תרגיל מסכם

תיאור כללי:

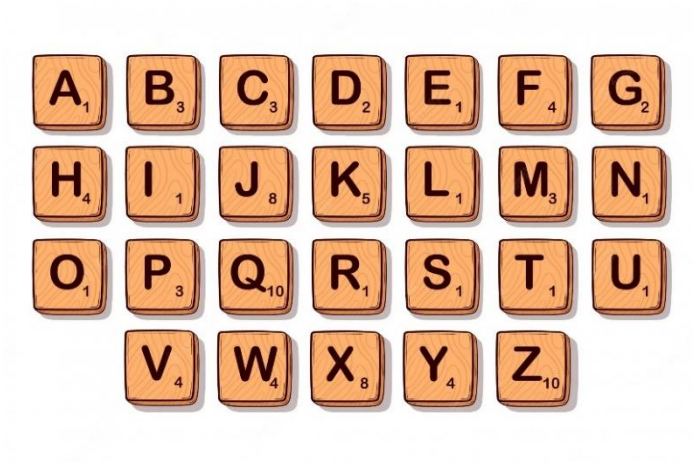
בתרגיל זה נבנה את המשחק Book Scrabble – בדומה למשחק Scrabble ("שבץ נא" בגרסה העברית) השחקנים יצטרכו להרכיב מילים המצטלבות זו עם זו כמו בתשבץ ולצבור נקודות. אולם, המילים החוקיות הן לא כל המילים במילון האנגלי, אלא רק מילים שמופיעות בספרים שנבחרו למשחק.

בתרגיל זה נבנה בשלבים חלקים של שרת גנרי (ללא התקשורת) שיאפשר למשתמש לשחק מול השרת.

הגדרות:

אריח – Tile

- לוח קטן המכיל אות (באנגלית) ואת ערכה במשחק - כמות הנקודות שהאות שווה.
- בתרשים הבא ניתן לראות כמה כל אות שווה במשחק



- הניקוד מבוסס על יחס הפוך לשכיחות האות בשפה האנגלית. כלל שהאות נדירה יותר כך תקבל ניקוד יותר גבוה.

שק – Bag

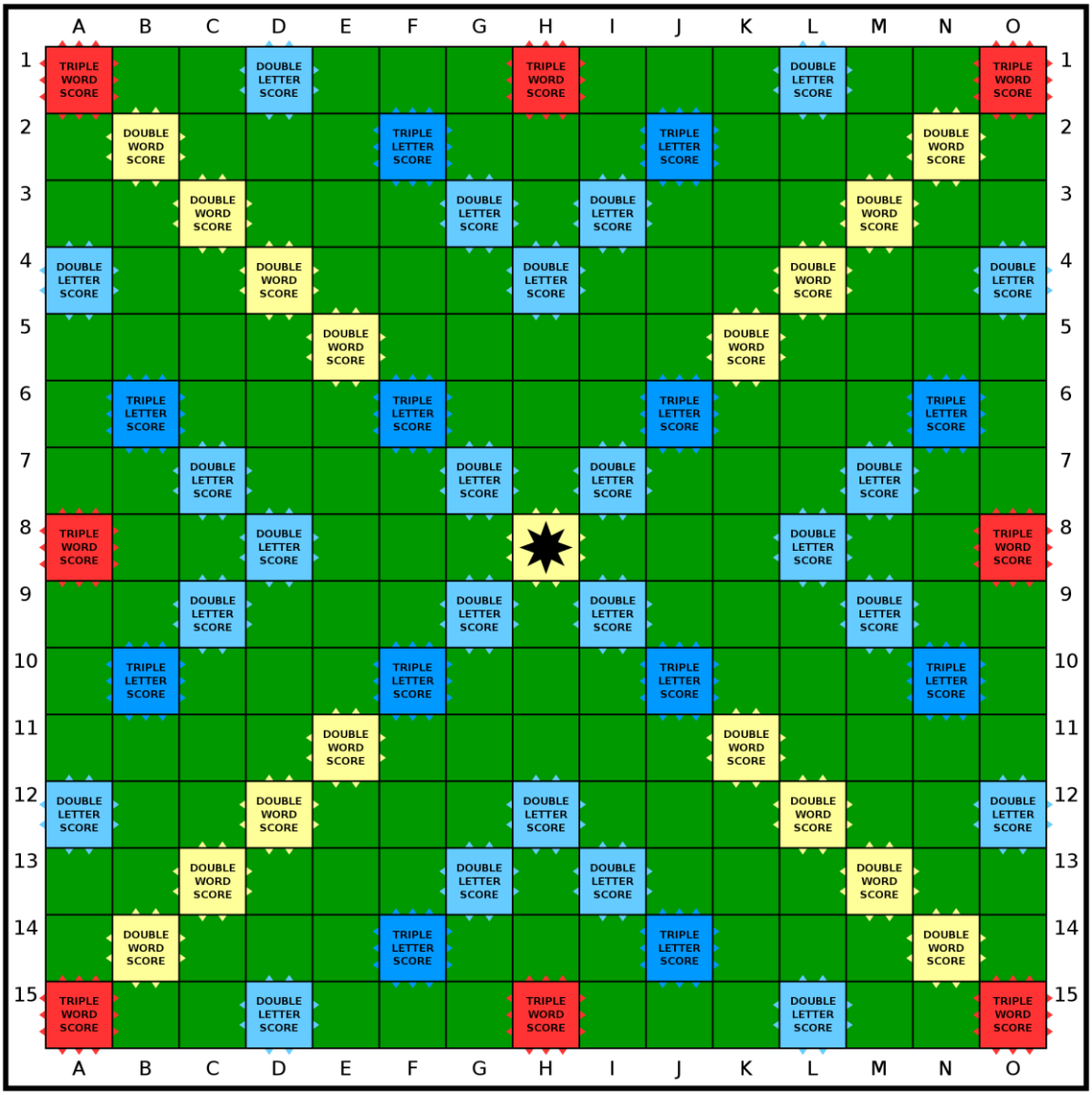
- שק המכיל 98 אריחים *
- מאפשר לשחקנים להוציא באקראי אריחים (כלומר ללא שיראו מה הם מוציאים)
- כמות האריחים בשק לכל אות בתחילת משחק:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
9	2	2	4	12	2	3	2	9	1	1	4	2	6	8	2	1	6	4	6	4	2	2	1	2	1

