מבני נתונים – תרגיל 4 – 2D-Range Tree

תאריך פרסום: 1.5.2014

תאריך הגשה: 22.5.2014

מרצה ומתרגלים אחראים: פרופ' קלרה קדם, ענר בן-אפרים, כפיר וולפסון

1. נושאי העבודה:

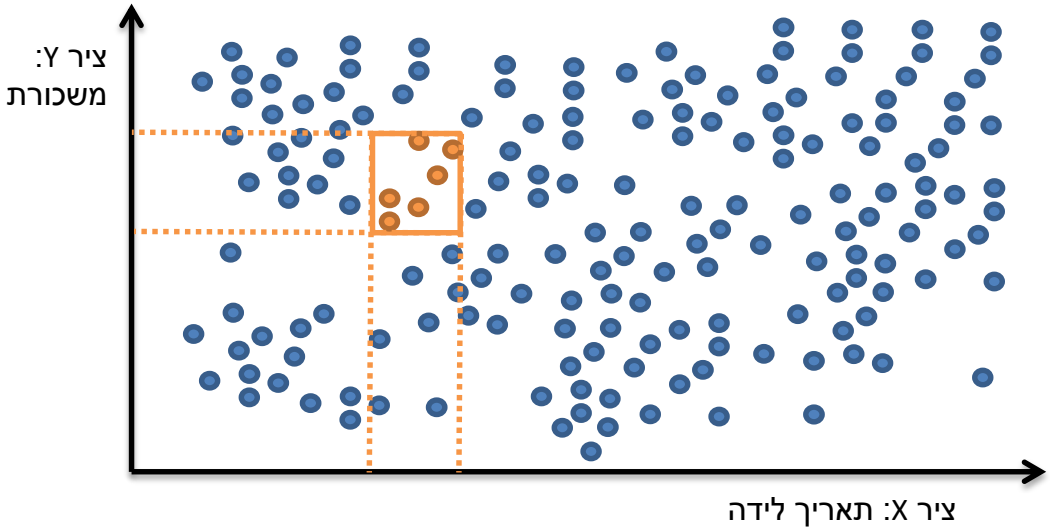
- תכנון מבנה נתונים יעיל מבוסס 2D-Range Tree
- ניתוח זמן הריצה של המבנה
- מימוש המבנה ב-Java ובדיקתו

הערות ושאלות יש להפנות בפורום הקורס בכתובת הבאה : http:/www.cs.bgu.ac.il/~ds142/Forum

2. מבוא:

חברת הסטארט-אפ "נקודה.קום" רוצה לפתח מוצר מבוסס מבנה נתונים המכיל אוסף של נקודות במרחב דו-מימדי, הנבנה במהירות וכן מסוגל לבצע פעולות ולענות על שאילתות בצורה יעילה. דוגמאות לאפליקציות אפשריות של המוצר:

- מציאת צפיפות נקודות ישוב במפה
- זיהוי פרצופים בתוך תמונה
- הצלבה של נתונים מתוך מאגר בשני מימדים, כגון מיהם העובדים בטווח הגילאים 20 עד 30, המרוויחים משכורת שבין 10,000 ל-17,000 ₪. ראו איור 0.



איור 0: שאילתת טווח דו מימדית

3. רקע:

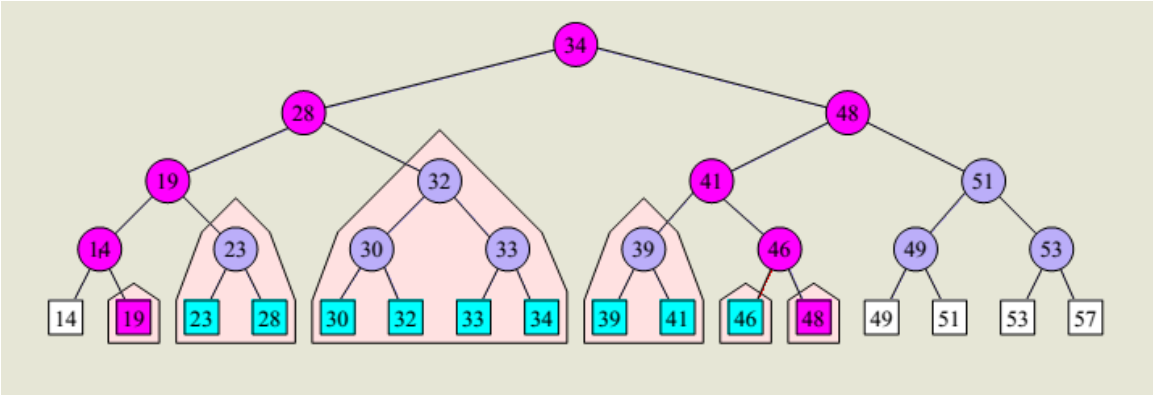
3.1. 1D-Range Tree

תהא S קבוצה של n נקודות במרחב $\mathbb{Z}$, כלומר אוסף מספרים שלמים. שאילתת חיפוש חד מימדית תבקש לדווח את כל הנקודות מ- S המוכלות בטווח השאילתה $[x_1, x_2]$, כאשר $x_1<x_2$ (כולל ערכים ששווים ממש לקצוות הטווח). השאילתה מומחשת ויזואלית באיור 1.

