**מבני נתונים – פרויקט מעשי 2 - קובץ תיעוד**

**פרטים אודות מגישות-**

הגר פייתן

* שם משתמש - hagarleap
* תעודת זהות - 206825176

גל קריאל

* שם משתמש - galkariel
* תעודת זהות - 318459666

**חלק ראשון – תיעוד אודות הקוד**

|  |  |
| --- | --- |
| שם המחלקה | interface IHashTable |
| תיאור | מנשק של הטבלאות hash |
| פירוט המתודות במחלקה | Insert  Delete  Find |
| מחלקות פנימיות | מחלקות שזורקות שגיאות (extends Exception)- |

|  |  |
| --- | --- |
| שם המחלקה | OAHashTable |
| תיאור | מחלקה אבסטרקטית שממשת את המנשק IHashTable. |
| השדות במחלקה | table - טבלת ה-hash (מטיפוס מערך של hte) deleted – שדה סטטי שמסמל תא שנמחק (טיפוס the עם מפתח מינוס 1). |
| בנאי של המחלקה | this.table = new HashTableElement[m]  אתחול טבלת ה-hash כמערך בגודל m |
| פירוט המתודות במחלקה | Insert- מקבלים איבר מטיפוס hte.  שומרים משתנה בוליאני שמייצג אם הצלחנו להכניס את האיבר. ומשתנה first\_deleted\_index שמאותחל במינוס אחת ומטרתו לשמור את האינדקס של האיבר הראשון המחוק.  עוברים בלולאה על סדרת הבדיקות להכנסת איבר. מחשבים בכל איטרציה hash על המפתח של האיבר, לכל תוצאה (נסמנה index) נבדוק את הדברים הבאים:   1. אם המפתח של האיבר שאנחנו רוצים להכניס שווה למפתח של index נחזיר שגיאת KeyAlreadyExistsException. 2. אחרת, אם יש null ב- indexאז נפצל לשני מקרים: 3. אם לא חל שינוי ב- first\_deleted\_index אז נכניס את האיבר שלנו ב index. 4. אחרת, נכניס את האיבר במיקום בטבלה first\_deleted\_index. 5. אחרת, אם יש deleted ב- index וגם עוד לא בוצע שינוי בערך ההתחלתי של first\_deleted\_index אז נשנה את first\_deleted\_index להיות index.   לאחר סיום הלולאה.  נבדוק   1. האם הערך הבוליאני של הכנסת האיבר שווה לשקר – במידה וכן נבדוק גם האם first\_deleted\_index לא שווה למינוס אחת- ואז נכניס את האיבר שלנו במיקום first\_deleted\_index. 2. אחרת נחזיר שגיאה TableIsFullException.   **האופטימיזציה** שעשינו בפונקציה זו, הנה הבדיקה האם האיבר שרוצים להכניס לטבלה כבר קיים **תוך כדי סדרת** הבדיקות להכנסה (ולא קריאה נפרדת ל-find בכל פעם), שכן עבור תאים שמסומנים כמחוקים אי אפשר ישר להכניס את האיבר אלא צריך להמשיך לבדוק עד שמגיעים ל-null כי האיבר לא קיים.  Delete- מקבלים מפתח מטיפוס int . מחפשים אם קיים איבר עם המפתח הזה (FindIndex)- אם כן מחליפים את האיבר עם deleted. אם לא – מחזירים שגיאת KeyDoesntExistException.  Find- מקבלים מפתח מטיפוס int ומחפשים אותו בטבלת ה-hash לפי האלגוריתם שראינו בכיתה. אם קיים מחזירים את האיבר (מטיפוס hte), אחרת מחזירים null.  FindIndex- בדומה ל-find רק שמחזירים את האינדקס של האיבר שמחפשים ולא את האיבר עצמו. פונקציה זו שימושית עבור Delete.  Hash (מתודה אבסטרקטית) |

