Project 2 Documentation

Yoav Malichi - yoavmalichi - 208392290
Eviatar Shemesh - eviatars - 322623182

ModHash

Fields

- *private long a* - coefficient of ‘x’ in the polynom ax+b.
- *private long b* - free coefficient := b in the polynom ax+b.
- *private long p* - prime.
- *private int m* - table size.

Methods

- *private ModHash(int m, long p, long a, long b)* - The constructor which GetFunc calls to return an instance of ModHash (initializes a, b, p, m fields respectively to the inputs).

```
private ModHash(int m, long p, long a, long b) {
    this.m = m;
    this.p = p;
    this.a = a;
    this.b = b;
}
```

- *public static ModHash GetFunc(int m, long p)* - Randomly generates a function from the universal family and returning it as a new ModHash.

```
public static ModHash GetFunc(int m, long p){
    long a = ThreadLocalRandom.current().nextLong( origin: 1, p);
    long b = ThreadLocalRandom.current().nextLong( origin: 0, p);
    return new ModHash(m, p, a, b);
}
```

- *public int Hash(long key)* - Calculates and returns the Hash value, according to the fields a, b, m, p as we've learned.

```
public int Hash(long key) {
    return (int)(((this.a * key + this.b) % p) % m);
}
```

LPHashTable

Fields

- *private ModHash h* - Saving the ModHash randomized function for this LPHashTable.

Methods

- *public LPHashTable(int m, long p)* - The constructor of LPHashTable, basically just calling super() and initializing the ModHash h field.

```
public LPHashTable(int m, long p) {
    super(m);
    this.h = ModHash.GetFunc(m,p);
}
```

- *public int Hash(long x, int i)* - Return the calculation of hash on current object(x) and index(i) according to the ModHash function of this LPHashTable and LP formula.

```
@Override
public int Hash(long x, int i)
{
    return (int)(((long)this.h.Hash(x)+(long)i)%this.m);
}
```

QPHashTable

Fields

- *private ModHash h* - Saving the ModHash randomized function for this QPHashTable.

Methods

- *public QPHashTable(int m, long p)* - The constructor of QPHashTable, basically just calling super() and initializing the ModHash h field.

```
public QPHashTable(int m, long p) {
    super(m);
    this.h = ModHash.GetFunc(m, p);
}
```

- *public int Hash(long x, int i)* - Return the calculation of hash on current object(x) and index(i) according to the ModHash function of this QPHashTable and QP formula.

```
@Override
public int Hash(long x, int i) {
    return (int)(((long)(this.h.Hash(x)) + ((long)i * (long)i)) % this.m);
}
```

AQPHashTable

Fields

- *private ModHash h* - Saving the ModHash randomized function for this AQPHashTable.

Methods

- *public AQPHashTable(int m, long p)* - The constructor of AQPHashTable, basically just calling super() and initializing the ModHash h field.

```
public AQPHashTable(int m, long p) {
    super(m);
    this.h = ModHash.GetFunc(m,p);
}
```

- *public int Hash(long x, int i)* - Return the calculation of hash on current object(x) and index(i) according to the ModHash function of this AQPHashTable and AQP formula.

```
@Override
public int Hash(long x, int i) {
    if (i % 2 == 0) {
        return (int)((((long)this.h.Hash(x)+(long)Math.pow(i,2))%this.m));
    }
    else {
        return (int)((((long)this.h.Hash(x)-(long)Math.pow(i,2))%this.m + (long)this.m)%this.m);
    }

}
```

DoubleHashTable

Fields

- *private ModHash h1* - Saving the First ModHash randomized function for this DoubleHashTable.
- *private ModHash h2* - Saving the Second ModHash randomized function for this DoubleHashTable.

Methods

- *public DoubleHashTable(int m, long p)* - The constructor of DoubleHashTable, basically just calling super() and initializing both ModHash h1,h2 fields.

```
public DoubleHashTable(int m, long p) {
    super(m);
    this.h1 = ModHash.GetFunc(m, p);
    this.h2 = ModHash.GetFunc( m: m - 1, p);
}
```

- *public int Hash(long x, int i)* - Return the calculation of hash on current object(x) and index(i) according to the ModHash h1, h2 functions of this DoubleHashTable and DoubleHashTable formula (second hash function output is similar to what we've seen in the recitation [1, m-1]).

```
@Override
public int Hash(long x, int i) {
    return (int)(((long)this.h1.Hash(x) + ((long)i * ((long)this.h2.Hash(x) + 1))) % this.m);
}
```

