סדנאות תכנות בשפת C ו-(67312) ++C

C – תרגיל 3

תאריך הגשה יום רביעי, 11 לדצמבר 2019, בשעה 23:55

הגשה מאוחרת (בהפחתת 10 נקודות): יום חמישי, 12 לדצמבר 2019, בשעה 23:55

1 הקדמה

בתרגיל זה תתרגלו שימוש ב- struct, מצביעים לפונקציות, הקצאות דינמיות וגנריות בשפת C על ידי מימוש מבנה נתונים גנרי של עץ אדום-שחור.

בתרגיל יפורטו כל ההנחות שניתן להניח לגבי הקלט והבדיקות, **לא ניתן להניח הנחות נוספות**.

1.1 מבני נתונים גנריים

מבנה נתונים הוא ארגון של מידע בצורה שמאפשרת גישה ושינוי יעילים. ישנם מבני נתונים רבים (עץ חיפוש בינארי, טבלאות גיבוב ועוד) אשר שונים במימוש וביעילות של פעולות שונות. חשוב להפריד בין הרעיון האבסטרקטי של מבנה נתונים למימוש שלו. לדוגמה, Set הוא מבנה נתונים שלא מכיל כפילויות, אך ניתן לממש אותו בדרכים שונות – עץ, רשימה מקושרת וכדו'.

תכונה אחת חשובה למימוש של מבני נתונים היא **גנריות** – היכולת לייצר מבני נתונים שונים המחזיקים סוג שונה של מידע (int, struct, char*...) ללא צורך לכתוב מחדש את המימוש של מבנה הנתונים עבור כל סוג מידע. כך למשל, מימוש של עץ חיפוש בינארי גנרי יאפשר ליצור מופע (instance) של עץ חיפוש בינארי שמחזיק int וגם לייצר מופע אחר של עץ חיפוש בינארי שמחזיק char* - כל זה ע"י שימוש באותו קוד מקור.

אחת הדרכים לייצר גנריות בשפת C היא להשתמש במצביעים ל- void*, אותם ניתן להמיר (cast) לכל סוג אחר של מצביע. למשל:

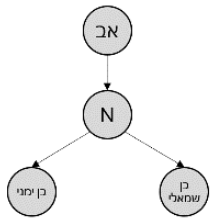
```
typedef struct Example
{
    int id;
    char *value;
} Example;

void genericFuncExample(void *p)
{
    Example *e = (Example *) p;
    printf("ID: %d, Value:%s\n", e->id, e->value);
}
```

בדוגמה ניתן לראות כיצד פונקציה יכולה לקבל void*, להמיר אותו לסוג המצביע "האמיתי" של המידע ולאחר מכן להשתמש בו כמו בכל מצביע לסוג נתונים זה. שימו לב – הפונקציה עצמה אינה גנרית! היא ספציפית לסוג מסוים של מידע שמועבר אליה כפוינטר. הגנריות מתבטאת בכך שהיא מקבלת void*, כלומר – פוינטר לכל טיפוס נתונים שהוא. הפונקציה צריכה להמיר את ה-void* למצביע לסוג המידע הספציפי עליו היא מתוכננת לפעול. כיצד משתמשים בפונקציה כזו (בעלת חתימה גנרית) לטובת מימוש מבנה נתונים גנרי יובהר בהמשך המסמך.

1.2 עץ אדום-שחור

עץ חיפוש בינארי בנוי מקודקודים (Nodes), כאשר לכל קודקוד יש 4 שדות לפחות (ראו איור משמאל):

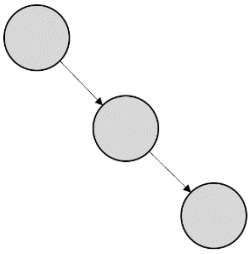


- מצביע לקודקוד "אב".
- מצביע ל"בן ימני".
- מצביע ל"בן שמאלי".
- מידע.

הקודקוד הראשון בעץ חיפוש בינארי (קודקוד שאין לו אב) נקרא "שורש" וקודקוד ללא בנים נקרא "עלה".