|  |  |
| --- | --- |
| שם המחלקה | ModHash |
| תיאור | מחלקה המייצגת פונקציית hash מהמשפחה האוניברסלית של פונקציות לינאריות מהצורה מודולו ראשוני מודולו גודל הטבלה . |
| השדות במחלקה | * a – מספר מטיפוס long * b– מספר מטיפוס long * m– גודל הטבלה מטיפוס int * p – מספר ראשוני מטיפוס long |
| בנאי של המחלקה | מאתחל את הערכים לפי הערכים שמקבל. |
| פירוט המתודות במחלקה | GetFunc- מקבלת את גודל הטבלה m ומספר ראשוני P, מגרילה שני מספרים a,b (בטווחים המתאימים כפי שראינו בשיעור) ומחזירה מופע חדש של ModHash עם מספרים אלה.  Hash- מקבלת מפתח ומחזירה חישוב hash מהצורה הבאה מודולו ראשוני מודולו גודל הטבלה . |

|  |  |
| --- | --- |
| שם המחלקה | HashTableElement |
| תיאור | מחלקה המייצגת איבר בטבלת ה-hash. |
| השדות במחלקה | * key - המפתח של האיבר מטיפוס long * value - הערך הנוסף מטיפוס long |
| בנאי של המחלקה | מאתחל את הערכים לפי הערכים שמקבל. |
| פירוט המתודות במחלקה | GetKey- מחזירה את המפתח.  GetValue- מחזירה את הערך. |

|  |  |
| --- | --- |
| שם המחלקה | LPHashTable |
| תיאור | מחלקה היורשת מ-OAHashTable. |
| השדות במחלקה | * m– גודל הטבלה מטיפוס int * p – מספר ראשוני מטיפוס long * Func- פונקציית ה-hash מטיפוס ModHash |
| בנאי של המחלקה | מאתחל את הערכים m,p לפי הערכים שמקבלת.  מאתחלת את Func- קריאה לקבלת פונקציית ממחלקת ModHash.GetFunc(m,p) |
| פירוט המתודות במחלקה | Hash- מקבלת מפתח ואינדקס. מחשבים את Func על המפתח מוסיפים את האינדקס ואז עושים מודלו m. |

|  |  |
| --- | --- |
| שם המחלקה | QPHashTable |
| תיאור | מחלקה היורשת מ-OAHashTable. |
| השדות במחלקה | * m– גודל הטבלה מטיפוס int * p – מספר ראשוני מטיפוס long * Func- פונקציית ה-hash מטיפוס ModHash |
| בנאי של המחלקה | מאתחל את הערכים m,p לפי הערכים שמקבלת.  מאתחלת את Func- קריאה לקבלת פונקציית ממחלקת ModHash.GetFunc(m,p) |
| פירוט המתודות במחלקה | Hash- מקבלת מפתח ואינדקס. מחשבים את Func על המפתח מוסיפים את האינדקס בריבוע ואז עושים מודלו m. |

|  |  |
| --- | --- |
| שם המחלקה | AQPHashTable |
| תיאור | מחלקה היורשת מ-OAHashTable. |
| השדות במחלקה | * m– גודל הטבלה מטיפוס int * p – מספר ראשוני מטיפוס long * Func- פונקציית ה-hash מטיפוס ModHash |
| בנאי של המחלקה | מאתחל את הערכים m,p לפי הערכים שמקבלת.  מאתחלת את Func- קריאה לקבלת פונקציית ממחלקת ModHash.GetFunc(m,p) |
| פירוט המתודות במחלקה | Hash- מקבלת מפתח ואינדקס. בודקים זוגיות של האינדקס (על ידי בדיקה פשוטה של הביט הראשון כאשר האינדקס מיוצג כמספר בינארי)-   1. אם זוגי- מחשבים את Func על המפתח **מוסיפים** את האינדקס בריבוע ואז עושים מודלו m. 2. אם אי זוגי- מחשבים את Func על המפתח **מורידים** את האינדקס בריבוע ואז עושים מודלו m.   זו **אופטימיזציה** – שכן עשינו זאת על מנת לא לחשב את מינוס 1 בחזקת האינדקס שכן זה פעולה יקרה. |