* במשחק המקורי ישנם גם שני אריחים ריקים אך במשחק שלנו נתעלם מהם

לוח המשחק – Board

- לוח דו-ממדי בגודל 15×15
- ללוח כמה משבצות בונוס:
  - המשבצת המרכזית (מסומנת בכוכב) מכפילה את ערך המילה שכתובה עליה
  - משבצות שמכפילות את ערך האות שנמצאת עליהן (תכלת)
  - משבצות שמשלשות את ערך האות שנמצאת עליהן (כחול)
  - משבצות שמכפילות את ערך המילה כולה (צהוב)
  - משבצות שמשלשות את ערך המילה כולה (אדום)
- משבצות הבונוס מפוזרות כבתרשים הבא:



חוקי ומהלך המשחק

לצורך הפרויקט נגדיר מערכת חוקים מעט פשוטה יותר מהמשחק המקורי:

1. כל שחקן שולף באקראי אריח מתוך השק
2. סדר השחקנים נקבע ע"פ סדר האותיות שנשלפו (מהקטן לגדול)
   a. אם נשלף אריח ריק נחזיר אותו לשק ונשלוף אחר.
3. כל האריחים חוזרים לשק
4. כל שחקן שולף באקראי 7 אריחים
5. השחקן הראשון (זה שהוציא את האות הקטנה ביותר בהגרלה) צריך להרכיב מילה חוקית שעוברת דרך המשבצת המרכזית (הכוכב) בלוח.
   a. אך ורק הוא מקבל עבורה ניקוד כפול.
   b. הוא משלים מהשק כך שיהיו לו שוב 7 אריחים.
6. בהדרגה, כל שחקן בתורו, מרכיב מילה חוקית מהאריחים שברשותו
   a. כאשר כמו בתשבץ, כל מילה חייבת להישען על אחד האריחים שקיימים על הלוח.
   b. לאחר כתיבת המילה השחקן משלים מהשק ל 7 אריחים
   c. הניקוד שלו מצטבר בהתאם לכל המילים שנוצרו על הלוח בעקבות השמת האריחים
      i. אריחים שמונחים על משבצות כפל או שילוש **אות**, יוכפל או ישולש ערכם בהתאמה
      ii. לאחר מכן המילה מקבלת את סכום ערך האריחים שלה
      iii. סכום זה יוכפל או ישולש עבור כל משבצת כפל או שילוש **מילה** שאחד האריחים מונחים עליה (כלומר, תיתכן למשל הכפלה ב 4 או 9 אם המילה תפסה שתי משבצות כפל מילה או שילוש מילה בהתאמה)
      iv. החישוב לעיל נכון לכל **מילה חדשה** שנוצרה על הלוח בעקבות ההשמה בתור
7. שחקן שאינו יכול להרכיב מילה חוקית מוותר על התור שלו.
8. המשחק יסתיים לאחר *N* סבבים.