איור 1: שאילתת טווח חד מימדית.

בכדי לענות ביעילות על שאילתות טווח חד מימדיות, נהוג לבנות One-Dimensional Range Tree, (עץ טווחים עבוד מימד יחיד), שהוא סוג של עץ חיפוש בינארי, בו **הערכים השמורים במבנה נמצאים רק בעלים**. את העץ בונים מראש מאוסף של ערכים בציר יחיד (ציר X), כלומר סקלרים.

בכל צומת **פנימית** בעץ, נחזיק את הערך המקסימלי שנמצא בתת העץ שמאלי של הצומת, כפי שניתן לראות בדוגמה שבאיור 2. המבנה באיור מכיל 16 ערכים – העלים של העץ (שבצורת ריבוע). כל שאר המפתחות הנמצאים בצמתים פנימיים (בצורת עיגול), נמצאים שם כדי לעזור לנו לחפש בתוך המבנה בצורה יעילה. החוקיות היא כמו בעץ חיפוש בינארי: הערכים שבתת העץ השמאלי של כל צומת פנימית קטנים או שווים למפתח השמור בצומת, והערכים בתת העץ הימני גדולים ממש ממנו.



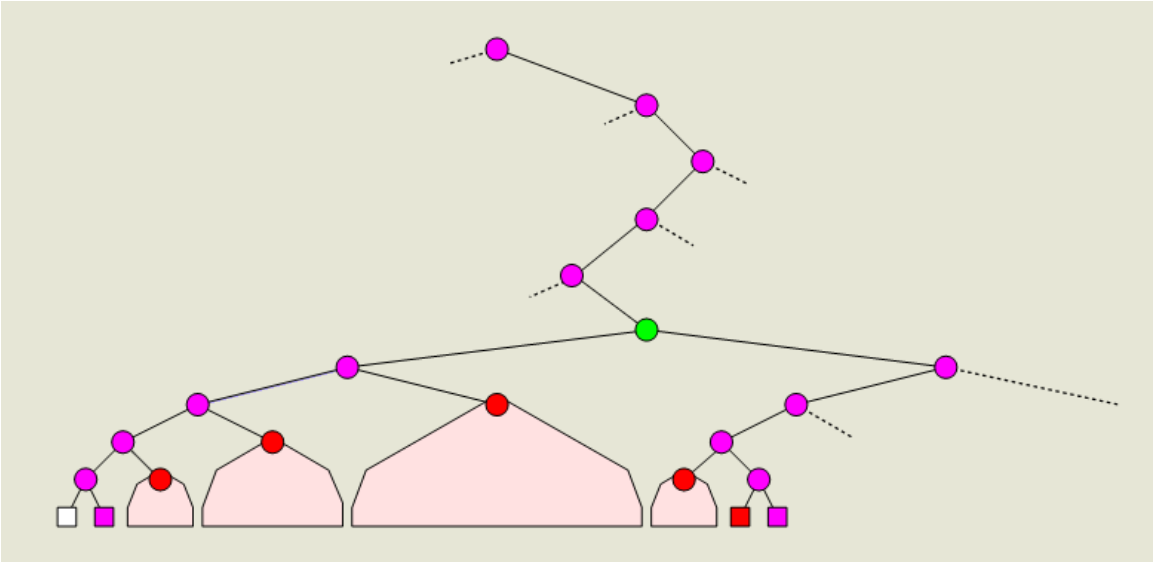
איור 2: דוגמה ל - 1D-Range Tree

שאילתה נפוצה בעץ טווחים עבוד מימד יחיד היא, כאמור, מציאת כל האיברים במבנה שנמצאים בתוך טווח נתון $[x_1, x_2]$. שאילתה נוספת עשויה לבקש רק את **כמות** האיברים שנמצאים בטווח הנתון.

היזכרו בשאלות מתרגול 5, בהן מצאנו בעץ חיפוש בינארי את כמות האיברים ואת האיברים עצמם בטווח מסוים. בעץ טווחים הפתרון הוא דומה, רק שאנחנו נחזיר רק ערכים שנמצאים בעלים (כאמור, כל שאר המפתחות נמצאים בעץ כדי לעזור בחיפוש, אך אינם "איברים" בתוך מבנה הנתונים).

באיור 2, ניתן לראות שאילתה עבור הטווח [17,48]. שאילתת הכמות תחזיר "11". שאילתת הערכים תחזיר [19, 23, 28, 30, 32, 33, 34, 39, 41, 46, 48], לאו דווקא מסודרים בסדר הזה.

הדוגמה הנ"ל, בה השורש הוא "נקודת הפיצול" אינה המקרה הכללי. במקרה הכללי, נצטרך לחפש קודם נקודה זו בעץ – זהו הצומת המשותף האחרון למסלול החיפוש של X_1 ושל X_2 כשמתחילים את החיפוש משורש העץ. לצומת זה נקרא v_{split}. איור 3 מציג זאת בצורה סכמתית.