|  |  |
| --- | --- |
| שם המחלקה | DoubleHashTable |
| תיאור | מחלקה היורשת מ-OAHashTable. |
| השדות במחלקה | * m– גודל הטבלה מטיפוס int * p – מספר ראשוני מטיפוס long * Func1- פונקציית ה-hash מטיפוס ModHash * Func2- פונקציית ה-hash מטיפוס ModHash |
| בנאי של המחלקה | מאתחל את הערכים m,p לפי הערכים שמקבלת.  מאתחלת את Func1- קריאה לקבלת פונקציית ממחלקת ModHash.GetFunc(m,p)  מאתחלת את Func2- קריאה לקבלת פונקציית ממחלקת ModHash.GetFunc(m-1,p) |
| פירוט המתודות במחלקה | Hash- מקבלת מפתח ואינדקס. מחשבים את Func1 על המפתח מוסיפים את האינדקס כפול (Func2 על המפתח + 1) ואז עושים מודלו m. |

**חלק שני- ניסויי/תיאורטי**

**שאלה 3 – השוואה בין quadratic probing ל-alternating quadratic probing**

**שאלה 3 - סעיף 1**

גדלי הקבוצות:

( נבחין כי הגודל של זה בדיוק וכי הגודל של זה בדיוק q )

**שאלה 3 - סעיף 2**

מדדנו עבור quadratic probing את כמות השגיאות הנובעות משגיאה כי הטבלה מלאה ומשגיאה כי מפתח כבר קיים במערך.

ראינו כי אין שגיאות שנובעות ממפתח שקיים כבר במערך, אבל קיבלנו בין 0 ל-3 שגיאות בכל ניסוי שנובעות מכך שהטבלה מלאה.

לעומת זאת עבור alternating quadratic probing לא נזרקו שגיאות כלל וכל האיברים הוכנסו לטבלה.

הסבר לשוני בין התוצאות-

ראינו מהסעיף הקודם כי כאשר אנחנו משתמשים במודל של quadratic probingאז כתוצאה סדרת הבדיקות שלנו עוברת רק על קבוצת מספרים בגודל חצי מהמספר הראשוני q () . כלומר פונקציית ה-hash במודל של quadratic probing לא נותנת לנו את כל האינדקסים בטבלה ולכן ייתכנו סדרות של בדיקות שלא עוברות על כל האינדקסים בטבלה כלומר נקבל שגיאות כי הטבלה מלאה למרות שבפועל יש בה עוד מוקמות להכניס איברים.

לעומת זאת כאשר אנחנו משתמשים במודל של alternating quadratic probing ראינו בסעיף הקודם שסדרת הבדיקות שלנו עוברת על קבוצת מספרים בגודל q () כלומר אין חזרות של מספקים ואנחנו עוברים **על כל** האינדקסים בטבלה– כלומר בהכרח נמצא מקום פנוי אם קיים לטובת הכנסת מפתח חדש לכן לא נקבל שגיאות ונצליח להכניס את כל המפתחות.

**שאלה 3 - סעיף 3**

תיאוריה אודות שאריות ריבועיות-

מספר a נקרא שארית ריבועית אם כש-x מספר שלם.

* אם p מספר ראשוני, יש בדיוק שאריות ריבועיות (למעט השארית 0(, ו- שאריות לא ריבועיות.
* עבור p ראשוני, כאשר מתקיים כי - אם **a** הינו שארית ריבועית, אז **-a** הינו שארית לא ריבועית.

כעת נבחין כי בשאלה 1 בחלק זה והוא מספר ראשוני שמקיים .

הדבר מסתדר עם התופעה שראינו שהגודל של זה בדיוק והגודל של זה בדיוק q :

ב- אנחנו מחשבות רק מספרים חיוביים לכן נקבל מספרים שונים.