מילה חוקית חייבת לעמוד בכל התנאים הבאים:

- כתובה משמאל לימין או מלמעלה למטה (ולא באף צורה אחרת)
- מילה שמופיעה באחד הספרים שנבחרו למשחק
- נשענת על אחד האריחים הקיימים על הלוח
- לא מייצרת על הלוח מילים אחרות שאינן חוקיות

דוגמת משחק:

נניח לצורך הדוגמה שיש 2 שחקנים והאות *R* יושבת על הכוכב.

- שחקן א כתב את המילה *Horn* השווה 7 נק', אך קיבל ניקוד כפול (בונוס כוכב) והשלים מהשק 4
- שחקן ב כתב *Farm* השווה 9 נקודות והשלים מהשק 3
- שחקן א כתב את המילה *Paste* השווה לבדה 7 נק', אולם
  - האותיות *P* ו *E* נפלו על משבצת "אות משולשת" ולכן המילה שווה 15 נק'

- בנוסף נוצרה מילה *Farms* השווה 10 נק' ולכן הוא קיבל בסך הכל 25 נקודות
- שחקן ב' כתב את המילה *Mob* (והשלים מהשק 2)
  - חוץ ממנה נוצרו על הלוח גם המילים *Not* ו *Be*
  - ויחד עם משבצות הבונוס בלוח הוא קיבל 18 נק' בסך הכל
- שחקן א' כתב *Bit* ובאופן דומה קיבל ניקוד מצטבר של 22 נק' עובר *Bit, Pi* ו *At*.

אבן דרך 1

ממשו את הטיפוסים הבאים ע"י שימוש בתבנית flyweight

המחלקה Tile (אריח)

- נרצה שאובייקטים מסוג המחלקה יהיו immutable – כלומר לא ניתנים לשינוי.
  - נשיג תוצאה זו ע"י כך שהשדות שלה יוגדרו כ final.
  - הבנאי יצטרך לאתחל משתנים אלו
- תגדירו את השדות char letter עבור אות, ואת ה int score עבור הניקוד.
  - מכיוון שהם final, אין לנו בעיה שיוגדרו כ public
- הוסיפו אוטומטית באמצעות ה IDE שלכם
  - בנאי שמאתחל את השדות הללו, equals, ו hashCode

לא נרצה שכל מי שירצה יוכל לייצר אריחים. אנו רוצים לשלוט בכמויות שלהם לטובות המשחק. לכן ההרשאה של הבנאי תהיה private!

אולם, נממש מחלקה פומבית וסטטית בשם Bag (שק) **בתוך** המחלקה Tile, וכך תהיה מחלקה זו היחידה עם האפשרות ליצור אריחים. היא המחלקה שתנהל את ה flyweight.

המחלקה Bag (שק)

- תחזיק מערך של 26 int-ים המייצגים את הכמות של כל אות ע"פ הגדרות המשחק.
  - למשל תא 0 מייצג A ובו הערך 9 שמייצג שקיימים 9 אריחים מסוג A
  - בתא 1 שמייצג B יהיה 2 וכך הלאה... בתא 25 המייצג Z יהיה 1.
- תחזיק מערך של 26 אריחים, מסודרים לפי ה ABC
  - כל אריח עם האות והערך שלה ע"פ הגדרות המשחק (כל האותיות ב capital)
  - למעשה, אין לנו צורך בעוד אובייקטים מסוג Tile מלבד אלה המוגדרים במערך.
- המתודה ()getRand תחזיר אריח אקראי מתוך השק
  - היא מחזירה למעשה (by value) reference לאחד התאים במערך האריחים.
  - היא מורידה את הכמות המתאימה ממערך הכמויות
  - כמובן לא ניתן לקבל אריח כלשהו אם הכמות שלו ירדה ל 0.
  - אם השק ריק היא פשוט תחזיר null
- המתודה ()getTile תפעל באופן דומה ל getRand פרט לכך שהיא תקבל char ותוציא אריח שזו האות שלו מהשק אם ניתן, אחרת תחזיר null.
- המתודה ()put בהינתן אריח היא "תחזיר אותו לשק"
  - למעשה רק צריך לעדכן את הכמות.
  - בכל מקרה, מתודה זו לא תאפשר הכנסה מעבר לכמות המוגדרת בחוקי המשחק
- המתודה size תחזיר כ int את כמות האריחים שבתוך השק.
- לצורך הבדיקה, המתודה getQuantities תחזיר **העתק** של מערך הכמויות