OAHashTable

Fields

- *private final HashTableElement DELETED_HTE* - Since keys are non negative, we can keep only one DELETED_HTE per HashTable that will be used to indicate a deleted cell (that is ok to insert in, but not to stop the find process).
- *private HashTableElement[] table* - The hash table itself.
- *protected int m - The table size (for easiness of accessing and similarity to formulas).*

Methods

- *public OAHashTable(int m)* - The constructor of OAHashTable, basically just initializing a new hash table of size m, this.m to be m and this.DELTED_HTE to be an HashTableElement with a negative key (-1).

```
public OAHashTable(int m) {
    this.table = new HashTableElement[m];
    this.m = m;
    this.DELETED_HTE = new HashTableElement(-1, -1);
}
```

- *public HashTableElement Find(long key)* - As shown in class, we go through the probing sequence of the current HashTable implementation. We stop our Find process when the earlier from the following occurs:
    - Reach an EMPTY cell that is not a deleted one, i.e. the key doesn't exist in the table - return null.
    - Reach a cell that contains an HTE such that its key equals key (the input), i.e. we find the key in the table - return that HTE.
    - Went through the entire probing sequence and none of the above happened, i.e. the key isn't in the table - return null.

```java
@Override
public HashTableElement Find(long key) {
    int i = 0;
    HashTableElement hte;
    while (i < this.m) {
        int j = this.Hash(key, i);
        hte = this.table[j];
        if (hte == null) {
            return null;
        }
        else {
            if (hte.GetKey() == key) {
                return hte;
            }
        }
        i += 1;
    }
    return null;
}
```

- *public void Insert(HashTableElement hte)* - As shown in recitation. If a HashTableElement with the same key as hte is currently inside the table, we throw a KeyAlreadyExistException(hte) exception. Otherwise, iterate i through the size of the array according to the Hash function, until a deleted/empty cell was found.

In this location, insert hte.
If the table is full and we couldn't insert it, throw a TableIsFullException(hte) exception.

```
public void Insert(HashTableElement hte) throws TableIsFullException,KeyAlreadyExistsException {
    long key = hte.GetKey();
    if (Find(key) != null) {
        throw new KeyAlreadyExistsException(hte);
    }
    int i = 0;
    HashTableElement hteCurr;
    while(i < this.m)
    {
        int j = this.Hash(key, i);
        hteCurr = this.table[j];
        if (hteCurr == null || hteCurr.GetKey() == -1)
        {
            this.table[j] = hte;
            return;
        }
        i += 1;
    }
    throw new TableIsFullException(hte);
}
```

- *public void Delete(long key)*
  Iterate through the array using i. j is the calculated index according to (key, i) calculation of Hash Function.
  If the current cell is null, the key doesn't exist in the HashTable and we throw an appropriate Exception.
  Otherwise, if the searched key is in the current cell, we put there a DELETED_HTE element to mark it as a deleted cell.
  This will help us to implement insert, because we can insert a new HashTableElement instead of a deleted element.

```java
public void Delete(long key) throws KeyDoesntExistException {
    int i = 0;
    HashTableElement hte;
    while (i < this.m) {
        int j = this.Hash(key, i);
        hte = this.table[j];
        if (hte == null) {
            throw new KeyDoesntExistException(key);
        }
        else {
            if (hte.GetKey() == key) {
                this.table[j] = this.DELETED_HTE;
                return;
            }
        }
        i += 1;
    }
    throw new KeyDoesntExistException(key);
}
```

חלק ניסויי

שאלה 3

1. עבור Q1 - גודל הקבוצה הוא 3286.
   עבור Q2 - גודל הקבוצה הוא 6571.

2. עבור QPHashTable:
   נזרקה השגיאה IHashTable$TableIsFullException ב74 ניסויים מתוך 100.

   עבור AQPHashTable:
   לא נזרקו שגיאות.

ניתן להסביר תוצאות אלו מסעיף 1, שכן נובע ממנו שבבואנו להכניס מפתח לטבלה, אנו למעשה נקבל Probing Sequence שמכיל 3286 אינדקסים שונים עבור QPHashTable, ואילו בAQPHashTable, נקבל Probing Sequence שמכיל 6571 איברים - כלומר פרמוטציה כלשהי של 0-6570.

במקרה של AQPHashTable, מובטח לנו כי במידה ונכניס כמות איברים שקטנה או שווה לאורך הטבלה נצליח להכניס את כולם ללא שגיאות מכך שנקבל לכל מפתח Probing Sequence שהוא פרמוטציה של אינדקסי הטבלה.

