סדנאת תכנות בשפת C ו-++C (67315)

תרגיל 3

תאריך הגשה יום רביעי, 3 ליוני 2020, בשעה 23:25

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.

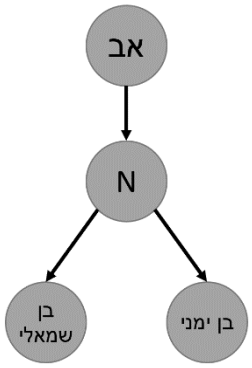
אחת הדרכים לייצר גנריות בשפת C היא להשתמש במצביעים ל- void*, אותם ניתן להמיר (cast) לכל סוג אחר של מצביע. למשל:

```
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



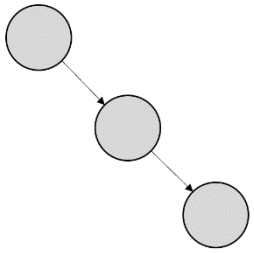
עץ חיפוש בינארי בנוי מקודקודים (Nodes), כאשר לכל קודקוד יש 4 שדות לפחות (ראו איור משמאל. *הערה – ניתן לממש עץ חיפוש בינארי בלי מצביע לאב*):

- 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.

***הערה** – באופן כללי במסמך, האיורים המוצגים אינם בהכרח של עץ אדום-שחור תקין, אלא של תת-עץ של עץ אדום-שחור תקין ומטרתם להמחיש את האלגוריתם.*

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

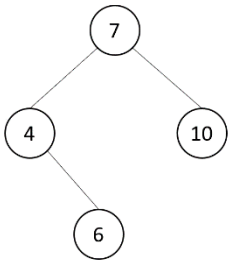
חשוב תחילה לבצע את האבחנה הבאה – בעץ מהסוג שהכרנו עד כה העלים הינם קודקודים ששדות הבנים הימני והשמאלי שלהם מכילים NULL.
בעץ אדום-שחור קודקודים אלו **אינם נחשבים עלים**. העלים בעץ זה הם למעשה הבנים הימני והשמאלי של קודקודים אלה, כלומר – ההפניות NULL. נשים לב כי קודקודים אלו **אינם נמצאים בעץ** אבל חשובים 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

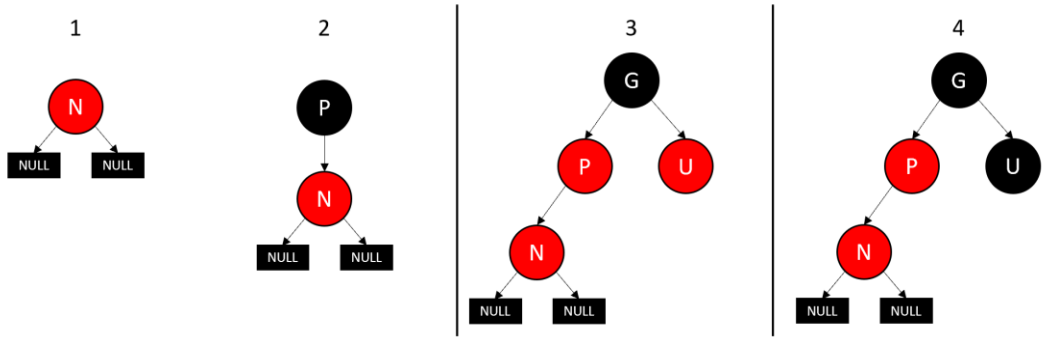
הערה *תכונות אלה מבטיחות חיפוש בזמן לוגריתמי, מאחר שהמסלול הארוך ביותר מהשורש לעלה כלשהו הוא לכל היותר פי שניים מהמסלול הקצר ביותר לעלה כלשהו (אותו מספר של קודקודים שחורים בשני המסלולים, לכל היותר יש קודקוד אדום נוסף עבור כל קודקוד שחור במסלול הארוך)*

פעולת ההכנסה בעץ אדום-שחור (שימור התכונות) –
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.