המחלקה Word

מחלקה זו מייצגת השמה אפשרית של מילה על לוח המשחק. נגדיר את השדות הבאים:

- tiles – מערך של האריחים המרכיבים את המילה
- row, col – המגדירים את מיקום (שורה, עמודה) האריח הראשון במילה על לוח המשחק
- vertical – בוליאני המייצג האם המילה כתובה בצורה אנכית (מלמעלה למטה). אם הוא 'שקר' אז המילה כתובה בצורה אופקית (משמאל לימין)

בנאי המחלקה יאתחל את כל השדות ע"פ הסדר לעיל. לכל שדה יהיה getter משלו. בנוסף, נצטרך את מתודת ה equals. תוכלו לכתוב את הכל בצורה אוטומטית באמצעות ה IDE שלכם.

המחלקה Board

- מחלקה זו מחזיקה את לוח המשחק (בחרו לבדכם כיצד)
- המתודה ()getTiles תחזיר מערך דו-ממדי של אריחים בהתאם למצב הלוח.
  - היכן שאין אריח על הלוח יהיה פשוט null.
  - שימו לב! האריחים הם immutable אך המערך לא. מישהו יוכל להוסיף לו אריחים שלא דרך Board ולכן גם כאן נרצה להחזיר העתק של המערך.
    - וזה לא נורא כי בסוף מדובר רק במצביעים.

במתודות הבאות מתייחסות להשמה של מילה אפשרית על הלוח. תשימו לב איך במקום מתודה אחת של placeWord שהיתה צריכה לבצע את ההשמה בפועל ולבדוק שהמילה חוקית על הלוח וע"פ המילון ולחשב את הניקוד לכל מילה שנוצרה וכו', אנו מפרקים את זה לכמה מתודות שונות ע"פ עיקרון ה **Single Responsibility**.

- המתודה ()boardLegal תקבל מופע של Word ותחזיר 'אמת' אם:
  - כל המילה נמצאת בתוך הלוח
  - נשענה על אחד האריחים הקיימים על הלוח כבתשבץ (אריח צמוד או חופף)
    - ההשמה הראשונה כזכור נשענת על משבצת הכוכב
  - לא דרשה החלפה של אריחים קיימים.

אחרת היא תחזיר 'שקר'.

למשל מתוך הדוגמה לעיל, בתור הראשון (HORN) ראינו שכל המילה נכנסה בתוך הלוח, ושאכן אחד האריחים נשען על הכוכב.

עבור ההשמה של FARM נוודא בנוסף שאחד האריחים צמוד או חופף לאחד האריחים הקיימים על הלוח. האריח R מספק דרישה זו. כמו כן, נצטרך לוודא שה R זהה ל R שהיתה קיימת כבר על הלוח בהשמה של HORN כך שהמילה HORN לא הוחלפה.

- המתודה ()dictionaryLegal תבדוק האם המילה חוקית מבחינת מילון המשחק (מילים המופיעות בספרים שנבחרו). לעת עתה היא תמיד תחזיר true.
- המתודה ()getWords – בהינתן Word היא תחזיר לנו מערך של כל המילים החדשות שיווצרו על הלוח כולל אותה המילה, לו היתה השמה כזו על הלוח. דוגמאות:
  - עבור PASTE בתור 3 לעיל, יוחזר מערך שמכיל את PASTE ואת FARMS.
  - עבור MOB תור 4 לעיל, יוחזר מערך שמכיל את MOB, NOT, BE.
  - סדר המילים במערך לא משנה
  - במקום מערך פרמיטיבי []Word תחזירו הפעם אובייקט מסוג ArrayList<Word>
    - אובייקט זה מאפשר למערך שהוא מחזיק לגדול באופן דינמי.
- המתודה ()getScore בהינתן Word היא תחשב את הניקוד הכולל של המילה, כולל כל משבצות הבונוס שעליהן היא מונחת.