עץ חיפוש בינארי מקיים את החוקים הבאים:

- לכל קודקוד יכולים להיות לכל היותר 2 בנים – בן ימני ושמאלי. לעץ שמתחיל בבן הימני/שמאלי קוראים תת-עץ ימני/שמאלי בהתאמה.
- לכל קודקוד, כל הקודקודים בתת העץ השמאלי מכילים מידע שקטן מהמידע שבקודקוד.
- לכל קודקוד, כל הקודקודים בתת העץ הימני מכילים מידע שגדול מהמידע שבקודקוד.



ע"י שמירה על חוקים אלה, ניתן לשמור מידע בצורה ממוינת בעץ כך שהזמן הממוצע לביצוע פעולה (הכנסה, הוצאה, חיפוש) הוא נמוך ($O(log(n))$). למה ממוצע? מכיוון שייתכן מצב בו לכל קודקוד בעץ יש בן יחיד (למשל, ההכנסה של המידע לעץ נעשית בצורה ממוינת), כך שנוצרת מעין רשימה מקושרת – בה פעולות שונות (כגון חיפוש, גישה) יקרות יותר בזמן ($O(n)$).

בכדי לפתור את הבעיה הזו אנו נדרשים לאלץ את העץ להיות "מאוזן" – להימנע ממצב כזה של רשימה מקושרת. עץ אדום-שחור הינו עץ חיפוש בינארי מאוזן שכזה, כלומר – תהליך ההכנסה וההוצאה של קודקודים עלול לגרור שינוי במבנה העץ בכדי להקטין את גובה העץ ולהבטיח זמן ריצה נמוך ($O(log(n))$) לפעולות הכנסה, הוצאה וחיפוש.

אז מה התכונות של עץ אדום-שחור, וכיצד שומרים על איזון העץ?

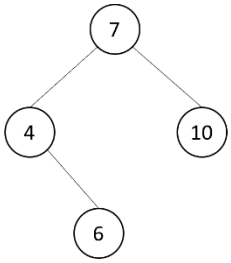
חשוב תחילה לבצע את האבחנה הבאה – בעץ מהסוג שהכרנו עד כה העלים הינם קודקודים ששדות הבנים הימני והשמאלי שלהם מכילים NULL.
בעץ אדום-שחור קודקודים אלו **אינם נחשבים עלים**. העלים בעץ זה הם למעשה הבנים הימני והשמאלי של קודקודים אלה. נשים לב כי קודקודים אלו **אינם נמצאים בעץ** אבל חשובים לנו כיוון שהם נחשבים כבעלי צבע שחור. לכן ,כאשר רוצים לספור את כמות הקודקודים השחורים בעץ, ניקח בחשבון את העלים שלנו.

בעץ אדום-שחור, לכל קודקוד יש גם צבע (שחור או אדום) בנוסף לשדות הרגילים והעץ כולו צריך לקיים את התכונות הבאות:

1. העץ הוא עץ חיפוש בינארי – עבור קודקוד כלשהו v בעץ יכולים להיות לכל היותר שני בנים (ימני ושמאלי), כך שלכל קודקוד u שנמצא בתת העץ השמאלי מתקיים $u.data < v.data$ ולכל קודקוד w בתת העץ הימני מתקיים $v.data < w.data$.
2. לכל קודקוד יש צבע בנוסף לשדות הרגילים (מצביעים לאב, בן ימני, בן שמאלי ומידע). קודקוד יכול להיות אדום או שחור (רק אחד מהשניים).
3. שורש העץ תמיד שחור.
4. כל ההפניות לקודקודים לא קיימים (NULL) נחשבות קודקודים שחורים.
5. כל הילדים של קודקוד אדום הם שחורים.
6. בכל מסלול פשוט מקודקוד v בעץ לכל אחד מהצאצאים העלים שלו (הפניות ל-NULL) יש אותו מספר של קודקודים שחורים