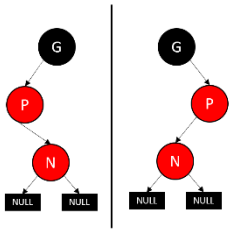


אבל, לא סיימנו את העבודה בעץ אדום-שחור לאחר הכנסה שכזו – ייתכן שעלינו לשנות את העץ בכדי לשמר את התכונות של עץ אדום-שחור (ניתן לקרוא על ההפרות השונות ומדוע התהליכים שנפרט משמרים את התכונות המופרות בקישור הבא).

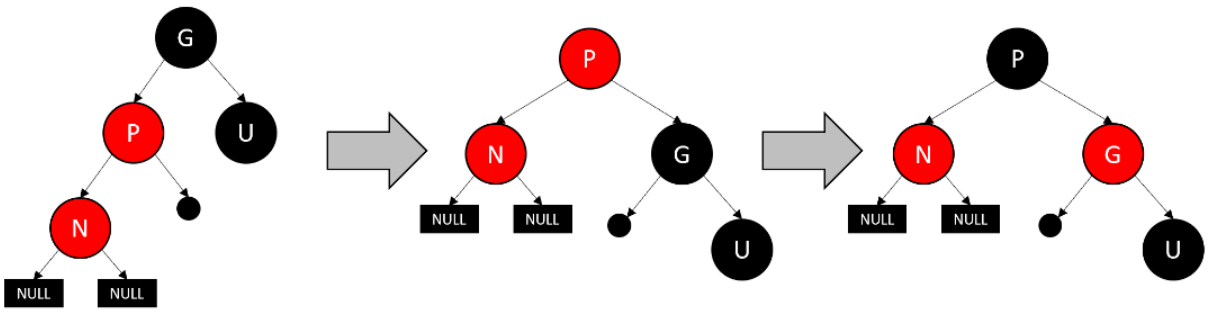
קודקוד חדש שנכניס יתחיל תמיד בצבע אדום. לאחר ההכנסה של הקודקוד (תזכורת – הקודקוד מחליף עלה (NULL) והבנים שלו הם עלים), יש 4 מצבים אפשריים למצב העץ לאחר הכנסת הקודקוד החדש (נסמנו N):



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), כלומר – נתחיל את תהליך התיקון בצורה רקורסיבית כאשר נסמן את G להיות N.
4. **קודקוד האב (P) של N אדום והדוד (U) שלו שחור:**

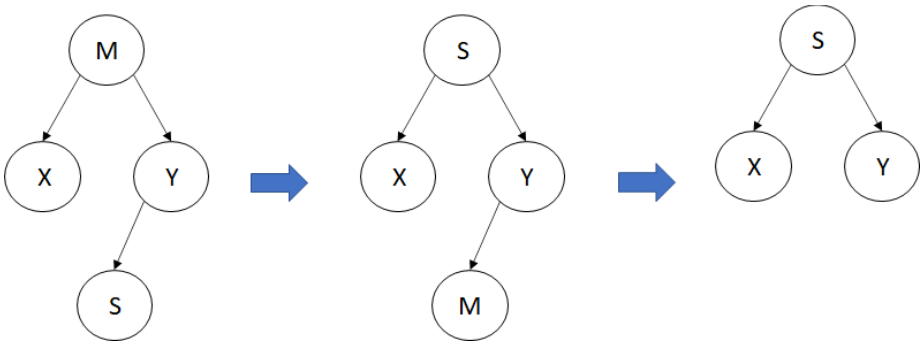


   א. נבדוק האם N הוא בן ימני של בן שמאלי או בן שמאלי של בן ימני. אם כן (כלומר, המצב הוא כפי שמופיע בציור משמאל) נבצע רוטציה על P כך שתיווצר "שרשרת" ואז נמשיך לשלב 4.ב. למשל, אם N בן ימני אז P יהפוך להיות הבן השמאלי של N, ו-G יהפוך להיות האב של N. אם לא – נמשיך ישר לשלב 4.ב בלי לבצע רוטציה.
   ב. עבור בן שמאלי של בן שמאלי (המצב המתואר בציור למטה) נעשה רוטציה ימנית ל- G – הבן הימני של P הופך לבן השמאלי של G ו-G הופך לבן הימני של P. המקרה ההפוך זהה (בן ימני של בן ימני), מלבד הפיכת כיוון הרוטציות.