שימו לב! עד כה אף מתודה לא מבצעת השמה על הלוח בפועל. אלו היו מתודות עזר.

כעת בהינתן מילה אפשרית להשמה, נוכל לבדוק באמצעות המתודות לעיל, האם היא חוקית מבחינת הלוח, אם כן אז לדרוש את כל המילים החדשות שהיו נוצרות מההשמה האפשרית של המילה, ועבור כל מילה כזו לבדוק האם היא חוקית מבחינת מילון המשחק. אם כל המילים אכן חוקיות אז נוכל סוף סוף לבצע את ההשמה בפועל על הלוח ולכן נחזיר את הניקוד המצטבר לכל מילה חדשה שנוצרה. בכל מקרה אחר, לא תבוצע השמה ונחזיר ניקוד 0.

בדיוק את זה עליכם לממש במתודה ()tryPlaceWord אשר בהינתן Word היא תחזיר ניקוד מתאים.

<u>שימו לב</u> שההשמה מכילה רק את האריחים **החדשים** שיש להניח על הלוח, ואילו הבדיקות השונות מכילות את כל המילה. לדוגמה כאשר ביצענו השמה ל FARM בתור השני, הנחנו רק FA_M על הלוח (במקום ה R יש אריח null) אך כל הבדיקות השונות לפני ההשמה על הלוח בדקו את המילה FARM במלואה.

כעת, תארו לכם לרגע, שאת כל החוקים האלה היינו מממשים במתודה אחת.

האם היא היתה קריאה? האם היא בכלל היתה טסטבילית (ניתנת לבדיקה)? כיצד הייתם מדבגים אותה?

ואיך הקוד היה נראה ללא המחלקה Word?

אז גם עבור המתודות לעיל, מאד רצוי להשתמש במתודות עזר פרטיות!

אבן דרך 2

הקדמה

באבן דרך זו נרצה לממש את הלוגיקה של חיפוש המילים במילון הספרים. נרצה לדאוג שהפתרון שלנו סקאלבילי (scalable) – כלומר גם כאשר מספר הספרים ו\או מס' הלקוחות המבקשים שירות באותו הזמן ילך ויגדל, הפתרון שלנו עדיין יעבוד בצורה יעילה ללא גידול משמעותי במשאבים; הגידול במשאבים צריך להיות ליניארי ביחס לגודל הבעיה שאותה אנו מנסים לפתור.

הספרים נתונים כקובצי טקסט. תארו לכם שעל כל שאילתה לגבי קיומה של מילה כלשהי נצטרך לחפש אותה בכל הקבצים. זה ידרוש המון פעולות של I/O ולכן לא סקאלבילי.

תארו לכם שנשמור את כל המילים ב HashSet<String>. עדיף לחפש בזיכרון (RAM) מאשר בדיסק, אולם, מהר מאד (יחד עם גידול הבעיה) עלול להיגמר לנו המקום. המחשב יכנס לתהליך של trashing (החלפה של דפים בין ה RAM לדיסק) ושוב הביצועים ירדו עד לקריסה אפשרית של השרת.

לכן המילון שלנו ינקוט במספר מסננים:

1. Cache Manager – שיחזיק בזיכרון את התשובות לשאילתות הנפוצות ביותר. החיפוש בו יהיה ב O(1) זמן וגודלו יהיה קבוע ע"פ פרמטר שנגדיר. כך, בהינתן שאילתה נבדוק בזריזות מהי התשובה. אם התשובה קיימת אז נחזיר אותה. אחרת, נעביר את השאלה למסנן הבא.
2. Bloom Filter – אלגוריתם יעיל וחסכוני מאד במקום, שיודע לומר בוודאות מוחלטת האם מילה לא נמצאת במילון הספרים, ובהסתברות גבוהה כרצוננו האם מילה כן נמצאת.
3. אם בכל זאת המשתמש בוחר לאתגר את המילון, במחשבה שהמילון טעה והמילה דווקא לא נמצאת, אז יתבצע חיפוש מבוסס I/O. חוקי המשחק שלנו יקנסו בנקודות את המאתגר אם הוא הטריח את השרת לחינם, או שיתנו לו בונוס אם הוא צדק.