איור 3: ציור סכמתי של חיפוש בעץ טווח עבור מימד יחיד. הקודקוד v_{split} מסומן בירוק.

3.2. בנייה של עץ טווח

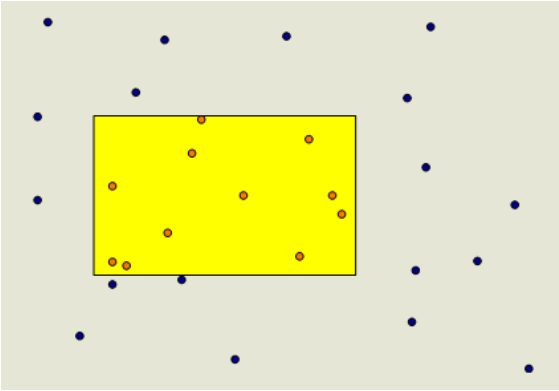
מכיוון שהשאילתות בעץ טווח תלויות בגובה העץ, הבנייה של עץ הטווח תדאג שהוא יהיה מאוזן ככל האפשר. הבניה תיעשה על ידי כך שתחילה נמיין את הנקודות, ואז נבנה את העץ בצורה רקורסיבית. חישבו מדוע המיון עוזר בבניית עץ מאוזן.

```
Build-1D-RangeTree(points)
input: an array of points
output: a 1D-Range Tree

1. sortedPoints ← Sort(points)
2. return Build-1D-RangeTree(sortedPoints, 0, points.size)
```

```
Build-1D-RangeTree(sortedPoints, from, to)

1. new Node
2. if (to-from=1) //leaf
      a. Node.data ← sortedPoints[from]
      b. return Node
3. mid ← from + ⌊(to-from)/2⌋
4. Node.data ← sortedPoints[mid-1]
5. Node.left ← Build-1D-RangeTree(sortedPoints, from, mid)
6. Node.right ← Build-1D-RangeTree(sortedPoints, mid+1, to)
7. Return Node
```

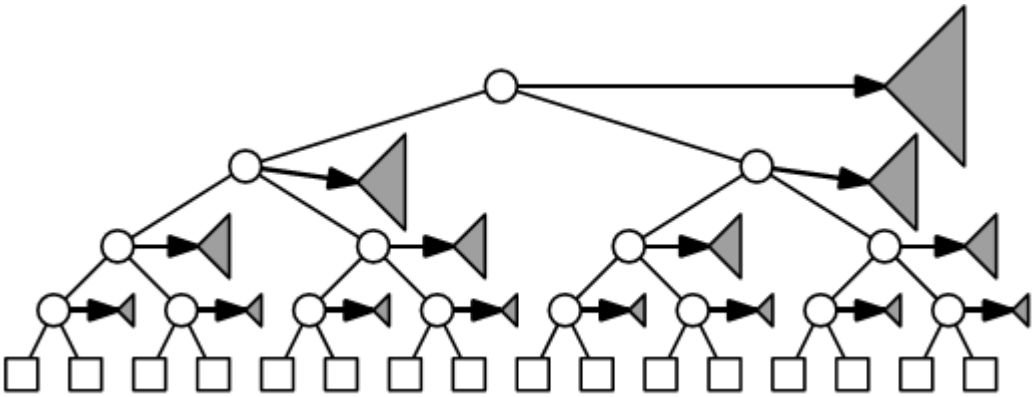
3.3. מעבר לשני מימדים – 2D-Range Tree

תהא S קבוצה של נקודות במרחב $\mathbb{Z}^2$, כלומר קואורדינטות שלמות במישור. שאילתת חיפוש דו-ממדית עשויה לבקש את אוסף הנקודות ב- S שנמצאות במלבן $[x_1, x_2] \times [y_1, y_2]$. כלומר במלבן מקביל לצירים, כאשר $x_1<x_2$, $y_1<y_2$, והטווח כולל את קצוותיו, כפי שניתן לראות באיור 4.



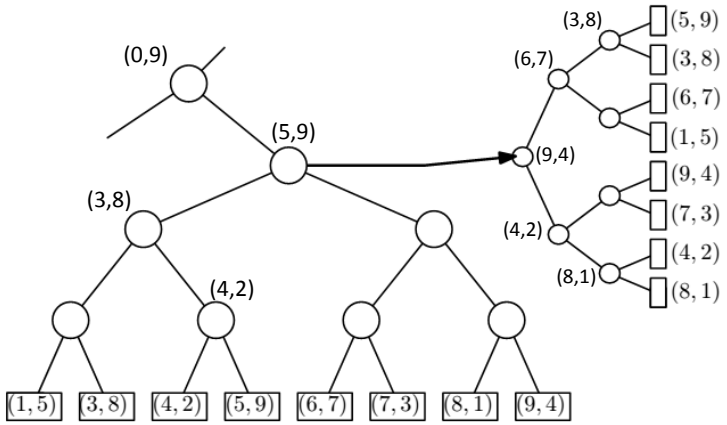
איור 4: שאילתת טווח דו-מימדית

בכדי לענות ביעילות על שאילתות טווח דו מימדיות, אנחנו נשתמש בעץ טווחים עבור שני ממדים, או 2D-Range Tree. מבנה זה מחזיק אוסף נקודות במישור (x, y), ומכיל מספר עצים: **עץ ראשי**, שנקרא גם עץ ה-X, הינו 1D-Range Tree המחזיק נקודות וממויין על פי ציר ה-X של הנקודות. כל צומת פנימית בעץ הראשי תכיל מצביע לעץ נוסף, **עץ Y,** גם הוא 1D-Range Tree אך הוא ממויין על פי ציר ה-Y של הנקודות שבו. ראו איור 5.