במקרה של QPHashTable, מובטח לנו כי במידה ונכניס כמות איברים שהיא לכל היותר 3286, נצליח להכניס את כולם ללא שגיאות מכך שנקבל לכל מפתח Probing Sequence עם 3286 אינדקסים שונים.
כאשר ה-Load Factor יהיה גדול מחצי, כלומר בבואנו להכניס את האיבר ה-3287 ואילך, ה-Probing Sequence שלנו יכיל כ-3286 אינדקסים שונים, שכאמור יכולים להיות איזשהם 3286 אינדקסים שבהם כבר קיימים איברים, ולכן:

- נזרקו חריגים של TableIsFullException במרבית הניסויים מתוך 100 (בהם מתישהו קיבלנו Probing Sequence בגודל 3286 אשר נופל על 3286 אינדקסים בהם יש כבר איברים).
- ככל שכן נצליח להכניס כל איבר מעבר ל3286 ללא שגיאה ונתקרב להכנסת כמות איברים כגודל הטבלה, כמות האינדקסים התפוסים גדלה והסיכוי להצליח להכניס את האיבר הולך וקטן שכן גדל הסיכוי

שה-Probing Sequence באורך 3286 יהיה כולו על אינדקסים תפוסים כבר (למעשה סיכוי של בערך חצי בבואנו להכניס את האיברים האחרונים בטבלה).

3. נסביר את התופעה המתרחשת בהתייחס לשאריות ריבועיות.

לכל מספר ראשוני יש כ-$\frac{p-1}{2}$ שאריות ריבועיות למעשה +1 אם כוללים את 0.

לכן, לפי הגדרת השארית הריבועית, מספר q נחשב שארית ריבועית של מספר p אם קיים מספר טבעי x כך ש $x^2 = q\ (mod p)$ ולכן מאחר ובשאלה שלנו $p = 6571$ נובע כי קבוצת השאריות הריבועיות היא בגודל $\frac{p-1}{2} = \frac{6571-1}{2} + 1 = 3285 + 1 = 3286$. שזהו המקרה של QPHashTable, ולכן קיבלנו את גודל הקבוצה שקיבלנו בסעיף 1.

עבור AQPHashTable, הסיבה שאכן מתקבלת פרמוטציה טמונה בשאריות הריבועיות ובשקילות המספר הראשוני ל3 מודולו 4.

מתקיים שעבור מספר ראשוני ששקול ל3 מודולו 4, בהינתן שארית ריבועית, אזי המינוס שלה אינה שארית ריבועית, ולכן הAlternating Signs למעשה מאפשרים לנו לקבל ערכים שאינם שאריות ריבועיות.
גודל קבוצת הלא שאריות ריבועיות היא זהה לגודל השאריות הריבועיות, במקרה שלנו 3285 (ללא 0). 3286 איברים נותנים לנו שאריות ריבועיות ו3285 איברים (השליליים) נותנים לנו איברים שאינם שאריות ריבועיות, ובסך הכל מתקבלת הפרמוטציה (6571 איברים בקבוצה).

בנוסף, מתקיים שעבור מספר ראשוני ששקול ל-1 מודולו 4, בהינתן שארית ריבועית אזי המינוס שלה גם שארית ריבועית ולכן ה-Alternating Signs במקרה זה עבור ראשוני ששקול ל-1 מודולו 4, לא תכיל פרמוטציה.

כלומר, התופעה שקיבלנו אינה מתקיימת לכל ראשוני, ובזכות זה ש6571 הוא ראשוני ששקול ל-3 מודולו 4 זה אכן מתקיים עבורו.
הסברנו את התופעה המתרחשת בתרגיל באמצעות שאריות ריבועיות.

שאלה 4

1. תוצאות הניסוי (זמן יצירת איברי הסדרה לא כלול בתוצאות):

Class	Running Time(Seconds)
LPHashTable	1.234
QPHashTable	1.201
AQPHashTable	1.260
DoubleHashTable	1.553

עשינו 100 הרצות על כל שיטת דגימה, ולקחנו את הממוצע על פני ההרצות הללו, מה שמהווה מדגם יחסית נרחב ומקטין את המקריות של סדרת ההכנסות.

הסבר התוצאות:
ראשית, בהכנסת מחצית מהאיברים לטבלה, זמן החישוב של QP הוא המהיר ביותר מבין שיטות הדגימה. אפשר להסביר זאת על ידי כך שהחישוב של הפונקציה עצמה הוא לא מאוד מסובך, (כפל של שני מספרים וחיבור למספר אחר, מודולו P), ולא נוצרים בלוקים רציפים של תאים תפוסים באותה מידה כמו ב LP, מה שקראנו לו בתרגול בעיית ההצטברות המשנית של QP.
כתוצאה מניסויי עזר שביצענו, מצאנו שההפרש הממוצע בין אינדקסים בסדרת החיפוש הוא בערך 5, כלומר אנו מסיקים שבעיית ה Cache Misses זניחה במימוש זה.