בכל מקרה, כאשר חוזרת תשובה נעדכן את ה cache manager כדי לחסוך חיפושים מיותרים.

Cache Manager

בקובץ CacheManager.java עליכם לממש את המחלקה CacheManager.

בהמשך, נרצה לייצר שני מופעים של CacheManager עבור המשחק: אחד עבור השאילתות למילים שאכן נמצאות בספרים, והשני עבור השאליתות למילים שאינן נמצאות בספרים. כך נוכל לשאול כל אחד מהם האם המילה נמצאת אצלו ולהחזיר תשובה מתאימה. אם המילה לא נמצאת אצל אף אחד מהם, אז נוכל להעביר את השאלה למסנן הבא ובהתאם לתשובתו לעדכן את ה CacheManager המתאים.

כאשר נרצה להוסיף מילה ל CacheManager נרצה לוודא שלא נעבור את מגבלת הזיכרון שהוגדרה לו, ואם כן לבחור מילה כקורבן ולהוציא אותה מה cache. אולם, בחירה זו היא פונקציונליות שצריכה להינתן ל CacheManager כפרמטר, ולא ע"י מימוש קבוע בתוך המחלקה. למשל, אולי נרצה שאת ה CacheManager של המילים שנמצאות בספרים ננהל לפי אלגוריתם מסוים ואילו את ה CacheManager של המילים שאינן נמצאות בפרים לפי אלגוריתם אחר... לשם כך נצטרך להשתמש בתבנית עיצוב בשם Strategy Pattern.

בתבנית עיצוב זו נחשוף ממשק שמגדיר את הפונקציונאליות הרצויה, ונבקש במחלקה שלנו פרמטר של אובייקט מסוג ממשק זה. כך יוכלו להזין לנו אובייקטים שונים שמימשו את הממשק הזה – כל אחד בדרכו שלו – ואז אנו נוכל להפעיל את הפונקציונליות שהוגדרה בממשק באופן פולימורפי – כלומר מבלי שאיכפת לנו באמת מהמימוש.

ובהקשר שלנו, נגדיר את הממשק CacheReplacementPolicy:

```
public interface CacheReplacementPolicy{
      void add(String word);
      String remove();
}
```

ממשק זה מגדיר את המדיניות של תחלופת המילים ב cache.

- המתודה add מסמלת שניתנה שאילתה עבור המילה word
- המתודה remove תחזיר לנו את המילה שיש להוציא מה cache

כעת ממשו שתי מחלקות המממשות את הממשק לעיל:

- LRU – כאשר ב remove היא מחזירה את המחרוזת המהווה את ה least recently used. כלומר זו ששאלנו עליה לפני הכי הרבה זמן. לדוגמה, אם הכנסתי את "A" ואז את "B" ואז את "C" ואז שוב את "A", ובקשו remove, אז אחזיר את "B".
- LFU – כאשר ב remove היא מחזירה את המחרוזת המהווה את ה least frequently used. כלומר את זו שבקשו הכי מעט פעמים. במקרה של שוויון נחזיר את זו שנכנסה קודם. לדוגמה אם הבקשות היו (משמאל לימין) A,B,B,A,B,C ובקשו remove אז נחזיר את C שכן אותו בקשו רק פעם אחת.

כדי לממש את האלגוריתמים לעיל תצטרכו לבחור מבני נתונים כלשהם (ואולי אפילו יותר מאחד למחלקה). הקפידו לבחור מבנה נתונים יעיל שמתאים לדרישות האלגוריתם. הקפידו לתחזק את הנתונים בצורה נכונה. למשל, אם הוספתם מילה ב add תצטרכו להסיר אותה ב remove, או למשל אם שיניתם איבר שנמצא בתוך תור עדיפויות, הקפידו להוציאו ולהכניסו מחדש. ובכל מקרה, גודל מבני הנתונים הללו צריך להיות מוגבל לגודלו של ה CacheManager שעושה בהם שימוש.

כעת נוכל לממש בקלות את המחלקה של CacheManager:

1. הבנאי יקבל כפרמטר את size הגודל המקסימלי של ה cache (כ int), ואת crp – מופע של CacheReplacementPolicy.
2. נתחזק ב HashSet<String> את המילים שב cache.
3. המתודה query בהינתן מילה פשוט תחזירי לנו בוליאני האם המילה נמצאת ב cache או לא.
4. המתודה add בהינתן מילה, היא תעדכן את ה crp, תוסיף את המילה ל cache, ואם גודלו גדול מהגודל המקסימלי, אז נסיר ממנו את המילה שבחר ה crp.

