פרויקט משחק Book Scrabble (צד שרת)

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...

האחראית על כך תהיה המחלקה IOSearcher. עליכם לממש את המתודה הסטטית search אשר בהינתן מילה, ורשימת פרמטרים של שמות קבצים (String...fileNames) היא תחפש בכל הקבצים את המילה הנתונה. ברגע שתמצא אותה היא תחזיר אמת. ואם נסרקו כל הקבצים ולא נמצאה המילה אז היא תחזיר שקר. על המתודה תזרוק חריגה אם היתה כזו בעת החיפוש. זכרו שיש לסגור את כל הקבצים הפתוחים.

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 IOSearch ותחזיר את תשובתו. כמובן יש לעדכן את ה CacheManager המתאים בתשובה. אם קרתה חריגה יש להחזיר "שקר".

פרטי ההגשה נמצאים במודול.

בהצלחה!