לאחריו, בהכנסת מחצית מהאיברים לטבלה, מגיע ה -LP, שהחישוב עליו הוא הכי פשוט (חיבור שני מספרים והפעלת מודולו P), אך נוצרים בלוקים רציפים של תאים תפוסים בזיכרון. אבל, ה Load Factor שלנו קטן שווה ל 0.5 לאורך כל סדרת ההכנסות, ולכן ההשפעה של הבלוקים הרציפים לא תהיה משמעותית כמו במקרים בהם ה Load Factor גדול יותר ובסיכוי גבוה יותר נתקל בבלוק רציף בזיכרון, שישפיע רבות על זמן הריצה (נעבור אחד אחד על האיברים בבלוק, מהגדרת LP), לכן לא מושפע באופן מהותי בסעיף זה (אך ישפיע בסעיף ב' בצורה משמעותית).
בתרגול קראנו לסוגייה זו "בעיית ההצטברות הראשונית".

שיטת הדגימה הבאה בסדר זמני הריצה, בהכנסת מחצית מהאיברים בטבלה היא ה - AQP. ניתן להסביר זאת על ידי כך שזמן החישוב שלה לוקח זמן גדול יותר מקודמיו (חישוב הסימן, העלה בחזקה, הפעלת מודולו פעמיים כאשר i אי זוגי, ופעם אחת כאשר i זוגי). בנוסף, כמות סדרות הבדיקות שלה היא m, בדומה לקודמותיה, ולכן

הגיוני שזמן הריצה שלה יהיה גדול מהן כי אופן החישוב שלה מורכב יותר. בנוסף, מניסוי עזר שערכנו המרחק הממוצע בין שני אינדקסים בסדרת הבדיקות הוא 10, גדול מבין השיטות הקודמות, אך זניח ביחס לאורך הרשימה.

לאחר מכן ועם זמן הריצה הארוך ביותר בהכנסת מחצית מהאיברים לטבלה, יגיע ה DH, שאמנם יש לה $m(m-1)$ סדרות של בדיקות, לעומת m שיש לכל שאר שיטות הדגימה (מה שיעזור לשיטה זו בניסוי בסעיף 2), אך החישוב שלה הוא הכי מורכב מבין שאר השיטות, מה שמגדיל את זמן הריצה (הפעלת 2 פונקציות האש, כפל של מספרים וחיבורם, ולאחר מכן מודולו P). בנוסף לזמן החישוב המורכב, ב DH יש הרבה Cache Misses באופן משמעותי לעומת שאר שיטות הדגימה (בניסוי העזר, המרחק הממוצע בין שני איברים בסדרת הבדיקות הוא בסדר גודל של 3,000,000), מה שמגדיל את זמן הריצה שלה, ומסתדר עם תוצאות הניסוי.

2. <u>תוצאות הניסוי (זמן יצירת איברי הסדרה לא כלול בתוצאות):</u>

Class	Running Time(Seconds)
LPHashTable	15.695
AQPHashTable	6.545
DoubleHashTable	8.036

<u>הסבר התוצאות (ממוצע על פני 100 ניסויים):</u>

לא נבצע ניסוי זה עבור QPHashTable מהסיבות המתוארות בשאלה 3, שכן עלול להיזרק חריג TableIsFullException. מיותר שנחזור על ההסבר המלא והמפורט שכן הוא כתוב במלואו תוך התייחסות לנושא השאריות הריבועיות בשאלה 3.

ראשית, אנו שמים לב כי להכניס את המחצית הראשונה של האיברים לוקח פחות זמן מלהכניס פחות מהחצי השני של האיברים. דבר שמתיישב עם הציפיות מאחר וראינו בהרצאה שזמן הריצה גדל ככל שהLoad Factor גדל.

שנית, אכן ניתן לראות כי ההבדל בביצועים לעומת הסעיף הקודם שונה בהתאם לסוג הטבלה.
אנו רואים אמפירית כי היחס בין זמני הריצה בסעיף זה לבין זמני הריצה בסעיף הקודם אינו זהה לכל סוגי הטבלאות.
בפרט, יחס הגדילה בזמני הריצה עבור LPHashTable הוא הגדול ביותר, שגדל פי 12.71, לאחר מכן AQPHashTable שגדל פי 5.19, ולבסוף DoubleHashTable שגדל פי 5.17 (יחס הגדילה של AQP & DoubleHash יחסית זהה).
נסביר תוצאות אלו ע״י דברים שלמדנו בהרצאה ובתרגול.