איור 5: 2D-Range Tree, **כל תת-עץ (קודקוד פנימי) בעץ הראשי (עץ ה-X), מצביע על עץ** Y **משלו.**

הנקודות בעץ המשני, עץ ה-Y, יהיו בדיוק אותן הנקודות שנמצאות בתת העץ שמצביע עליו, כפי שניתן לראות באיור 6.



איור 6: קטע מתוך 2D-Range Tree. כל תת-עץ בעץ ה-X, מצביע על עץ Y. עץ Y זה מכיל בדיוק את אותן הנקודות שמכיל תת-העץ שמצביע עליו, אך ממויין על פי ציר ה-Y שלהן. לשם הנוחות, לא שורטטו כל המפתחות בקודקודים הפנימיים.

כזכור, הערכים במבנה נשמרים רק ב**עלים**, לכן אוסף העלים בעץ ה-Y ובתת העץ שמצביע עליו יהיו שווים, אך לא הצמתים הפנימיים של עצים אלו. מן הסתם, הסדר בין העלים יהיה שונה כי עץ ה- Y ממויין על פי ציר Y של הנקודות.

לשם נוחות, לא שורטטו באיור 6 כל המפתחות בקודקודים הפנימיים בעצים. מפתחות אלו גם הם נקודות, וכמו במימד יחיד, המפתח בקודקוד פנימי יהיה זהה לנקודה "הגדולה ביותר" בתת העץ השמאלי שלו. ההגדרה של "גדולה ביותר" שונה בין העצים: בעץ ה-X המיון נעשה על פי ערך ה-X, ובעצי ה-Y על פי ערך ה-Y של הנקודות.

אז כמה עצי Y נמצאים ב 2D-Range Tree? לכל צומת פנימית יש עץ Y משלו, ועל כן יש O(n)עצי Y.

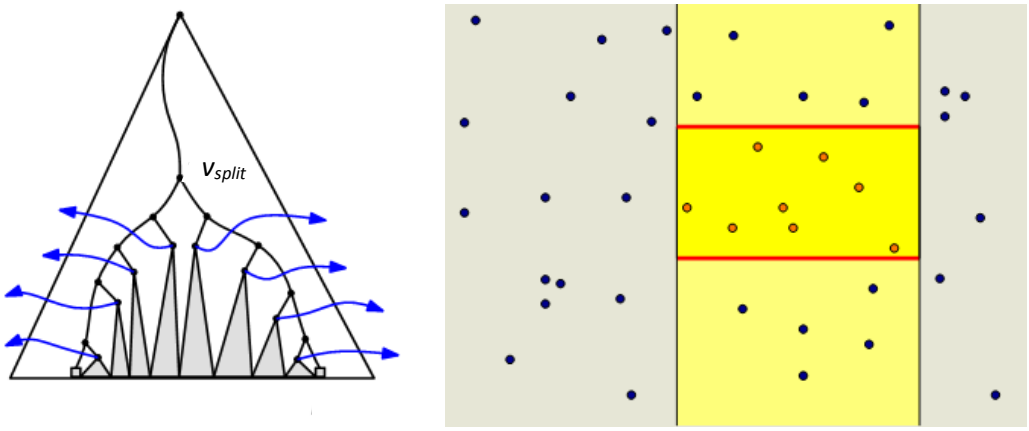
3.4. בנייה של 2D-RangeTree

בנייה של 2D-RangeTree תיעשה בדומה ל 1D-RangeTree, אך נשים לב למיין מראש את הנקודות על פי שני הצירים. חישבו מדוע.

```
Build-2D-RangeTree(Points[] points)
input: an array of points
output: a 2D-Range Tree

1. sortedPointsX ← SortAccordingToX(points)
2. sortedPointsY ← SortAccordingToY(points)
3. return
    Build-2D-RangeTree(sortedPointsX, 0, points.size, sortedPointsY)


Build-2D-RangeTree(sortedPointsX, from, to, sortedPointsY)

1. new Node
2. if (to-from=1) //leaf
      a. Node.data ← sortedPointsX[from]
      b. return Node
3. mid ← from + ⌊(to-from)/2⌋
4. Node.data ← sortedPointsX[mid-1]
5. Node.treeY ←
     Build-1D-RangeTree(sortedPointsY, 0, sortedPointsY.size)*
6. (leftPts, rightPts) ← Partition(sortedPointsY, Node.data)**
7. Node.left ← Build-2D-RangeTree(sortedPointsX, from, mid, leftPts)
8. Node.right ← Build-2D-RangeTree(sortedPointsX, mid+1, to, rightPts)
9. return Node

Notes:

* treeY will hold the points sorted according to the Y coordinate.

** Partition(A, P) function partitions the input array A into two
arrays. The first contains elements less than or equal to P (according
to X coordinate) and the second array contains the rest, while
preserving order of A elements.
```