ב- אנחנו מחשבים גם מספרים חיוביים וגם מספרים שליליים (יש את הכפל במינוס 1 שמתחלף), לכן נקבל גם שאריות ריבועיות וגם שאריות לא ריבועיות:

כלומר נקבל מספרים שונים.

לאור האמור לעיל, תופעה זו מתקיימת עבור מספרים ראשוניים q שמקיימים , וזהו גם התנאי לקיום התופעה.

**שאלה 4 – נשווה בין המימושים השונים ל-open addressing**

**שאלה 4 - סעיף 1**

|  |  |
| --- | --- |
| Class | Running Time |
| LPHashTable | 0.510 |
| QPHashTable | 0.498 |
| AQPHashTable | 0.514 |
| DoubleHashTable | 0.704 |

הסבר מילולי להבדלים בזמני הריצה בין סוגי הטבלאות-

נבחין כי זמני הריצה של DoubleHashTable יותר ארוכים באופן יחסי לעומת שאר המימושים. ושאר המימושים רצים בזמנים מאוד דומים.

ראינו בהרצאה כי הסיכוי להתנגשויות כאשר משתמשים בפונקציות hash מהמשפחות האוניברסליות הנו נמוך ( כאשר m זה גודל טבלת ה-hash). בנוסף אנחנו ממלאים רק את חצי הטבלה, ולכן נעריך כי אפילו אם יש התנגשויות לא יהיו "רצפים" גדולים מידי של איברים בטבלה - שעלולים היו לפגוע בזמני הריצה עבור LPHashTable למשל). כלומר בעיית הרצפים (הצטברות ראשונית/משנית) עבור סעיף זה לא תשפיע הרבה על זמני הריצה.

בנוסף לכך, נרצה לציין כי ציפינו ש- LPHashTable ייקח פחות זמן מ- QPHashTableu ומ-AQPHashTable כיוון שראינו במבוא מורחב כי חיבור מספר בודד זה פעולה זולה יותר מהעלאה בחזקה. אבל נראה כי אין משמעות גדולה בזמני הריצה לפעולות חישוב אלה (חיבור והעלאה בחזקה).

עבור LPHashTable, QPHashTable ו- AQPHashTable מתבצעת אותה כמות של פעולות מודולו וראינו כי זמני הריצה שלהם דומים.

לעומת זאת ב- DoubleHashTable יש יותר פעולות מודולו (אנחנו מפעילים שתי פונקציות hash עבור כל בדיקה להכנסת איבר) ולכן אנו מעריכות כי השוני בזמני הריצה נובע מפעולת המודולו.

**שאלה 4 - סעיף 2**

|  |  |
| --- | --- |
| Class | Running Time |
| LPHashTable | 5.551 |
| AQPHashTable | 2.776 |
| DoubleHashTable | 3.836 |

ההסבר לכך שאין לבצע סעיף זה עבור הנה - כפי שראינו בשאלה הקודמת, פונקציית ה-hash במודל של quadratic probing לא נותנת לנו את כל האינדקסים בטבלה, ובפרט נותנת לנו רק חצי מהם. לכן הכנסה של כמות איברים שגדולה מחצי מהטבלה עלולה לגרום לשגיאות "table is full" כיוון שסדרת הבדיקות לטובת הכנסה תהיה רק על חצי מהטבלה.

בניסוי זה אנו מכניסים יותר מחצי איברים של הטבלה ולכן ייתכן ויהיו שגיאות (בשאלה זו לא אמורים להיזרק חריגים) ולכן לא נבצע בסעיף את המימוש עבור .

הסבר להבדל בביצועים (זמני הריצה) לעומת הסעיף הקודם:

באופן כללי, נשים לב כי כמות ההתנגשויות שנצפה בניסוי זה משתנה באופן הבא:

תוחלת של ניסוי 1:

תוחלת של ניסוי 2:

כלומר אם לא הייתה בעיית הצטברות וזמן הריצה היה רק תלוי בחישוב יחיד של אינדקס חדש בכל מקרה של התנגשות, אז היינו מצפות לזמן ריצה לגדול פי 3.6 ().

