדה נוסף למסד המנהל המתאר את מספר הפרויקטים המנוהלים בצוות אותו הוא מנהל.

הדרכה

אנו מממשים מבנה נתונים גנרי אשר יכול להחזיק נתונים מכל טיפוס שהוא (בתנאי שכל הנתונים הם מאותו הטיפוס). לצורך כך ה-constructor מקבל מספר מצביעים לפונקציות בעזרתם המשתמש של מבנה הנתונים מורה למבנה כיצד לעשות השוואות בין איברים, כיצד להדפיס אותם וכו'. אנו נרצה שהמבנה ישמור את הערך של המצביעים לפונקציות הנ"ל במשתנים פנימיים כלשהם כדי שנוכל להשתמש בהם בשלב מאוחר יותר. למשל, נרצה לשמור על המצביע לפונקצית הפיכת אובייקט מהטיפוס הגנרי לטקסט כדי שבעתיד אם נרצה להדפיס את הטיפוסים הגנריים נדע לעשות זאת.
להלן דוגמא למימוש אפשרי של struct המחלקה הגנרית, אתם רשאים גם לבחור מימוש אחר:

```
struct GenRangeTreeRec
{
   NodeP _root;
   int (*lmCmp)(Const Element, Const Element)
   Element (*lmCpy)(Const Element);
   Char*(*lmLbl)(Const Element);
   void (*lmFre)(Element);
   ...
};
```

השדה root שומר את שורש העץ (כמו בעץ המקורי).
השדות lmCmp, lmLbl וכו' שומרים את המצביעים לפונקציות הרלוונטיות (השוואה בין אלמנטים, הפיכת אלמנט למחרוזת לצורך הדפסתו וכו').
הגדרת struct זו תיכתב בקובץ ה-c. זאת כיוון שאין למשתמש בספרייה שלנו צורך לגשת למבנה ה-struct, אלא מספיקה לו גישה לקובץ ה-h הנותן את ה-interface.

דוגמא לשימוש במבנה הנתונים לצורך הדפסת רשומה:

```
void printElement(ConstGenRangeTreeP theTree, ...)
{
    ...
    char *s = theTree->lmLbl(e); //e is a Const Element for printing
    ...
    printf("%s\n", s);
    free(s);                /* Remember that Lbl allocates memory
                               for the string – it's your
                               responsibility to free it. */
    ...
}
```

ספרייה סטטית

בתרגיל זה, עליכם ליצור מהקובץ GenRangeTree ספרייה סטטית. שם הקובץ של הספרייה צריך להיות libGenRangeTree.a.
כאשר אתם באים להשתמש בספריה זו (לדוגמא בדרייבר GenSalary.c), עליכם לעשות linkage לספריה הסטטית שיצרתם.

unit-testing

בכתיבת תוכנה נהוג לכתוב unit-testing. אלו יחידות קוד קטנות, המבצעות בדיקות תקינות של הפונקציות השונות, בדרך כלל במבנה מורכב יחסית. למשל במקרה שלנו אנו נרצה לוודא שפונקציה מסוימת בעץ עובדת ונרצה לבצע בדיקה זו בנפרד ככל האפשר משאר פעולת העץ. בדיקות אלו לא נועדו לבדוק את פלט התכנית, אלא בדיקות ברמה הרבה יותר בסיסית ומפורטת. כל בדיקה שכזו, מריצה את אחת הפונקציות עם פרמטרים שונים, ומשווה את ערך ההחזרה של השיטה עם ערך ההחזרה הצפוי. במקרה של שגיאה יש להדפיס פלט מתאים הכולל את שם השיטה שנכשלה, מה הפרמטרים, מה הערך הצפוי ומה היה ערך ההחזרה.

בתרגיל זה אתם מתבקשים לכתוב unit-testing לקובץ GenRangeTree.c. הבדיקות הינן חלק מתהליך המימוש, ולכן עליהן להיות בתוך הקובץ GenRangeTree.c. מכיוון שאנו לא מעוניינים לספק אותן ללקוח, על הבדיקות להיות עטופות ב `#ifndef NDEBUG`. כך, כאשר הקוד יקומפל עם דגל זה אז הבדיקות לא תהיינה נגישות. על ה-unit testing לכלול פונקציית main ובה מתבצעות בדיקות תקינות של השיטות השונות ב GenRangeTree.
הפקודה make tests תקמפל קובץ ריצה **ותריץ** את הבדיקות שכתבתם. פורמט הבדיקות והפלט הינו פורמט חופשי, אך הרצת make tests צריכה לתת פלט שמסביר בצורה טובה מה נבדק והאם זה עבר את הטסטים או לא.

הנכם מתבקשים לכתוב unit-testing לשתי שיטות לפחות (מומלץ לבדוק כמובן את כל הפונקציות).