3.5. שאילתות בשני מימדים

שאילתות ב 2D-RangeTree, המבקשות את כל הנקודות במלבן מסוים, תחילה יחפשו בעץ ה-X את תתי העצים המחזיקים את הנקודות המתאימות לטווח המבוקש בציר X. עבור כל תת-עץ כזה, יש לעבור לעץ ה-Y שלו ולחפש שם את כל הנקודות המתאימות גם לטווח המבוקש בציר ה-Y. ראו איור 7. חשבו מה אתם עושים אם הגעתם לעלה בעץ ה-X, על מנת לענות נכונה על השאילתה.



איור 7: שאילתת טווח דו-מימדית תסנן קודם כל על פי ציר ה-X, ואז על פי ציר ה-Y.

מקורות האיורים:

[Lecture 8, Computational Geometry course, University of Florida]

[Lecture 8, Geometric Data Structures course, Otto von Guericke University]

4. מבנה הנתונים:

בעבודה זו תתכננו ותממשו את מבנה הנתונים המתאים למספר פעולות נדרשות. על המימוש להיות מבוסס 2D Range Tree. המבנה יחזיק נקודות במישור: מופעים של המחלקה Point, שמימושה מצורף לתרגיל. Point מכיל שלושה שדות: ערך על ציר X, ערך על ציר Y ושם הנקודה.

הערות חשובות:

1. **אין לשנות את המחלקה Point.**
2. לא יהיו 2 נקודות בעלות אותן קואורדינטות בדיוק (כלומר גם אותו X וגם אותו Y), **אך ייתכנו מספר נקודות עם אותו ערך X וכן מספר נקודות עם אותו ערך Y.** (לדוג' (7,5), (3,5), (3,9))

הפעולות שמבנה הנתונים נדרש לתמוך בהן:

#	OPERATION		Running Time
1	(constructor)	**build TwoDRangeTree** (Point[] points)	O(nlog n)
2	int	**numOfPointsInRectangle**(Point p1, Point p2)	$O(\log^2 n)$
3	Point[]	**getPointsInRectangle**(Point p1, Point p2)	$O(k+\log^2 n)$
4	void	**addPoint**(Point point)*	$O(\log^2 n)$
5	bool	**removePoint**(Point point)*	$O(\log^2 n)$ upon success O(logn) upon failure
6	bool	**existsPoint**(Point point)	O(log n)
7	int	**numOfPointsInHalfPlaneX**(int X, bool greaterThan)	O(log n)
8	int	**numOfPointsInHalfPlaneY**(int Y, bool greaterThan)	O(log n)
9	Point[]	**getAllPoints**()	O(n)

* **שימו לב:** ניתן להניח כי הפעולות addPoint() ו- removePoint() תקראנה לכל היותר O(log n) פעמים.

רשימת הפעולות הנ"ל, פרט לבנאי, נמצאת בממשק (Java Interface) בשם DRT אותו עליכם לממש (implement) בעזרת מחלקה שאתם תכתבו בשם TwoDRangeTree.

עליכם לחשוב כיצד אתם ממשים את מבנה הנתונים. תכננו היטב את המחלקות שלהן תידרשו.

4.1. פירוט הפעולות

בכל אחת מן הפעולות הבאות, n מייצג את מספר הנקודות ההתחלתי במבנה.

1. `TwoDRangeTree(Point[] points)`

בניית המבנה. קלט: מערך של n נקודות. זמן ריצה: O(nlogn)

ניתן להניח שכל הנקודות שונות זו מזו, אך עשויות להיות מספר נקודות עם אותו ערך X **או** אותו ערך Y.

2. `int numOfPointsInRectangle(Point p1, Point p2)`

שאילתה המחזירה את מספר הנקודות בתוך מלבן נתון. זמן ריצה: $O(\log^2 n)$

קלט: שתי נקודות המייצגות את פינות המלבן. ניתן להניח כי תמיד $p1.x \le p2.x$ וגם $p1.y \le p2.y$.
לא ניתן להניח כי הנקודות p1 או p2 נמצאות במבנה, הן רק מייצגות את מלבן השאילתה.

על השאילתה לספור גם נקודות הנמצאות ממש על צלעות המלבן.

3. `Point[] getPointsInRectangle(Point p1, Point p2)`

שאילתה המחזירה את הנקודות הנמצאות בתוך מלבן נתון. זמן ריצה: $O(k+\log^2 n)$
(k = מס' הנקודות בטווח, כלומר אורך המערך המוחזר ע"י הפונקציה)

קלט: שתי נקודות המייצגות את פינות המלבן. ניתן להניח כי תמיד $p1.x \le p2.x$ וגם $p1.y \le p2.y$.
לא ניתן להניח כי הנקודות p1 או p2 נמצאות במבנה, הן רק מייצגות את מלבן השאילתה.