ניזכר שבLPHashTable קיימת בעיית ההצטברות. לכן, ככל שנכניס יותר איברים לטבלה, ה-Load Factor גדל, מה שגורם לכך שכמות הבלוקים ואורכם יהיו גדולים ובכל הכנסה הסיכוי לעבור בחלק גדול מבלוק ארוך גדל. מסיבות אלו, זמן הריצה של LPHashTable הוא הארוך ביותר.
בנוסף, ל-LPHashTable ול-AQPHashTable יש אותה כמות של סדרות בדיקות - m סדרות בדיקות שונות.
ל-LPHashTable אין בעיית Cache Misses מאחר וסדרת הבדיקות היא רציפה.
כמו כן, מניסוי עזר שביצענו, מתגלה כי ל-AQPHashTable, ההפרש הממוצע של אינדקסים בסדרת חיפוש הוא 10, כלומר גם שם בעיית הCache Misses זניחה.

לכן, לא מפתיע שיחס הגדילה של LPHashTable גדול יותר מיחס הגדילה של AQPHashTable שכן שם ישנה בעיית הצטברות משנית, שכאמור פחות חמורה מבעיית ההצטברות של LPHashTable.

ב-DoubleHashTable לא קיימת בעיית ההצטברות, ובנוסף יש $m(m-1)$ סדרות של בדיקות לעומת m סדרות של בדיקות ב-LPHashTable וב-AQPHashTable. אולם, ל-DoubleHashTable קיים החסרון של ה-Cache Misses שדיברנו עליו בהרצאה. זו בעיה שנוצרת כתוצאה מגישות רבות יותר לצ׳אנקים שונים בזכרון (וזו נחשבת פעולה יקרה יחסית). בפרט, מניסוי עזר שביצענו, מתגלה כי ההפרש הממוצע בין אינדקסים בסדרת החיפוש הוא מסדר גודל של כ3 מיליון.
התברר לנו בסעיף א׳ שסדרת הבדיקות המפוזרת של DoubleHashTable כאשר הכנסנו את מחצית האיברים הראשונים לטבלה היוותה חסרון גדול יותר ביחס לחסרונות המימושים האחרים. אולם, מתברר לנו שכאשר מכניסים את 45% האיברים הנותרים וגורם העומס גדל, סדרת הבדיקות המפוזרת מהווה יתרון.

בנוסף, ניתן לראות לפי התוצאות האמפיריות, כי יחס הגדילה של DoubleHashTable ושל AQPHashTable יחסית זהים. כלומר, חסרון בעיית ההצטברות המשנית על פני m סדרות שונות של בדיקות ״דומה״ לחסרון של ה-Cache Misses על פני $m(m-1)$ סדרות של בדיקות שונות בניסוי זה.

שאלה 5

תוצאות הניסוי (זמן יצירת איברי הסדרה לא כלול בתוצאות, הרצנו 5 ניסויים שונים ומיצענו, מה שמקטין את המקריות):

Iterations	Running Time (Seconds)
First 3 Iterations	8.479
Last 3 Iterations	27.695

הסבר:
ראשית, כפי שניתן לראות בתוצאות, יש הבדל משמעותי בין 3 האיטרציות הראשונות לאחרונות (יותר מפי 3 בזמן הריצה).

תהליך ההכנסה מושפע מכמות האיברים המחוקים, כי בתהליך החיפוש שהוא עושה לבדוק האם המפתח נמצא ברשימה, במהלך ה-Probing Sequence כאשר הוא מגיע לאיברים מחוקים, הוא ממשיך בהליך החיפוש, לעומת המקרה בו יש איבר null שהוא מבין שאין בהמשך ה-Probing Sequence את המפתח שאנו מחפשים.

לכן, מאחר ובאיטרציה הראשונה אין בכלל איברים מחוקים, היא תהיה המהירה ביותר. לאחריה, בכל איטרציה נק׳ הפתיחה במבנה הנתונים היא שיש לפחות מחצית מהאיברים מסומנים כמחוקים.

לכן, ככל שאנו מתקדמים באיטרציות (2-6), נקודת הפתיחה שבה הטבלה נמצאת היא כזו שיש בה יותר (קיים סיכוי מאוד קטן שההכנסות יפלו בדיוק על אותם האיברים המחוקים וכך כמות האיברים המחוקים תישמר בין איטרציה לאיטרציה) איברים מחוקים מזו שקדמה לה, ולכן תהליך ההכנסה מתארך מהסיבה שתיארנו לעיל, מה שמתיישב עם תוצאות הניסוי שקיבלנו.