הנחיות ורמזים נוספים

- מיקומי הקבצים:

~slabc/public/ex3/
www.cs.huji.ac.il/~slabc/ex3/ex3_files.tar

- עם התרגיל מפורסמת דוגמא של מימוש מחסנית גנרית. הדוגמא יכולה לסייע בהבנה של השימוש במצביעים לפונקציות במבנה נתונים גנריים.
- שימו לב כי Element הוא typedef ל-* void (כלומר מצביע כללי). היררכית המצביעים ב-C יכולה להיות דבר מבלבל עבור מתכנתים לא מנוסים. לפני שאתם מתחילים לכתוב פונקציה מסוימת ודאו שאתם סגורים על מה בדיוק כל משתנה מצביע (ישירות על המידע? מצביע עקיף אל המידע?). עיקר הקושי בתרגיל זה הוא הבנת הקונספט של מצביעים לפונקציות, וכיצד ניתן להשתמש בהם לצורך מימוש מבנה נתונים גנרי ב-C. מבחינת קוד שצריך להקליד, אין הרבה עבודה בחלק זה (כ-50 שורות קוד) – עיקר העבודה הוא בהבנת הקוד שקיבלתם ובהבנת השימוש במצביעים לפונקציות לקבלת קוד גנרי.
- הבהרה נוספת: הקבצים SimpleRangeTree.c ו-SimpleRangeTree.h אינם נדרשים עבור הקומפילציה של GenRangeTree.
- נתונים לכם קבצי האובייקט Manager.o, Worker.o לשם linkage (למשל ב-Makefile).
- מי שאיננו עובד בסביבת linux כמו ב-CS, יכול להשתמש בקבצי המקור הנתונים לכם Manager.c, Worker.c ולקמפל אותם. עם זאת עליכם לוודא שפקודות יצירת הבינאריים שלכם משתמשות בקבצי האובייקט לאחר קימפול (הווה אומר, לעשות רק linkage) ולא לעשות קימפול מקבצי המקור בכל פעם.
- כמו במבנה הנתונים המקורי, במקרה של שגיאה (כישלון בהקצאת זיכרון, פרמטר לא חוקי לפונקציה, וכו') יש להדפיס הודעת שגיאה מתאימה ל-standard error ולצאת. אין צורך לשחרר זיכרון לפני היציאה או לבצע פעולות דומות (כיוון שגם בד"כ הזכרון משוחרר ביציאה וטיפול במצב חירום אינו עיקר התרגיל, אנו מעוניינים לשמור על דברים פשוטים ככל הניתן).
- למען הפשטות אנו נניח כי אין שני עובדים עם אותה משכורת או עם אותו ערך לדירוג של מקום העבודה.
- כדי לשמור על הדברים פשוטים ככל הניתן, טווח החיפוש מועבר לפונקצית החיפוש ע"י מבנים של עובדים / מנהלים ולא ע"י מספרים ממשיים (ראה GenSalary.c). כלומר, אנו מעבירים לפונקציה PrintRange שני עובדים / מנהלים p1,p2 כך שהמשכורת של p1 נמוכה / שווה למשכורת של p2. הפונקציה תדפיס את כל העובדים / מנהלים שמשכורת גבוהה/שווה ל-p1, וקטנה/שווה ל-p2. שימו לב כי החיפושים תמיד כוללים את הקצוות (אי שוויון חלש).
- לפונקציות רבות ב-Interface חלקי קוד משותפים - הימנעו משכפול קוד.
- שימו לב שאתם משחררים את **כל** הזיכרון שהקצאתם (חוץ ממקרים של שגיאות ויציאה מוקדמת מהתוכנית כפי שתואר לעיל).
- עליכם לבדוק שהקצאות זיכרון אכן מצליחות ולדאוג לשחרור כל הזיכרון בסוף התוכנית, אלא אם התוכנית מסתיימת בעקבות שגיאה כפי שתואר לעיל.
- שימו לב כי לא מובטח לכם שכל ניסיונות להקצאת זיכרון יצליחו, עליכם לבדוק זאת.

- חובה עליכם להשתמש בפונקציה assert ולהכניס הוראות debugging לתוכנית. אמנם הוראות רבות כאלו כבר קיימות בקוד שניתן לכם, אך במידה ואתם נדרשים לכך במהלך שינוי הקוד שאתם מבצעים, דאגו לעשות זאת.