על השאילתה להחזיר גם נקודות הנמצאות ממש על צלעות המלבן.

אין חשיבות לסדר הנקודות במערך המוחזר.

4. `void addPoint(Point point)`

פעולה המוסיפה נקודה למבנה. זמן ריצה: $O(\log^2 n)$

קלט: נקודה חדשה שניתן להניח שאיננה נמצאת כבר במבנה

חשוב: פעולה זו תקרא לכל היותר $O(\log n)$ פעמים.

5. `boolean removePoint(Point point);`

פעולה המסירה נקודה מן המבנה. קלט: נקודה ש<u>לא יודעים</u> אם היא במבנה או לא.

אם הנקודה אינה קיימת במבנה על הפונקציה להחזיר false ולרוץ בזמן: $O(\log n)$

אם הנקודה אכן קיימת במבנה, על הפונקציה להסיר אותה, להחזיר true ולרוץ בזמן: $O(\log^2 n)$

חשוב: פעולה זו תקרא לכל היותר $O(\log n)$ פעמים.

6. `boolean existsPoint(Point point)`

פעולה הבודקת אם נקודה נמצאת במבנה. קלט: נקודה שלא יודעים אם היא במבנה או לא.

זמן ריצה: $O(\log n)$

7. `int numOfPointsInHalfPlaneX(int X, boolean greaterThan)`

שאילתה המחזירה את מספר הנקודות במבנה, בחצי המישור המבוקש על ציר X. ראו דוגמה באיור 8.

קלט: ערך "X" וערך בוליאני המציין האם הנקודות המבוקשות גדולות או קטנות מ-"X".

על הפונקציה להחזיר כמה נקודות קיימות במבנה, אשר ערך ה- X שלהן גדול או שווה לקלט "X" (אם greaterThan הוא true), או קטן או שווה לקלט "X" (אם greaterThan הוא false).

זמן ריצה: O(logn)

על השאילתה לספור גם נקודות הנמצאות ממש על גבי גבול חצי המישור, כלומר ערך ה-X שלהן שווה לקלט "X".

```
8.  int numOfPointsInHalfPlaneY(int Y, boolean greaterThan)
```

שאילתה המחזירה את מספר הנקודות במבנה, בחצי המישור המבוקש על ציר Y.

קלט: ערך "Y" וערך בוליאני המציין האם הנקודות המבוקשות גדולות או קטנות מ-"Y".

על הפונקציה להחזיר כמה נקודות קיימות במבנה, אשר ערך ה- Y שלהן גדול או שווה לקלט "Y" (אם greaterThan הוא true), או קטן או שווה לקלט "Y" (אם greaterThan הוא false).

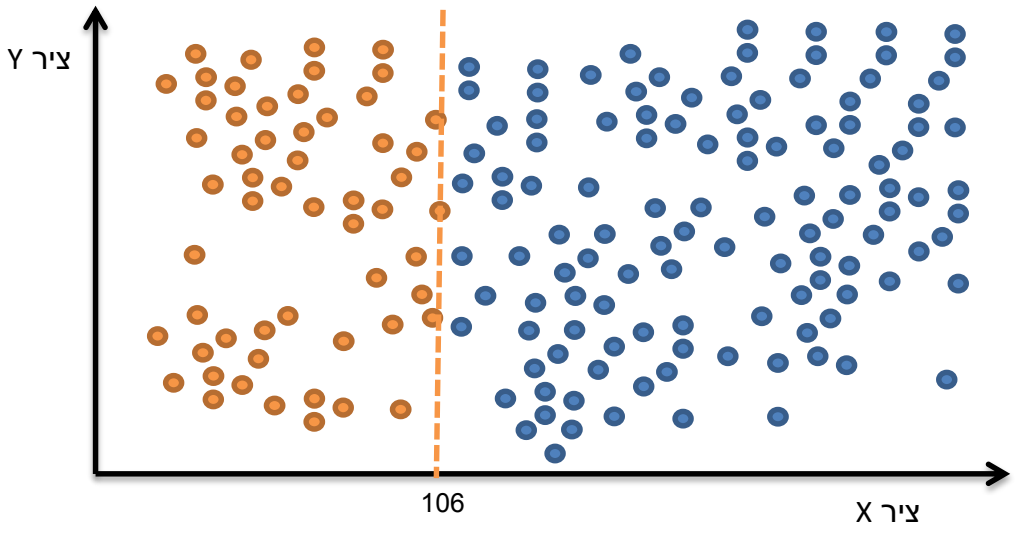
זמן ריצה: O(logn)

על השאילתה לספור גם נקודות הנמצאות ממש על גבי גבול חצי המישור, כלומר ערך ה-Y שלהן שווה לקלט "Y".

```
9.  Point[] getAllPoints ()
```

שאילתה המחזירה את כל הנקודות במבנה. זמן ריצה: O(n).

אין חשיבות לסדר הנקודות במערך המוחזר.



איור 8: השאילתה `numOfPointsInHalfPlaneX(106, false)` **תחזיר בדוגמה זו 56, כלומר מספר הנקודות הכתומות.**

5. חלק א' – תרגיל תיאורטי:

תארו בקצרה את המימוש שלכם במסמך מוקלד. יש בפרט להסביר את השינויים שביצעתם מגרסת הה2DRangeTree המוזכרת בפרק 3.