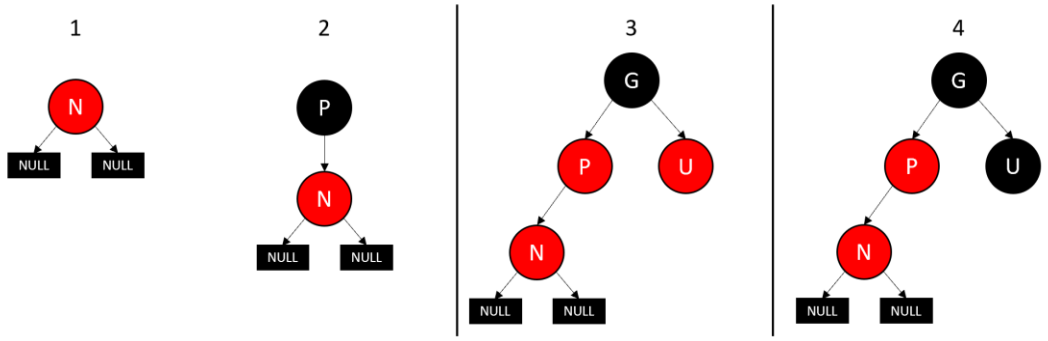
הערה *תכונות אלה מבטיחות חיפוש בזמן לוגריתמי, מאחר שהמסלול הארוך ביותר מהשורש לעלה כלשהו הוא לכל היותר פי שניים מהמסלול הקצר ביותר לעלה כלשהו (אותו מספר של קודקודים שחורים בשני המסלולים, לכל היותר יש קודקוד אדום נוסף עבור כל קודקוד שחור במסלול הארוך)*

פעולת ההכנסה בעץ אדום-שחור (איזון) –
עבור פעולת ההכנסה, נתייחס תחילה לעץ כאל עץ בינארי רגיל.
נתחיל בשורש ונלך לבן הימני/שמאלי בהתאמה לפי הערך שמופיע בקודקודים עד שנגיע לNULL, שם נשים את הקודקוד החדש. לדוגמה, הכנסת 5 לעץ שמופיע מתחת תעבוד כך – נשווה את 5 לערך שבקודקוד ונראה שהוא קטן, לכן נמשיך לבן השמאלי. כעת נשווה ל-4 ונראה שהוא גדול, לכן נלך לבן הימני. כעת משווים מול 6 ורואים שהוא קטן, אז הולכים לבן השמאלי. אבל, הבן השמאלי הוא NULL, לכן נחליף את ה-NULL בקודקוד חדש שמכיל את הערך 5.

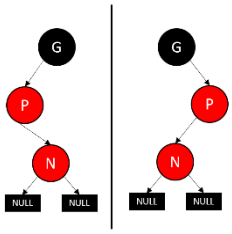


אבל, לא סיימנו את העבודה בעץ אדום-שחור לאחר הכנסה שכזו – ייתכן שעלינו לשנות את העץ בכדי לשמר את התכונות של עץ אדום-שחור (ניתן לקרוא על ההפרות השונות ומדוע התהליכים שנפרט משמרים את התכונות המופרות בקישור הבא).

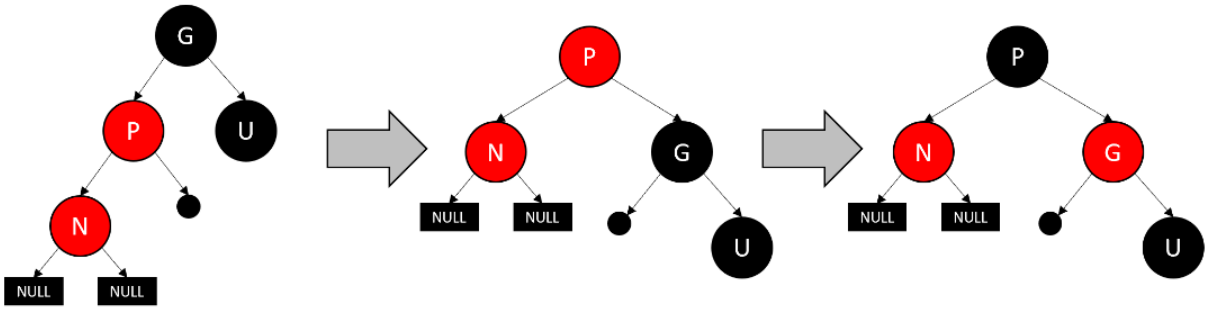
קודקוד חדש שנכניס יתחיל תמיד בצבע אדום. לאחר ההכנסה של הקודקוד (תזכורת – הקודקוד מחליף עלה (NULL) והבנים שלו הם עלים), יש 4 מצבים אפשריים למצב העץ לאחר הכנסת הקודקוד החדש (נסמנו N):