   ג. הופכים את הצבע של P לשחור ושל G לאדום.



פעולת המחיקה בעץ אדום-שחור (שימור התכונות) –
מחיקת איבר מעץ אדום-שחור בנויה משני שלבים, שכן קיימת יותר ממטרה אחת בפעולת המחיקה. נרצה להסיר את האיבר אך גם לשמור על תכונת האיזון של העץ.

בשלב הראשון, נסתכל על תהליך זה כמחיקת איבר מעץ חיפוש בינארי.

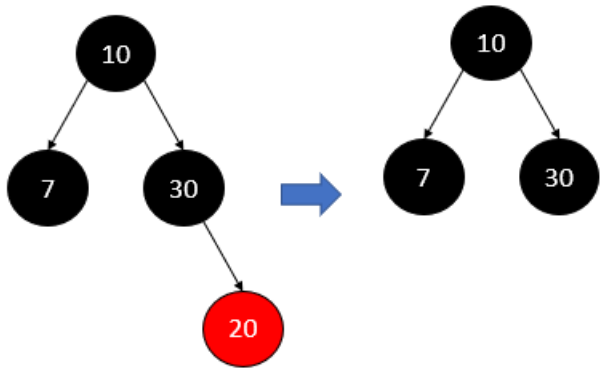


נסמן את הקודקוד שמכיל את המידע שאנחנו רוצים למחוק ב-M. אם ל-M אין ילדים - בעץ בינארי רגיל היינו מוחקים את M וסיימנו. אם ל-M יש ילד אחד - בעץ בינארי רגיל היינו מוחקים את M ומחברים את הילד שלו לאבא שלו במקום M. אם ל-M יש שני ילדים - אנחנו מוצאים את ה"יורש" (successor) של M – הקודקוד שמכיל את המידע בעץ שהכי קטן מבין כל הערכים שגדולים מהערך של M. היורש הוא תמיד הקודקוד השמאלי ביותר בתת העץ הימני של M (בדוגמה לעיל זהו S). לאחר מכן, נחליף בין הערך של M לזה של היורש S (שלב 2 בתרשים) ואז בעץ בינארי רגיל היינו מוחקים את הקודקוד של היורש (שלב 3 בתרשים). נשים לב שבסופו של דבר ליורש יהיה בן אחד או שלא יהיו לו בנים בכלל (וודאו שהבנתם מדוע :)).
כלומר, השלב הראשון במחיקה בעץ אדום-שחור הינו לבצע את ההחלפה בין הערך של M לערך של היורש שלו (במידה ול-M היו שני ילדים). אם ביצענו החלפה נסמן את הקודקוד היורש בתור M. שלב זה מבטיח לנו שבשלב הבא, הקודקוד M אותו אנחנו רוצים למחוק הוא אחד משני דברים - קודקוד "עלה בינארי" (כאשר ב"עלה בינארי" הכוונה עלה של עץ בינארי רגיל - קודקוד עם שתי הפניות NULL, בניגוד לעלה של עץ אדום שחור - שהוא הפניית ה-NULL) או קודקוד בעל בן אחד שהוא עלה) הפניית (NULL ובן אחד שאינו עלה

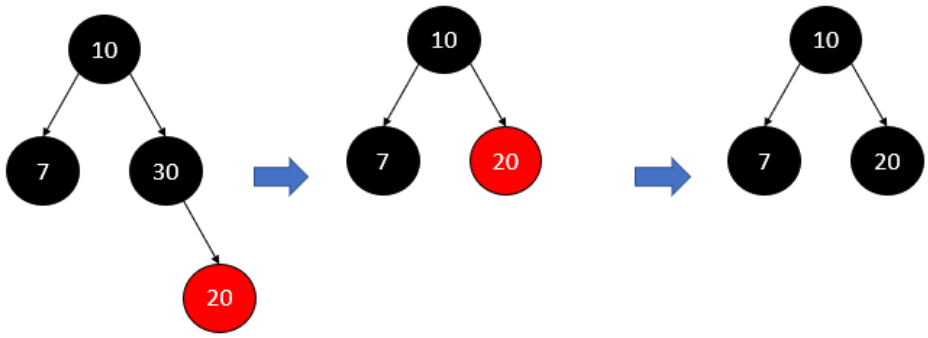
בשלב השני של המחיקה, נרצה לדאוג שתכונות העץ נשמרות כדי שהעץ יישאר מאוזן. תחילה נשים לב שהשלב הראשון מבטיח לנו ש-M יש לכל היותר ילד אחד שאינו עלה (כלומר, NULL). **נסמן את הבן של M ב-C, את האב של M ב-P ואת האח של M ב-S** (שימו לב - אם ל-M אין קודקודים שהוא מפנה אליהם, אלא רק עלים - זה נחשב שיש לו בן שחור, אחרת הכוונה לקודקוד אליו הוא מצביע).