הסבירו בקצרה את זמן הריצה של **כל אחת** מן הפעולות.

על המסמך להיות בפורמט PDF בלבד.

6. חלק ב' – תרגיל מעשי:

עליכם לממש את מבנה הנתונים.

לתרגיל מצורף קובץ ZIP ובו:

- המחלקה Point שאסור לכם לשנות
- הממשק DRT שאסור לכם לשנות
- המחלקה TwoDRangeTree שעליכם להשלים.
- מחלקה בשם Main עם מספר פונקציות לבדיקת נכונות המבנה. הבדיקות אינן מכסות את כל המקרים ומומלץ להוסיף בדיקות משלכם.
- המחלקה GUI (ראו סעיף 7)

מותר לכם להוסיף מחלקות בהתאם לצרכי המימוש שבחרתם.

אתם מקבלים את קובץ המחלקה TwoDRangeTree עם מימוש ריק. ניתן להוסיף לה מתודות, בנאים, ושדות.

6.1. הערות על המימוש

מספר שינויים בתרגיל שלנו לעומת 2D-RangeTree המתואר בפרק הרקע:

- חישבו כיצד אתם משפרים את מבנה הנתונים והאלגוריתם כדי לעמוד בזמני הריצה של כל השאילתות
- לרוב לא מוסיפים או מסירים איברים לעץ טווחים לאחר שכבר נבנה. חישבו כיצד אתם ממשים את הפונקציות אלו מבלי להרוס את המבנה או את זמני הריצה של השאילתות העתידיות עליו.
- בדוגמאות המספריות בפרק הרקע לא היו שתי נקודות עם אותו ערך X או אותו ערך Y. שנו מעט את האלגוריתמיקה כך שתתמוך בנקודות כאלו, כלומר שהמבנה יוכל למשל להחזיק את הנקודות הבאות (7,5) ,(3,5), (3,9). אין דרישה שהמבנה יתמוך ביותר מנקודה אחת עם בדיוק אותן קואורדינטות (X,Y). הסבירו (בסעיף א') את השינויים שבצעתם.

אין זה חובה, אך אתם רשאים להשתמש בפונקציות סטטיות של המחלקה Arrays, כגון Arrays.sort() הממיינת מערך, Arrays.copyOf(), Arrays.toString() וכו'. כדי להשתמש ב- sort() מומלץ לממש Comparator המשווה בין 2 נקודות. חישבו לכמה Comparators תזדקקו והיכן תוכלו להשתמש בהם בחלקים נוספים בעבודה. אם אינכם מכירים Comparator, תוכלו לחפש חומר ודוגמאות ברשת.

7. חלק ג' – ממשק משתמש גרפי (GUI):

בחלק זה תוכלו לבחון שימוש אפשרי למבנה שבניתם

בקובץ ה-ZIP תמצאו גם מחלקה בשם GUI, שהרצתה תפתח ממשק משתמש. לאחר מימוש חלק ב', ניתן להשתמש בממשק זה לבדיקת הקוד שלכם.

האפליקציה יודעת לטעון קובץ תמונה וקובץ TXT שמייצג בסיס נתונים של העצמים. תוך שימוש בפתרון חלק ב', האפליקציה יודעת להחזיר אילו עצמים מופיעים במלבן אותו אתם מסמנים, להוסיף נקודות ועוד.

האפליקציה מאוד פשוטה לתפעול. תחילה יש לטעון את התמונה וקובץ ה TXT שנכתב בפורמט מיוחד(ומתואר למטה) המתאים לתמונה המסוימת. לאחר מכן יש לבחור מלבן באמצעות לחיצה על העכבר וגרירה על פני התמונה. לאחר עזיבת העכבר הפלט יופיע למטה.

כמו-כן, בפעולה getPoints יודפסו הנקודות הקיימות במבנה הנתונים על התמונה. (שימו לב שבמצב הסטנדרטי, הנקודות אינן מוצגות על התמונה).

ניתן לבצע רק חלק מהפעולות שמימשתם בעזרת האפליקציה, על כן זכרו לבדוק גם באמצעות הוספת פעולות ושאילתות במחלקה ה-Main.

מצורפים לעבודה שני סטים של קבצים. כל סט מכיל תמונה (קובץ JPG) ומיקומי אובייקטים (קובץ TXT)

אתם יכולים להשתמש גם בהם כדי לבדוק את נכונות המבנה.