1. **הקודקוד שהכנסנו הוא השורש** – נדרש להחליף את הצבע של N מאדום לשחור.
2. **קודקוד האב (P) של N שחור** – אין צורך לתקן את העץ.
3. **קודקוד האב (P) של N אדום וגם הדוד (U) שלו אדום:**
   א. נהפוך את הצבע של P ו-U לשחור.
   ב. נהפוך את הצבע של הסב של N (האב של P, נסמנו G) לאדום.
   ג. נריץ את אלגוריתם התיקון על הסב של N (קודקוד האב של P)
4. **קודקוד האב (P) של N אדום והדוד (U) שלו שחור:**



   א. נבדוק האם N הוא בן ימני של בן שמאלי או בן שמאלי של בן ימני. אם לא – מדלגים על שלב זה, אחרת נבצע רוטציה עם N ו-P כך שתיווצר "שרשרת". למשל, אם N בן ימני אז P יהפוך להיות הבן השמאלי של N, והאב של N יהפוך להיות G
   ב. עבור בן ימני של בן שמאלי (המקרה ההפוך זהה, מלבד הפיכת כיוון הרוטציות) נעשה סיבוב ימני ל- G – הבן השמאלי של G הופך לבן הימני של P והבן הימני של P הופך ל-G
   ג. הופכים את הצבע של P לשחור ושל G לאדום.



2 משימה 1 – מימוש העץ

כפי שהזכרנו בהקדמה, בתרגיל תממשו עץ אדום-שחור גנרי. שלד של Node ו- RBTree מסופקים לכם בקובץ RBTree.h, יחד עם חתימות הפונקציות שעליכם לממש ומספר typedef.

בתרגיל זה, הגנריות של העץ ממומשת בכך שהמידע שהקודקודים בו מחזיקים הוא מסוג void*. נשים לב כי כל מופע של העץ יחזיק סוג יחיד של מידע ועליו לדעת להשוות בין פריטי מידע שונים ולשחרר את משאבי העץ. לכן, בעת יצירת מופע של עץ יש להעביר מצביעים לפונקציות שישמשו להשוואה בין איברים ולשחרור המשאבים.

למשל, בשביל עץ אדום-שחור שנועד להחזיק char* בהקצאה דינמית (כלומר, פריטי המידע נשמרים ב-heap ונדרש לשחרר את הזיכרון שלהם במחיקה של העץ), יש להעביר את הפונקציות הבאות:

- פונקציית השוואה שממירה שני מצביעים מסוג void* למצביעים מסוג char* ומחזירה את התוצאה של strcmp עליהם
- פונקציית שחרור שמקבלת void* ומשחררת אותו. בנוסף, בתרגיל זה לעץ יש שדה בשם size ששומר את מספר הקודקודים שיש בעץ.

להלן הסבר ל- typedef השונים לפונקציות של העץ:

1. `typedef int (*CompareFunc)(const void *a, const void *b)`

פונקציה בעלת חתימה כזו יכולה לשמש כפונקציית ההשוואה בין איברים בעץ, שתשמש בכדי לקבוע את מיקום ההכנסה המתאים להם בעץ. פונקציה בעלת חתימה כזו מקבלת שני מצביעים ל- void*, אלה האובייקטים ביניהם היא משווה. פונקציה שתממש את החתימה הזו (כלומר – תחזיר int ותקבל שני מצביעים קבועים ל- void*), תמיר את המצביעים לסוג המתאים ותבצע השוואה ביניהם.