הערה טכנית - בכל האיורים שיופיעו בחלק זה, קודקוד ללא ילד שמאלי/ימני מכיל הפניית NULL, כלומר - יש לו בן שהוא עלה (ולכן שחור). באיורים לא מופיעים העלים לטובת פשטות, אך שימו לב שהם נכללים בספירת הגובה השחור של העץ (black height).

נחלק את המחיקה של M למקרים:

1. **M הוא קודקוד אדום -** במקרה הזה, מובטח לנו מתכונות העץ ששני הילדים של M הם עלים (ולכן שחורים). לכן, פשוט נמחק את M ותכונות העץ יישמרו.
לדוגמא, בתרשים זרימה הבא נרצה למחוק את קודקוד "20", נסמנו M. מאחר ש-M הוא קודקוד אדום, אנחנו במקרה 1 ולכן ניתן למחוק אותו בלי לפגום באף תכונה של העץ.



2. **אם M הוא קודקוד שחור ו-C קודקוד אדום** – במקרה זה נמחק את M (כאשר אנחנו מחליפים אותו ב-C) ונשנה את הצבע של C לשחור. בדוגמה אנחנו רוצים למחוק את קודקוד "30", לכן אנחנו במקרה 2 – לכן נמחק את 30, נחליף אותו בבן האדום שלו (קודקוד "20") ונצבע את הבן בשחור.



3. **אם M הוא קודקוד שחור ו-C קודקוד שחור:**

תחילה נשים לב שמצב זה אפשרי רק כאשר M הוא "עלה בינארי", כלומר - שני הילדים שלו הם עלים (NULL)

כעת נגדיר מושג חדש - Double Black (DB). **מושג זה הינו סמנטי ואין צורך לסמן קודקודים בקוד.**
כאשר נמחק את M, ייתכן ואנו שוברים את התכונה של העץ לפיה בכל המסלולים מהשורש לעלה כלשהו יש אותו מספר של קודקודים שחורים. לכן, כאשר אנו מוחקים את M ומחליפים אותו ב-C (הפניית NULL במקרה זה), נסמן את C כ-DB, כדי שנדע שאנו צריכים לתקן את התכונה של העץ.
כלומר, סימון ה DB מסמל "במסלולים שעוברים בקודקודים הזה יש קודקוד שחור אחד פחות משאר המסלולים בעץ". **כעת, מוחקים את M ומחליפים אותו ב-C. כלל השלבים הבאים הם לאחר מחיקת M והשמת C במקומו, כאשר C מסומן ב- DB.** S ו-P נשארים זהים.
כעת נפעל ע"פ המקרים הבאים:

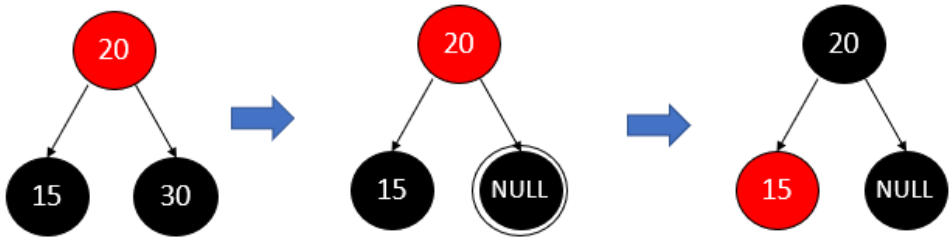
א. C הוא השורש:

נסיר את ה-DB וסיימנו עם המחיקה.