תזכורת: השימוש ב-assert נועד לבדוק את הפונקציות הפנימיות שלכם. במקרה של שגיאה של המשתמש (העברת פרמטרים שגויה לפי ה-user interface) – צריך לדווח ל-user על השגיאה ולא להשתמש ב-assert. במקרה של העברת פרמטרים שגויה לאחת הפונקציות הפנימיות שלכם – ה-assert צריך לקפוץ, כנראה שיש לכם באג. למשל, פונקציה פנימית המקבלות מצביעים כאחד הפרמטרים שלה – אפשר לוודא (בעזרת assert) כי הפרמטר אינו שווה ל-NULL. פונקציה פנימית המקבלת אינדקסים של תא – אפשר לוודא שהאינדקסים אינם שליליים. גישה למערך שגודלו ידוע ע"י משתנה – אפשר לוודא שהמשתנה חיובי ושאין חריגה מגבולות המערך.

- פונקציות פנימיות (שאינן מופיעות ב-interface) צריכות להיות מוגדרות כ- static function.
- הרצת תכנית מתוך ה-Makefile דורשת גם את הנתיב (path). אם קובץ ההרצה נמצא בתיקייה הנוכחית, אז יש להוסיף './' לפני שם הקובץ. לדוגמא להרצת הקובץ test יש לכתוב את השורה הבאה:

```
./test
```

עבודה עם valgring

ניהול זיכרון ב-C הוא נושא רגיש ומועד לפורענות – יש הרבה אפשרויות לטעות (לא להקצות מספיק זיכרון, לשכוח לשחרר זיכרון, להשתמש במצביעים שמצביעים לזבל וכו'). כמובן שהקומפיילר לא ידווח על שגיאה בכל המקרים הללו. יתכן שתגלו את השגיאות הללו בזמן ריצה, אך יתכן גם כי התוכנה תעבוד אצלכם "במקרה" והבעיות יתגלו דווקא בביתו של הלקוח.

ישנו מבחר די גדול של תוכנות בשוק שמטרתם לסייע באיתור בעיות זיכרון בקוד לפני שחרורו אל הלקוח. אנו נשתמש בתוכנת valgrind, שיחסית לתוכנה חינמית, נותנת תוצאות מעולות. בתרגיל זה אנו דורשים מכם להריץ את valgrind עם התוכנה שלכם. את הפלט שלה יש להגיש בקובץ בשם valdbg.out.

כדי להריץ את valgrind עליכם לבצע קומפילציה ו-linkage לקוד שלכם עם הדגל g- (הן בשורת הקומפילציה והן בשורת ה-linkage). לאחר מכן הריצו valgrind:

```
> valgrind GenRangeTree
```

אם קיבלתם הודעת שגיאה, יתכן שתצטרכו לבצע שינוי הרשאות:

```
> chmod 777 GenRangeTree
```

כמובן שאם valgrind דיווח על בעיות עם הקוד שלכם, עליכם לתקן אותן.

הנחיות הגשה

עליכם להגיש:

- הקובץ generalSwap.c המכיל את פתרון שאלה 1.
- את הקבצים GenRangeTree.h, GenRangeTree.c המכילים את המימוש שלכם.
- את קובץ הפלט של valgrind: valdbg.out
- קובץ makefile התומך בפקודות הבאות:

1. יצירת קובץ הרצה עבור שאלת generic swap בעזרת הפקודה

make generalSwap

2. יצירת הספריה הסטטית libGenRangeTree.a בעזרת הפקודה

make GenRangeTree

3. הרצת ה unit tests שכתבתם בעזרת הפקודה

make tests

4. מחיקת קבצים זמניים, קבצי ריצה, קבצי אובייקט שלכם ואת הספרייה הסטטית (ותזכרו מה קורה כשהם לא קיימים), בעזרת הפקודה

make clean

5. תמכו ב-PHONY. וצרפו אליו את המטרות הרלוונטיות לדעתכם.

- על הקומפילציה להתבצע עם הדגל Wall– ללא errors / warnings.
- קבצו את כל הקבצים הנ"ל לקובץ tar יחיד בשם ex3.tar
- בדקו שהקובץ עובר את ה-presubmission script לפני הגשה:

> ~slabc/public/presubmit_ex3 ex3.tar

מקורות

- לתיעוד של הפונקציות malloc, realloc, free וכו' השתמשו ב-man או חפשו ב-Google (המון מקורות אפשריים). כל ספר לימוד בסיסי מכיל תיאור של פונקציות אלו.
- מצביעים לפונקציות ב-C:
  יש המון חומר באינטרנט על מצביעים לפונקציות ב-C. למשל:
  http://www.newty.de/fpt/index.html.
- נסו ללמוד מהדוגמא שניתנה לכם למחסנית גנרית (Stack).
- ליצירת ספריה סטטית ניתן להסתכל בדוגמא הבאה:
  http://www.adp-gmbh.ch/cpp/gcc/create_lib.html
- הדפסה ל standard error בעזרת הפונקציה `fprintf`:
  http://www.cplusplus.com/reference/cstdio/fprintf/
  http://www.cplusplus.com/reference/cstdio/stderr/