2. `typedef int (*forEachFunc)(const void *object, void *args)`

פונקציה בעלת חתימה כזו יכולה לשמש כפונקציה שמפעילים על כל איברי העץ. שימו לב – הפונקציה אינה משנה את המידע שבעץ! פונקציה בעלת חתימה כזו מקבלת שני מצביעים ל- void*, אחד הוא מידע מקודקוד בעץ (object) והשני תלוי במטרת הפונקציה. ראו דוגמה לשימוש בפונקציה כזו בסעיף הזה.

```
int sumExample(const void *object, void *args)
{
    int *sum = (int *) args;
    int *obj = (int *) object;
    *sum += *obj;
    return 1;
}
```

עבור העץ, יהיה עליכם לממש את הפונקציות הבאות (שימו לב, אינכם צריכים לממש פעולת מחיקה):

1. **יצירת עץ חדש** – `RBTree *newRBTree(CompareFunc compFunc, FreeFunc freeFunc)`
פונקציה זו מקבלת שני מצביעים לפונקציות (פונקציית השוואה ושחרור משאבים) ומחזירה עץ אדום-שחור חדש ריק עם פונקציות ההשוואה והשחרור הנתונות. במידה ולא ניתן היה לייצר את העץ, יש להחזיר NULL.
2. **הכנסת איבר לעץ** – `int addToRBTree(RBTree *tree, void *data)`
פונקציה זו מקבלת מצביע לעץ ולמידע, מכניסה את המידע לעץ במקום המתאים ומבצעת תיקון לעץ במידה ונדרש. מחזירה 0 במקרה של כישלון (אם המידע נמצא כבר בעץ זה נחשב כישלון) וכל מספר אחר במקרה של הצלחה.
3. **חיפוש בעץ** – `int containsRBTree(RBTree *tree, void *data)`
פונקציה זו מקבלת מצביע לעץ ולמידע, מחזירה 0 אם המידע שמחפשים לא נמצא בעץ וכל מספר אחר אם הוא נמצא בעץ. את ההשוואות מבצעים בעזרת פונקציית ההשוואה שנמצאת בעץ.
4. **הפעלת פונקציה על כל איברי העץ (in-order)** –
`int forEachRBTree(RBTree *tree, forEachFunc func, void *args)`
לפעמים, נרצה לבצע פעולה מסוימת על כל איברי העץ (למשל, להדפיס אותם או לסכום אותם).
פונקציה זו מקבלת מצביע לעץ, לפונקציה ולמשתנה נוסף ומפעילה את הפונקציה על כל מידע שקיים בעץ, ב- in-order traversal (כלומר, בסדר עולה של האיברים). את המשתנה הנוסף מעבירים לפונקציה כשקוראים לה, המטרה היא לספק ארגומנטים נוספים לפונקציה או לאפשר מעבר של מידע בין ההפעלות השונות של הפונקציה, למשל – בעץ שמחזיק int, ניתן לממש פונקציה שסוכמת את איברי העץ ע"י קריאה ל- forEachRBTree עם מצביע ל-int עבור args ומצביע לפונקציה הבאה:

```
int sumTree(const void *object, void *args)
{
    if (object == NULL || args == NULL)
    {
        return 0;
    }
    int *sum = (int *) args;
    int const *data = (int *) object;
    *sum += *data;
    return 1;
}
```

בצורה הזאת, forEachRBTree מפעילה את sumTree פעם אחת עבור כל פריט שנמצא בעץ, כאשר כפרמטרים היא מקבלת את הפריט הנוכחי ואת המצביע ל-int (את args). חשוב לשים לב – ע"י העברת מצביע ל-struct ב-args, ניתן להעביר יותר מפריט מידע יחיד (רק int או char*).
5. **שחרור המשאבים של העץ** – `void freeRBTree(RBTree *tree)`
פונקציה זו מקבלת מצביע לעץ ומשחררת את המשאבים שלו.