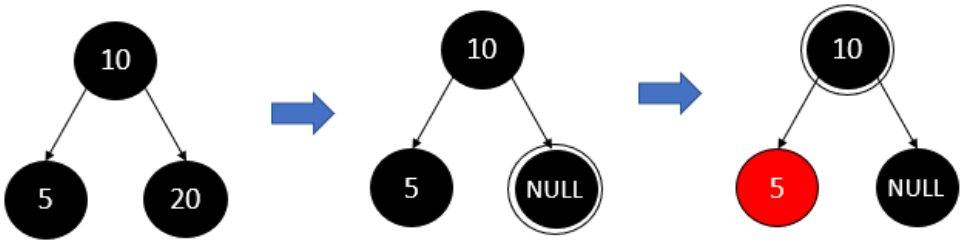
ב. האח של C (כלומר, S) הוא שחור ושני הבנים של S שחורים:

i. אם P הוא קודקוד אדום - נסיר את הDB, "נעביר" את השחור ל-P (כלומר P הופך לשחור) ואת S נהפוך לאדום.

לדוגמא, אם נרצה להסיר את "30" - מאחר שהוא שחור והבן שלו שחור (עלה) אנחנו במקרה 3, נמחק את M ונסמן את C שהחליף אותו ב-DB. כעת נעבור על המקרים לתיקון DB ונשים לב ש-P (הקודקוד עם הערך "20") הוא אדום, כלומר אנחנו במקרה 1.ב.1 - לכן נשנה את הצבע של P לשחור ואת הצבע של S (הקודקוד עם הערך "15") לאדום.



ii. אם P הוא קודקוד שחור - בדומה למקרה הקודם - "נשלח" את הבעיה ל-P, כלומר, "נעביר" שחור אחד ל-P. במקרה הזה P יהפוך להיות DB. בנוסף, נהפוך את S לאדום.



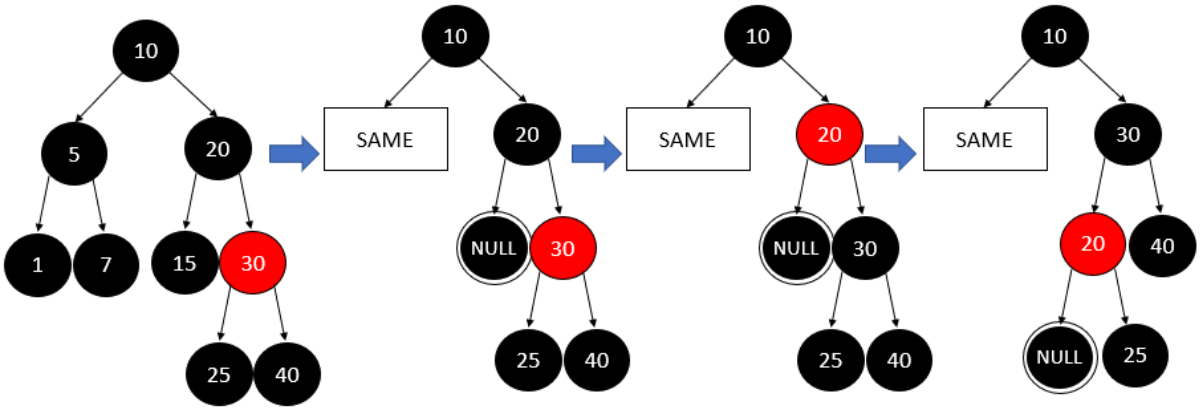
אך לא תיקנו את הבעיה שכן P הוא DB (שימו לב – המסלולים שעברו דרך P מכילים כעת קודקוד שחור אחד פחות משאר המסלולים בעץ). לכן, נסמן C=P (ואת כל הסימונים הנובעים מכך) ונפעל באופן רקורסיבי לתקן את ה-DB בהתאם למצב החדש - כאשר מתחילים משלב 3.א.