נשים לב לכמה דברים:

א. הפרדנו בין המתודה query לבין המתודה add. אלו 2 תחומי אחריות שונים. הראשונה בודקת האם המילה נמצאת והשנייה מכניסה אותה ל cache. זכרו שברצוננו ליצור שני מופעים של CacheManager. נניח שהיתה מילה שלא נמצאה אצל שניהם, אז שאלנו את המסנן הבא וגילינו שהמילה אכן נמצאת בספרים. אז כעת נצטרך להכניס אותה רק ל CacheManager הראשון ולא לשני...

ב. שימוש נכון ב Strategy Pattern מאפשר לנו לשמור על כל עקרונות SOLID, ובפרט:

a. להזריק לכל CacheManager איזה מופע של CacheReplacementPolicy שנרצה ולהחליף ביניהם בזמן ריצה (כל עוד שמרנו על העקרון של ליסקוב)

b. להוסיף CacheReplacementPolicy חדשים מבלי לשנות את הקוד של ה CacheManager (פתוח להרחבה, סגור לשינויים)

Bloom Filter

כאמור, אלגוריתם זה הינו אלגוריתם יעיל וחסכוני מאד במקום, שיודע לומר בוודאות מוחלטת האם מילה לא נמצאת במילון הספרים, ובהסתברות גבוהה כרצוננו האם מילה כן נמצאת.

בקצרה, הוא פועל כך:

- האלגוריתם מתחזק מערך של ביטים, למשל בגודל 256 (32 בתים), כלום כבויים בהתחלה.
- האלגוריתם מקבל כפרמטר K פונקציות hash שונות.
- בהינתן מילה, הוא יפעיל עליה את K פונקציות ה hash. כל אחת מחזירה ערך מספרי שונה.
- על כל ערך כזה נבצע מודולו לפי אורך מערך הביטים (למשל מודולו 256) ונקבל אינדקס בודד.
- נדליק במערך את הביטים באינדקסים שחזרו.

בדיקה האם מילה קיימת:

- בהינתן מילה, נריץ עליה את K פונקציות ה Hash ונבצע שוב את החישוב לעיל
- אלא שהפעם נבדוק האם כל אינדקס שחזר מצביע על ביט דולק במערך.
- מספיק שביט אחד כבוי כדי לדעת בוודאות שהמילה לא נמצאת
- אם כל האינדקסים דולקים אז סימן שיש סיכוי שהמילה אכן נמצאת
- אך יש גם סיכוי לטעות (False Positive), כלומר לומר שהמילה נמצאת למרות שהיא לא.
- ככל שמערך הביטים יהיה גדול יותר ונשתמש ביותר פונקציות hash כך הסיכוי לטעות יקטן.

לקריאה נוספת:

https://en.wikipedia.org/wiki/Bloom_filter

לטובתכם כמה ספריות שיעזרו לכם לממש Bloom Filter:

- BitSet מייצגת מערך של ביטים. הוא גדל ע"פ הצורך. ניתן להדליק או לכבות ביט בכל אינדקס שנרצה.
- MessageDigest –
  - מאפשר לכם לאתחל פונקציית hash ע"פ שמה (למשל MD5, SHA1 וכו') . לדוגמה
    - MessageDigest md=MessageDigest.getInstance("MD5")
  - מחזיר לכם מערך של בתים בחישוב פונקציית ה hash על מערך של בתים. לדוגמה
    - byte[] bts=md.digest("hello".getBytes())
    - מערך שכזה נוכל להזין ל:
- BigInteger - מחזיק ערך Integer גדול ככל שנרצה.
  - יש לו בנאי שמרכיב מספר בהינתן מערך בתים.
  - המתודה intValue תחזיר לנו את הערך הזה מגולם בתום int
    - (שימו לב שהוא עלול להיות שלילי)
- כעת תוכלו לקחת את הערך האבסולוטי של ה intValue, לבצע לו מודולו מתאים, ולהדליק ב BitSet את הביט באינדקס שחזר.

