### הגדרות

**אלגוריתם-** דרך שיטית וחד משמעית לביצוע של משימה מסוימת במספר **סופי** של צעדים

**פסאדו קוד –** תיאור מצומצם ולא רשמי לאלגוריתם של תוכנית מחשב

**זמן ריצה –** מספר פעולות **היסוד** המבוצעות על תוכנית כלשהי

**למה פעולות יסוד ולא זמן?** על מנת להתעלם מאספקטים טכנולוגים כמו סוגי מכונות

**דוגמא:** חיפוש בינארי מול חיפוש טרינרי

|  |  |
| --- | --- |
| חיפוש בינארי | חיפוש טרינרי |
|  |  |

**אסימפטומטיקה – הערכה** של קצב גידול של פונקציה

**חסם עליון –** נאמר ש- אם:

**חסם תחתון –** נאמר ש- אם:

**חסם הדוק –** נאמר ש- אם:

|  |  |
| --- | --- |
| שם | סימון |
| אקספוננציאלי |  |
| פולינומי |  |
|  |  |
| לינארי |  |
|  |  |
| לוגריתמי |  |
|  |  |
| קבוע |  |
|  |  |

### מיונים

**הקדמה:**

**מיון מבוסס השוואות** – מיון מבוסס השוואות מסדר אלמנטים במערך ע״י השוואה, בדרך כלל ע״י האופרטורים

כל אלגוריתם המיון המבוסס על פעולות השוואה דורש לפחות פעולות השוואה **במקרה הגרוע** ביותר

**הוכחה:** עבור מיון איברים יש צורך בעץ החלטה עם עלים (עבור כל אחת מהפרמוטציות השונות), בעץ החלטה בגובה יש לכל היותר עלים, ולכן:

*על מנת להשיג סיבוכיות מיון טובה יותר מ- (ליניאריים) נוכל לאבד את* ***הכלליות*** *ולהתעסק במקרים פרטניים כמו* ***מספרים נמצאים בטווח חסום*** *(Counting Sort),* ***סדרה של מספרים הנמצאים בטווח חסום*** *(Radix Sort)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *לא מבוססי השוואות* | | *מבוססי השוואות* | | | | |  |
| *Counting Sort* | *Radix Sort* | *Quick Sort* | *Merge Sort* | *Insertion Sort* | *Selection Sort* | *Bubble Sort* |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  | *בהסתברות גבוהה* |  |  |  |  |  |