למשל, בדוגמא הנ"ל לאחר השלבים שהתבצעו לעיל הגענו לנקודה שבה P הוא DB. אז נסמן את "10" כ-C (כלומר, הקודקוד שהיה P), את ההורה שלו ב-P ואת האח שלו ב-S.

ג. **אם S אדום:**

במקרה הזה נחליף את הצבעים של S ושל P, כלומר – נצבע את S בשחור ואת P באדום. כעת, נעשה רוטציה ל-P לכיוון C.

לאחר מכן עדיין ישאר לנו הDB (על אותו קודקוד שהיה לפני השלב בתיקון) ונבצע את התיקון ל-DB באופן רקורסיבי (כלומר, מ3.א). בדוגמה אנחנו רוצים למחוק את "15", לכן מוחקים ומחליפים אותו בבן שלו ומסמנים את הבן שלו ב-DB (תחילת 3). עכשיו מריצים את התיקון על DB (מחפשים את המקרה המתאים מ-3.א והלאה). רואים שאנחנו במקרה 3.ג – לכן מחליפים את הצבעים של "20" (P) ו-"30" (S) ועושים רוטציה מ-"20" לכיוון הקודקוד DB. עכשיו נריץ את התיקון DB **על אותו קודקוד איתו התחלנו את התיקון** (לא העברנו את ה DB לקודקוד אחר)



ד. **אם S שחור והבנים של S מקיימים שהבן הרחוק מ-C שחור והבן שקרוב ל-C אדום:**

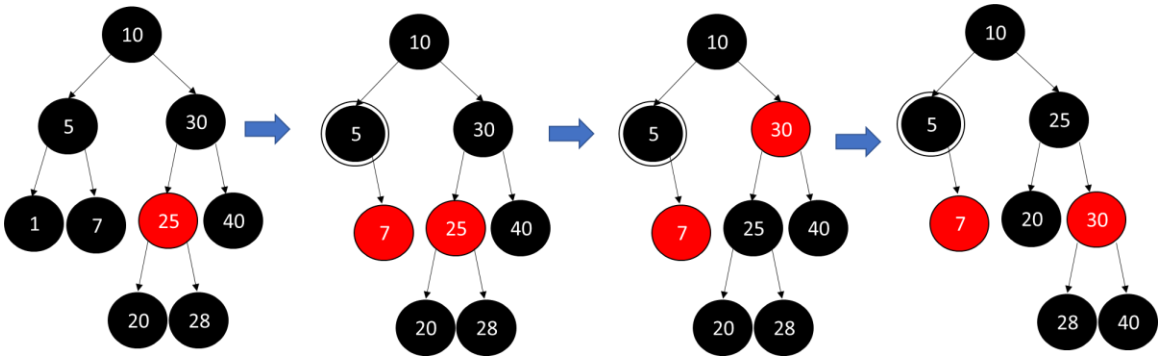
נסמן ב- S_C, S_F את הבן הקרוב והרחוק (מ-C, קודקוד ה-DB) של S בהתאמה. נפעל באופן הבא:

את S_C נהפוך לשחור ואת S לאדום. נעשה רוטציה ל-S לכיוון הנגדי מהDB. לאחר מכן נריץ את אלגוריתם תיקון ה-DB (בדיקת מקרים מ-3.א) על קודקוד ה-DB.

לדוגמא, נרצה למחוק את "1". נמחק ונחליף בבן העלה (NULL) שלו ונסמן ב-DB את העלה שהחליף את 1. בודקים את המקרים של תיקון DB ומבצעים את המקרה המתאים (3.ב.ii).

בסיומו מריצים שוב את תיקון ה-DB ורואים שאנחנו במקרה 3.ד. לכן, נחליף את הצבעים של S ("30") ו- S_c ("25"), ונבצע רוטציה מ-S לימין (הרחק מ-C, קודקוד ה-DB).