עליכם לממש את המחלקה BloomFilter ע"פ הדרישות הבאות:

- הבנאי יקבל את אורך מערך הביטים, ואת שמות האלגוריתמים כרשימת פרמטרים (String...algs)
  - לדוגמה: BloomFilter bf = new BloomFilter(256,"MD5","SHA1");
- המתודה add – בהינתן מחרוזת היא תכניס אותה ל bloom filter. כלומר
  - תפעיל עליה את כל פונקציות ה hash שקיבלנו בבנאי,
  - ותדליק את הביטים הרלוונטים במערך הביטים.

- המתודה contains – בהינתן מחרוזת היא תחזיר בוליאני האם היא נמצאת ב bloom filter.
- המתודה toString (דריסה של Object) תחזיר מחרוזת המורכבת מ {0,1} בהתאם לביטים הדולקים \ כבויים במערך הביטים.
  - מתודה זו תשמש אותנו לבדיקות \ דיבאג.
  - זכרו שגודל ה BitSet גדל באופן דינאמי ע"פ הצורך.

IO Searcher

כזכור, אם ה Cache Manager לא ידע האם מילה כלשהי נמצאת באחד הספרים, אז נשאל את ה Bloom Filter. אולם, יש לו הסתברות מסוימת לטעות ולכן המשתמש יכול בהמשך לאתגר את תשובת השרת. במקרה זה לשרת לא תהיה ברירה אלא לחפש את המילה בקובצי הטקסט של כל אחד מהספרים...

נתון הממשק FileSearcher שמגדיר את המתודות הבאות:

- search אשר בהינתן מילה (Word) ורשימת פרמטרים של שמות קבצים (String...fileNames) היא תחפש בכל הקבצים את המילה הנתונה.
  - ברגע שתמצא אותה היא תעצור את כל החיפושים ותחזיר "אמת".
  - אם נסרקו כל הקבצים ולא נמצאה המילה אז היא תחזיר "שקר".
  - אם קרתה חריגה כלשהי, יש להחזיר "שקר".
- stop אשר ברגע שהופעלה היא מפסיקה את כל החיפושים של המתודה search.

עליכם לממש את המחלקה IOSearcher כסוג של FileSearcher. זכרו שיש לסגור את כל הקבצים הפתוחים.

ParIOSearcher

מחלקה זו תהיה ה Proxy של IOSearcher. גם היא תהווה סוג של FileSearcher באמצעות reuse למופעים של IOSearcher. היא תשתמש ב thread pool מסוג cached thread pool אשר באמצעותו היא תחפש בכל קובץ בת'רד משלו.

- ברגע שהמילה נמצאה באחד הקבצים על כל החיפושים להסתיים מייד בכל הת'רדים.
- על ה thread pool עצמו להיפתח בבנאי, ולבצע shutdown במתודה finalize.
- במתודה stop יש לבצע shut down now ל thread pool.

Dictionary

כעת נתפור את הכל יחד במחלקה Dictionary.

- בבנאי היא תקבל רשימת פרמטרים של שמות קבצים (String...fileNames) המהווים סיפורים
  - הבנאי יצור CacheManager חדש בגודל 400 עם LRU עבור המילים שקיימות
  - הבנאי יצור CacheManager חדש בגודל 100 עם LFU עבור המילים שאינן קיימות
  - הבנאי יצור BloomFilter בגדול 256 עם הפונקציות MD5 ו SHA1.
  - הבנאי יכניס כל מילה מהקבצים ל Bloom Filter
- המתודה query – בהינתן מילה:
  - נחפש ב cache manager של המילים שקיימות, אם נמצא נחזיר "אמת", אחרת
  - נחפש ב cache manager של המילים שאינן קיימות, אם נמצא נחזיר "שקר", אחרת
  - נחפש ב BloomFilter ונחזיר את התשובה שלו, לאחר שנעדכן את ה Cache Manager המתאים בתשובה.
- המתודה challenge – בהינתן מילה היא תפעיל את ה ParIOSearch ותחזיר את תשובתו. כמובן יש לעדכן את ה CacheManager המתאים בתשובה.
- המתודה close תקרא ל stop של ה ParIOSearcher.

כעת כל שנותר הוא לעדכן את המתודה ()dictionaryLegal במחלקה Board כך שבהינתן מופע של Word ומופע של dictionary היא תחזיר האם המילה חוקית או לא ע"פ המילון.

אין צורך להגיש את Board באבן דרך זו.

ייתכן ואת המשחק המלא נממש בקורס תכנות מתקדם 😊

נתראה שם.

בהצלחה!