דוגמה למבנה אותו ניתן לשמור בעץ ופונקציית השוואה אחת המתאימה לו יחד עם בדיקות ה-presubmit מסופקים לכם בקובץ ProductExample.h (ללא coding style ובדיקות קומפילציה – בדיקות אלה עליכם לבצע בעצמכם).

3 משימה 2 – שימוש בספרייה

בחלק זה עליכם לממש מספר פונקציות שמשתמשות בספרייה (העץ) שכתבתם. עליכם לממש פונקציות שיאפשרו שימוש בשני סוגי מידע – מחרוזות ו-וקטורים. בקובץ structs.h מופיע מבנה שמייצג Vector עבור משתנים מסוג double (אורך לא קבוע). עליכם לממש את הפונקציות הבאות עבור וקטורים:

1. `int vectorCompare1By1(const void *a, const void *b)`

פונקציית השוואה איבר איבר (Element by Element), כלומר משווים בין האיבר הראשון של וקטור א׳ לאיבר השני של וקטור ב׳, אם הם שווים משווים את האיבר השני של וקטור א׳ לאיבר השני של וקטור ב׳ וכן הלאה – האיבר הראשון ששונה קובע את תוצאת ההשוואה. אם הוקטורים מכילים את אותם האיברים עד סוף אחד הוקטורים, הוקטור הקצר יותר ייחשב קטן יותר. אם הם זהים לחלוטין, הם ייחשבו שווים. להלן מספר דוגמאות ליחס בין וקטורים:

$$[1,2,3] < [1,5], [1,2,3] < [1,2,3,1], [1,2] = [1,2]$$

2. `int copyIfNormIsLarger(const void *vector, void *maxVector)`

פונקציה שמפעילים על וקטור – מקבלת 2 פוינטרים לוקטורים, מעתיקה את התוכן של הוקטור הראשון לוקטור השני במידה והנורמה (l_2) של הוקטור הראשון גדולה יותר מהנורמה של הוקטור השני. תזכורת, עבור וקטור $v = [a_1 \quad \dots \quad a_n]$ הנורמה היא

$$\|v\| = \sqrt{a_1^2 + \cdots + a_n^2}$$

יש להשתמש בפונקציה זו במימוש של הפונקציה הבאה:

3. `Vector *findMaxNormVectorInTree(RBTree *tree)`

פונקציה שמחפשת ומחזירה מצביע לוקטור חדש שהינו עותק של הוקטור עם הנורמה הגדולה ביותר הנמצא בעץ.

4. `void freeVector(void *vector)`

פונקציית שחרור משאבים עבור וקטור.

בנוסף אתם נדרשים לממש פונקציית השוואה ל- char* ופונקציית שחרור ל - char*, אשר חתימתן מופיעה בקובץ Structs.h.

4 דגשים והנחות מקילות

- שימו לב – עליכם לממש את כל הפונקציות שחתימתן מופיעה בקבצי ה-h שסופקו לכם. גם אם אין הסבר מפורט לחתימה של פונקציה כלשהי במסמך זה, יש לממש אותה ע"פ התיעוד שמופיע בקובץ.
- שימו לב ל**ניהול הזיכרון** – מתי צריך לבצע הקצאה דינאמית ומתי לא ולדאוג לשחרור כל המשאבים במחיקת העץ.
- בדקו את התרגיל עם valgrind על מחשבי בית הספר, דליפות זיכרון יגררו הורדה משמעותית של נקודות. בכדי להריץ valgrind על תוכנה בשם c_ex3 שלא מקבלת פרמטרים מהטרמינל יש לכתוב את השורה הבאה:

```
valgrind c_ex3
```

בכדי לקבל מידע מלא על דליפות הזיכרון (במידה ויש) יש להריץ את השורה הבאה:

```
valgrind --leak-check=full
```