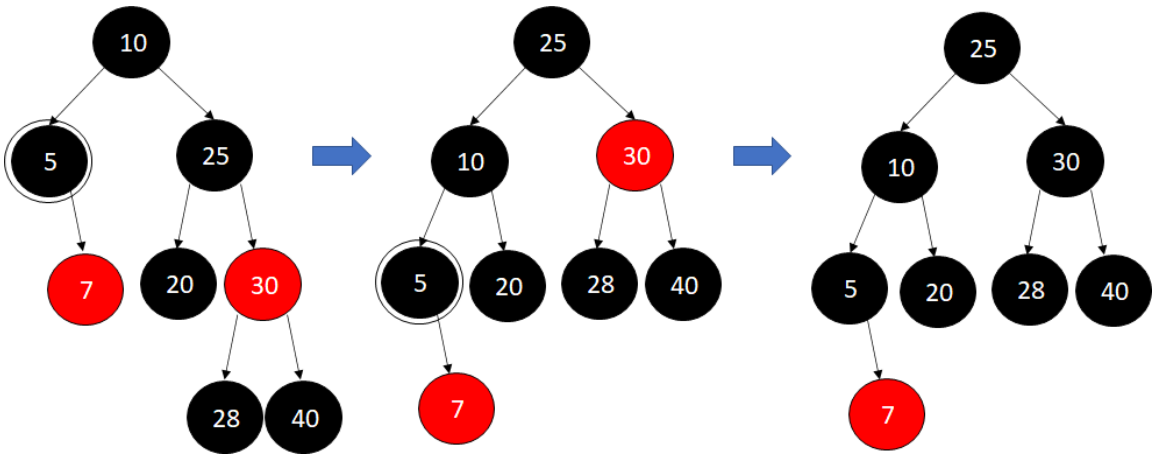
בסיום התיקון הגענו לעץ שהבעיה עדיין נוכחת (יש DB). נעשה בדיקה מחדש של המקרים - ונגלה שאנו נמצאים במקרה הבא - מקרה 3.ה.



ה. **אם S שחור והבנים של S מקיימים שהבן שרחוק מ-C אדום:**

במקרה הזה נבצע מספר פעולות - נחליף בין הצבעים של P ו-S, נעשה רוטציה ל-P לכיוון C (כלומר, S הופך להיות האבא של P) ונהפוך את מי שהיה הבן האדום של S (כלומר, מי שהיה הבן הרחוק שידוע שהוא אדום) לשחור. כעת נסיר את הDB מ-C (כלומר, כעת הוא יהיה פשוט בצבע שחור) וסיימנו.

לדוגמא, בהמשך לדוגמא הקדמת - נבצע את השלבים: הצבע של ההורה "10" זהה לצבע של האח "25", אז אין לנו צורך להחליף ביניהם את הצבעים. במעבר הראשון ניתן לראות שביצענו רוטציה לכיוון שמאל (לכיוון של ה-DB). במעבר השני הסרנו את הDB (נשאר רק B אחד) והחלפנו את הצבע של S_F ("30") לשחור.



שימו לב שיש ביצוע **רקורסיבי** בחלק מהשלבים – כאשר מפעילים את התיקון של העץ לאחר המחיקה.

חשבו – כיצד נכון ונוח לממש זאת?

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 (כי אנחנו יודעים שהוא מצביע לזיכרון שהוקצה דינמית).

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:

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 insertToRBTree(RBTree *tree, void *data)`
פונקציה זו מקבלת מצביע לעץ ולמידע, מכניסה את המידע לעץ במקום המתאים, מבצעת תיקון לעץ במידה ונדרש ומעדכנת את size. מחזירה 0 במקרה של כישלון (אם המידע נמצא כבר בעץ זה נחשב כישלון) וכל מספר אחר במקרה של הצלחה.
3. **מחיקת איבר מהעץ** – `int deleteFromRBTree(RBTree *tree, void *data)`
פונקציה זו מקבלת מצביע לעץ ומידע, מחפשת את המידע בעץ. אם המידע קיים בעץ, הפונקציה מוחקת אותו (ומפעילה עליו את פונקציית השחרור), מעדכנת את size ומבצעת את התיקונים הנדרשים לעץ. מחזירה 0 במקרה של כישלון (אם המידע לא בעץ זה נחשב ככישלון) וכל מספר אחר במקרה של הצלחה.
4. **חיפוש בעץ** – `int RBTreeContains(const RBTree *tree, const 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.