LPHashTable- זמני הריצה של מודל זה הנם הארוכים ביותר.

זמני הריצה גדלו בערך פי 10 לעומת הניסוי הקודם וזו הגדילה הכי גדולה לעומת AQP ו- DoubleHash.

אנו מעריכות כי הדבר נובע מבעיית ההצטברות הראשונית – נוצרים רצפים ארוכים של תאים תפוסים. סדרות הבדיקה להכניס איבר שיש רצפים ארוכים יהיו יקרות (הרבה פעולות מודלו).

AQPHashTable-

זמני הריצה גדלו בערך פי 5 לעומת הניסוי הקודם.

אנו מעריכות כי הדבר נובע מבעיית ההצטברות המשנית – זוהי צורה מתונה יותר של הצטברות.

יפה לראות כי בעיית ההצטברות המשנית, באמת מתונה יותר מבעיית ההצטברות הראשונית (זמני ריצה שגדלו פי 5 לעומת זמני ריצה שגדלו פי 10 ב-LP).

DoubleHashTable-

זמני הריצה גדלו בערך פי 5 לעומת הניסוי הקודם.

בשיטה זו תיאורטית לא אמורים להיות רצפים ארוכים של תאים תפוסים שכן סדרת הבדיקות עבור הכנסה מחושבת על ידי שתי פונקציות hash שונות.

תוצאות אלה מפתיעות אותנו לאור האמור לעיל, שכן היינו מצפות שזמני הריצה בניסוי זה יגדלו פחות מפי 5.

הסבר אפשרי לכך שזמני הריצה גדלו פי 5 בכל זאת יהיה, שכאשר הטבלה לקראת התמלאות יש יותר התנגשויות מכיוון שהתנגשויות ראשוניות יובילו להתנגשויות נוספות. הדבר יגרור יותר חישובים יקרים.

אפשר לראות אימפירית ששיטת הdoubleHash מאכזבת. הרעיון מאחוריה הנו להימנע מרצפים גדולים ובכך לתת זמני ריצה טובים, אבל בסופו של דבר ראינו כי היא איטית באופן יחסי אל מול שיטת ה-AQP בשני הניסויים.

**שאלה 5 – השפעת מחיקת איברים ב-open addressing**

|  |  |
| --- | --- |
| Iterations | Running Time |
| First 3 iterations | 4.513 |
| Last 3 iterations | 19.45 |

נשווה את זמן ביצוע 3 האיטרציות הראשונות לזמן ביצוע 3 האיטרציות האחרונות.

נבחין כי קיים הבדל משמעותי בין זמני הריצה, ב-3 האיטרציות הראשונות זמני הריצה קצרים משמעותית מ-3 האיטרציות האחרונות.

אנו מעריכות כי הסיבה להבדל הנה שאנו לא מאפסות בכל איטרציה מחדש את המבנה. לכן כשעושים יותר איטרציות יש יותר תאים בטבלה שמסומנים כ-deleted (כיוון שאנחנו מוחקים איברים את כל האיברים שהכנסנו למבנה בכל איטרציה) ולא כ-null . בפונקציית ה-Insert שלנו אנחנו מכניסים איבר בתא שמסומן deleted רק אחרי שעברנו על כל הטבלה לוודא שאין מפתח זהה לזה שאנחנו רוצים להכניס (כיוון שיתכן שהמפתח נמצא בהמשך הטבלה), לעומת זאת עבור תא שמכיל- null אנחנו מכניסים את האיבר החדש ישירות לטבלה שלנו ומסיימים. לכן ככל שיהיו יותר תאים שמסומנים deleted נקבל זמני ריצה ארוכים יותר עבור ההכנסות ולכן זמני ריצה ארוכים יותר עבור הניסוי.