- ניתן להניח כי לפונקציית השוואה שנמצאת בעץ יוזנו רק מצביעי void* שמצביעים לסוג המשתנה שהפונקציה מצפה לקבל.
- לא ניתן להניח שמצביעים שעוברים לפונקציות אינם NULL.
- ניתן ואף רצוי לממש פונקציות נוספות שאינן בקובץ RBTree.h לטובת מימוש הפונקציות שמופיעות בקובץ.
- שימו לב שהתרגיל מתקמפל **ללא שגיאות/אזהרות!** שגיאות/אזהרות בקומפילציה (עם הדגלים -Wvla -Wall -Wextra -std=c99) יגררו הורדת נקודות.
- התרגילים ייבדקו על מחשבי בית הספר לכן בדקו שהתרגיל מתקמפל ורץ על מחשבי בית הספר – על אחריותכם לבדוק זאת!
- שימו לב – מאחר ובתרגיל מימשנו מבנה נתונים, נייצר ממנו ספרייה סטטית ע"י קמפול לקובץ a. לאחר שמייצרים את הספרייה, מבצעים linkage יחד איתה עבור קבצים שמשתמשים בה.
  בצורה הזאת ניתן גם להריץ קוד שלכם עם פתרון בית הספר – בצעו linkage עם הספרייה והקבצים שאמורים להשתמש בה.

5 פקודות טרמינל

- כלל הפקודות שמופיעות בחלק זה מניחות כי הן רצות מחלון טרמינל שנפתח מתיקיה שמכילה את המימושים שלכם וקובץ ה-Makefile שסופק לכם.
- מימוש בית הספר לשתי הספריות זמין בנתיב

```
~labcc/www/c_ex3/school_solution/
```

- הרצת בדיקה ל-coding style:

```
~labcc/www/codingStyleCheck <code file or directory>
```

- הרצת presubmission על המימוש שלכם:

```
make presubmit
```

- הרצת presubmission על פתרון בית הספר:
  תחילה יש צורך להעתיק את פתרון בית הספר מהתיקיה בה הוא נמצא

```
~labcc/www/c_ex3/school_solution/
```

  לתיקיה שמכילה את המימוש שלכם. לאחר מכן, יש להריץ את הפקודה הבאה:

```
make school_presubmit
```

- כשברצונכם לבדוק את התנהגות פתרון בית הספר על מקרים מסוימים, עליכם לממש את הבדיקות הללו בקובץ בשם test_cases.c ולהריץ את הפקודה הבאה:

```
make school_tests
```

  שימו לב – הפקודה תיכשל במידה ותנסו להריץ את השורה הזאת ללא קובץ בשם test_cases.c בתיקיה
- הפקודה ליצירת tar מופיעה בקובץ ה- Makefile. בכדי להריץ אותה, כתבו את הפקודה הבאה:

```
make tar
```

6 נהלי הגשה

- **קראו בקפידה את הוראות תרגיל זה ואת ההנחיות להגשת תרגילים שבאתר הקורס. כמו כן, זכרו כי התרגילים מוגשים ביחידים. אנו רואים העתקות בחומרה רבה!**
- את השורה שמייצרת ספרייה סטטית (.a) מקובץ o. ניתן לראות ב-Makefile שניתן לכם עם התרגיל.
- אנא וודאו כי התרגיל שלכם עובר את ה-presubmission script ללא שגיאות או אזהרות.
  בדיקות ה- presubmission מסופקות לכם בקובץ ProductExample.c.
- הבדיקות ירוצו על המימוש שלכם של הספרייה הסטטית (אותה נייצר מהקובץ RBTree.c). לכן, אם המימוש שלכם לא עובד בכלל – כלל הבדיקות ייכשלו, גם הבדיקות של המימושים של הקובץ Structs.h.
- אנו ממליצים לקחת את תבנית הבדיקות שניתנה לכם בקובץ ProductExample.c ולהרחיב אותה עם בדיקות משלכם.
- אין להגיש את קבצי ה-h.
- יש להגיש קובץ tar בשם c_ex3.tar המכיל 2 קבצים בלבד:
  - Structs.c - המימוש של Structs.h
  - RBTree.c - המימוש של RBTree.h