כמו כן, אתם יכולים להוסיף סטים של קבצים.
הפורמט של קובץ הנקודות מסוג TXT צריך להיות:

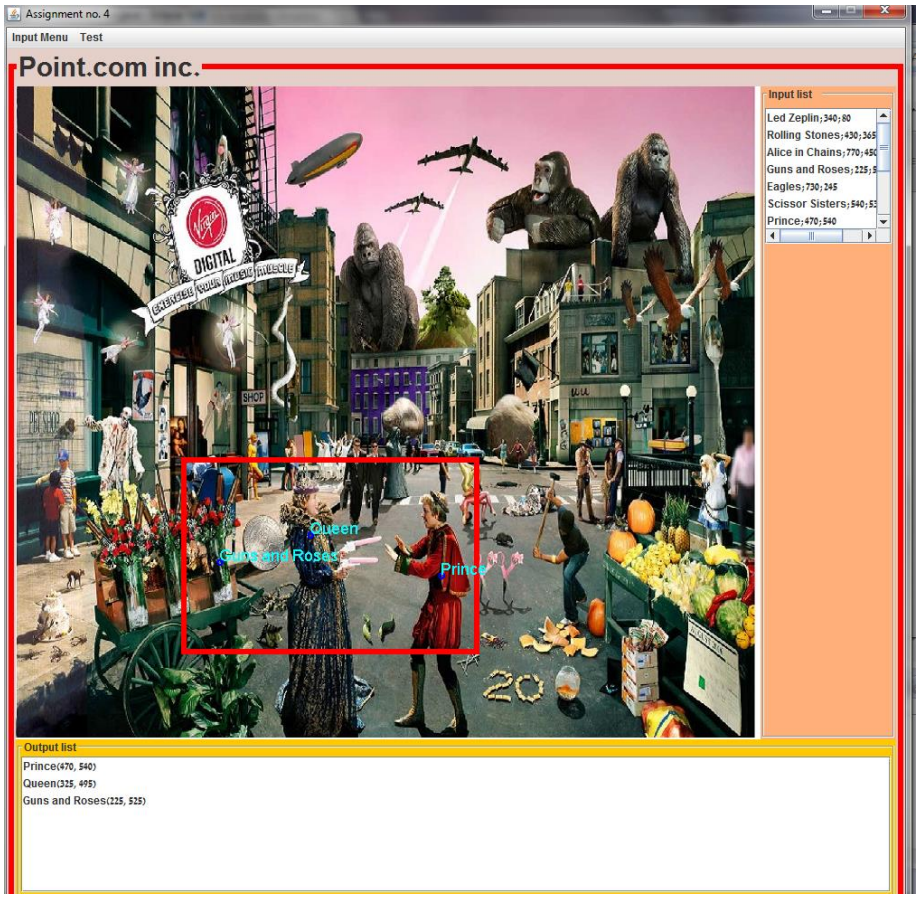
name;x_value;y_value

לדוגמה:

Moshe;30;55

אתם יכולים לצרף לעבודה את הסטים של הקבצים שהשתמשתם בהם.
נשקול לתת **בונוס של עד 10 נקודות** לציון העבודה במידה והם יהיו מקוריים במיוחד.

צילום מסך לדוגמה:



הערות חשובות ודרישות הגשה:

1. אתם רשאים להשתמש בפונקציות הסטטיות של המחלקה Arrays וממשק Comparator בשביל המיון.
2. פרט לכך, אין להשתמש במבני נתונים גנריים הקיימים ב - Java במימוש העבודה. כלומר, עליכם לממש את מבנה העץ וכל מחלקה נוספת שתידרשו לה בעצמכם. (במחלקות לבדיקה Main ו GUI אתם רשאים להשתמש בהכל, כיוון שלא מגישים מחלקות אלו).
3. ניתן להניח שהקלט יהיה תקין. לא תתבקשו להוסיף נקודה שכבר קיימת. לא תקבלו מתכנית הבדיקה שלנו ערך null כפרמטר לאף אחת מהפונקציות בממשק.
4. בין הקבצים, תקבלו גם קובץ Main.java שישמש אתכם לבדיקה. לאחר שתסיימו את התכנית אתם יכולים להריץ את ה- main כדי לדעת אם התכנית עובדת כמו שצריך. **הבדיקות אינן מכסות את כל המקרים ומומלץ להוסיף בדיקות משלכם**.
5. את העבודה יש להגיש ל Submission system.
6. נא לא ליצור package חדש, אלא להשתמש בdefault package (זה שנוצר אוטומטית ביצירת פרויקט חדש בeclipse). יצירת package חדש ימנע העלאת הקבצים לsubmission system.
7. עליכם להגיש קובץ מסוג zip בשם assignment4.zip המכיל בתוכו:
   a. תיקיית src ובה קבצי הג'אווה של העבודה, ללא המחלקות Main, GUI, Point
   b. מסמך PDF המתאר בקצרה את הפתרון ומסביר בקצרה את זמני הריצה (הפתרון של חלק א').
8. סביבת העבודה בה תיבדקנה העבודות הינה JavaSE-1.6/7
9. עליכם לדאוג כי עבודותיכם יתקמפלו וירוצו בסביבת eclipse תחת גרסאות Java הנזכרות לעיל.
10. עבודות שלא יתקמפלו – יקבלו ציון 0.
11. עבודותיכם יבדקו באמצעות כלי בדיקה אוטומטים הבודקים קורלציה בין עבודות. אין להעתיק! להזכירכם, המחלקה רואה בחומרה רבה העתקות.
12. נרצה לראות קוד מתועד, מתוכנן היטב ויעיל שמייצג הבנה.

בהצלחה!