5. **הפעלת פונקציה על כל איברי העץ (in-order)** –
`int forEachRBTree(const 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*).
6. **שחרור המשאבים של העץ** – `void freeRBTree(RBTree **tree)`
פונקציה זו מקבלת מצביע למצביע 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 הראשון 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 (deep copy) את התוכן של הוקטור הראשון לוקטור השני במידה והנורמה (l_2) של הוקטור הראשון גדולה יותר מהנורמה של הוקטור השני (אם הנורמה של הוקטור הראשון לא גדולה מהנורמה של השני – לא עושים כלום וזה לא נחשב כישלון!). תזכורת, עבור וקטור $v = [a_1 \quad \cdots \quad a_n]$ הנורמה היא

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

יש לבצע realloc לשדה vector במשתנה maxVector במידת הצורך.

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. יש לשחרר גם את השדה המוקצה וגם את הstruct עצמו.

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

- 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. באופן כללי, כאשר עולה לכם השאלה "האם אני צריך לשחרר את הזיכרון או המשתמש?" – נסו לחשוב כיצד **נכון** שספרייה תתנהג וכמובן – בדקו את פתרון בית הספר!
- 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 שלא יבוצעו שינויים לעץ שלא דרך פונקציות הספריה.
- ניתן להניח כי לפונקציית השוואה שנמצאת בעץ יוזנו רק מצביעי void* שמצביעים לסוג המשתנה שהפונקציה מצפה לקבל.
- ניתן להניח שמצביע שהוקצה בהקצאה דינמית והוכנס לעץ לא ישוחרר במהלך התוכנית, אלא רק באמצעות פונקציית השחרור של העץ.
- ניתן להניח כי מחרוזות שיינתנו לעץ יוקצו בצורה דינמית ויש לשחרר אותם בפונקציית השחרור.
- ניתן להניח שבסוף כל מחרוזת מופיע '\0'.
- ניתן להניח שהוקצה מספיק מקום למחרוזות בפונקציית השרשור.
- ניתן להניח כי שדה ה- double* ב-struct של Vector יוקצה דינמית.
- ניתן להניח שלא תגיעו לעומק רקורסיה מקסימלי.
- ניתן להניח שלא יועבר NULL במקום מצביע לפונקציה.
- יש לעקוב אחר גודל העץ באמצעות השדה size. יש לספור רק קודקודים שמכילים מידע (לא לספור את העלים)
- לא ניתן להניח שמצביעים שעוברים לפונקציות אינם NULL. במידה ועובר מצביע NULL לפונקציה שלא אמורה לקבל NULL – על הפונקציה להחזיר כישלון. אותו הדבר נכון לגבי שדה ב-struct.
- ניתן להניח כי פונקציות השוואה לא יקבלו ארגומנטים לא תקינים (מאחר ואין דרך לסמל כישלון בפונקציה כזו)
- ניתן ואף רצוי לממש פונקציות נוספות שאינן בקובץ RBTree.h לטובת מימוש הפונקציות שמופיעות בקובץ.
- אין לממש קודקוד מיוחד שמסמל עלים (הפניות NULL).
- אין לממש קודקוד עם data==NULL בתור עלה – יש להשתמש בהפניית NULL!
- הכנסה של NULL ב-data לעץ נחשבת שגיאה.
- DB הוא סימון סמנטי – אין צורך באמת לבצע סימון במהלך אלגוריתם המחיקה.

- לא מעניין אותנו באיזה אלגוריתם אתם משתמשים! אם אינכם רוצים לממש בצורה רקורסיבית – אל תממשו בצורה רקורסיבית. אם אתם רוצים להשתמש באלגוריתם תיקון אחר לעץ – עשו זאת. העיקר שהעץ יקיים את התכונות לאחר הכנסה ומחיקה.
- ניתן לממש מחיקה/הכנסה ע"פ אלגוריתם שאינו מתואר ב-PDF, אתם תיבדקו על תקינות העץ – לא על מימוש ספציפי שלו.
- 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

```
~labcc2/www/ex3/school_solution/
```

- הרצת בדיקה ל-coding style:

```
~labcc2/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

```
~labcc2/www/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