### מבני נתונים

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| סיבוכיות מחיקה | סיבוכיות חיפוש | סיבוכיות הכנסה | מוטיבציה | הגדרה | מבנה נתונים |
|  |  |  | גודל המערך ידוע בזמן קומפילציה   * רוצים **Random Access** (גישה ישירה בזמן קבוע) | מבנה שמורכב מאוסף של תאים **סדרתיים** בזיכרון בעל גודל **קבוע** | מערך סטטי |
|  |  | *עד כדי resize שלוקח* | **גודל המערך לא ידוע בזמן** **קומפילציה** | מבנה שמורכב מאוסף של תאים **סדרתיים** בזיכרון בעל גודל **ניתן לשינוי** | מערך דינמי |
|  |  | בגלל shifting של האיברים | נרצה לבצע **חיפוש** מהיר, כאשר יש **יותר חיפושים מאשר הכנסות** | אוסף של תאים סדרתיים בזיכרון כך שכל איבר קטן מהאיבר הבא לפי אופרטור השוואה כלשהו | מערך דינמי ממויין |
|  |  |  | **יותר הכנסות** או **אין צורך ב-Random Access, פחות אפקטיבי (זניח) מבחינת זיכרון** עבור שמירה גם על גודל המצביע | * לכל איבר יש ערך * לכל איבר יש לכל היותר מצביע יחיד לאיבר הבא | רשימה מקושרת |
|  |  |  | * מימוש לתור עדיפויות היות ו**מחיקה מהראש ומהסוף היא פעולה קלה** * מחיקה של איברים זהים כאשר **אין זיכרון** לטעון ל-HashSet לדוגמא | * לכל איבר יש ערך * לכל איבר יש לכל היותר מצביע יחיד לאיבר הבא, שהוא קטן מהאיבר הבא לפי אופרטור השוואה כלשהו | רשימה מקושרת ממוינת |
|  | - |  | * LIFO | * מבנה נתונים אשר תומך ב**סדר** פעולות היסטורי (האחרון הוא הראשון לצאת)   Regency Red 18 3/4" x 11 1/2" Plastic Grocery Market Shopping Baskets with  Stand and Sign | מחסנית |
|  | - |  | * FIFO   **הערה:** ניתן לממש תור בעזרת 2 מחסניות ומחסנית בעזרת 2 תורים (יש שקילות) | מבנה נתונים אשר תומך ב**סדר** פעולות היסטורי (האחרון הוא האחרון לצאת)  EasyFlow | Latest trends in Retail Queue Management | תור |
|  |  |  |  | עץ בינארי (לכל קודקוד יש ערך ולכל קודקוד יש רשימה של לכל היותר 2 קודקודים) כך שעבור **כל** קודקוד בעל ערך :   1. הערכים הקטנים מ- נמצאים בתת עץ השמאלי 2. הערכים הגדולים מ- נמצאים בתת עץ הימני | עץ חיפוש בינארי |
|  |  |  | * **הכנסה, מחיקה, חיפוש** יעיל * **חיפוש תווכים, איברים סמוכים, מבנה נתונים בעל סדר, InOrder** | עץ חיפוש בינארי כך שכמות הרמות של לוגריתמית ביחס לכמות האיברים בעץ   * AVL * עץ אדום שחור   [הוכחות עבור גובה עץ לוגרתמי](https://github.com/ZviMints/DataStructures_2020S/blob/master/%D7%AA%D7%A8%D7%92%D7%95%D7%9C%205%20%D7%A2%D7%A6%D7%99%D7%9D%20%D7%9E%D7%90%D7%95%D7%96%D7%A0%D7%99%D7%9D/%D7%AA%D7%A8%D7%92%D7%95%D7%9C%205%20-%20%D7%A2%D7%A6%D7%99%D7%9D%20%D7%9E%D7%90%D7%95%D7%96%D7%A0%D7%99%D7%9D.pdf)  היות ועץ AVL הוא עץ יותר מאוזן מעץ אדום-שחור, נעדיף להשתמש בו אם כמות החיפושים גדולה, אך אם מתבצעים הרבה שינויים כמות הכנסה או מחיקה, יתבצעו הרבה רוטציות ולכן נעדיף עץ אדום שחור שהוא יותר ״סלחני״ | עץ חיפוש מאוזן |
| מחליפים את השורש עם האיבר האחרון, מורידים את הגודל ב-1 ואז עושים Heapify מהשורש, כאשר בודקים מי מהבנים הוא עם עדיפות ומחליפים | עובר השורש עבור חיפוש כלשהו | מכניסים לסוף, ואז עושים SwapUp עד לשורש | * עץ עדיפות   (במקום חיפוש נרצה עדיפות לפי אופרטור)  **הערה:** בנייה של עץ ערמה הוא כי עושים Heapify כחצי מגודל המערך  [הוכחה](https://github.com/ZviMints/DataStructures_2020S/blob/master/%D7%AA%D7%A8%D7%92%D7%95%D7%9C%206%20%D7%A2%D7%A6%D7%99%20%D7%A2%D7%A8%D7%9E%D7%94/%D7%AA%D7%A8%D7%92%D7%95%D7%9C%206%20-%20%D7%A2%D7%A8%D7%9E%D7%94.pdf) | * כל האיברים בעץ ניתנים להשוואה ע״י אופרטור * הערך של כל קודקוד קטן או שווה ביחס לאופרטור מהערך של בניו * ממומש ע״י מערך כך שהבן השמאלי באינדקס והבן הימני באינדקס כאשר ראש הערמה באינדקס 0   מבנה נתונים אשר תומך ב**עדיפות** לפי אופרטור | ערמה |
| **Direct Access Table:**  **Open Addressing:**  **Separate Channing:**  *תלוי מימוש, מערך של רשימות מקושרות*  *בעוד שמערך של עצי חיפוש בינאריים*  *עם Load Factor ופונקציית גיבוב טובה, נוכל לקבל O****(1) Amortized*** | **Direct Access Table:**  **Open Addressing:**  **Separate Channing:**  *תלוי מימוש, מערך של רשימות מקושרות*  *בעוד שמערך של עצי חיפוש בינאריים*  *עם Load Factor ופונקציית גיבוב טובה, נוכל לקבל* ***O(1) Amortized*** | **Direct Access Table:**  **Open Addressing:**  **Separate Channing:**  *תלוי מימוש, מערך של רשימות מקושרות*  *בעוד שמערך של עצי חיפוש בינאריים*  *עם Load Factor ופונקציית גיבוב טובה, נוכל לקבל* ***O(1) Amortized*** | על מנת לאפשר חיפוש, הוספה ומחיקה בסיבוכיות קבועה, ניתן להשתמש ב-Direct Access Table על מנת לקבל סיבוכיות זו ע״י מקום כאשר זה מרחב המפתחות האפשריים, או לצמצם את הזיכרון ולפתור בעיות התנגשות (Collision)  *אך, משלמים על זה בכך שמבנים דינמאים הם יעילים מבחינת זיכרון, בעוד שטבלת גיבוב היא בזבזנית, בנוסף אם אנחנו רוצים להתעסק עם מידע ממוין, מציאת איברים סמוכים, חיפוש תווכים, להציג מבנה נתונים ממויין אז נעדיף להשתמש בעצים לדוגמא* | * לכל ערך יש מפתח * בעל פונקציית גיבוב   **פונקציית גיבוב טובה:**   1. כמות הפלטים קטנה מכמות הקלטים האפשריים 2. כמות התנגשויות צריכה להיות קטנה ככל שאפשר 3. קלה לחישוב 4. דטרמיניסטית 5. תשתמש בכל המידע שיש למפתח 6. מפזרת אחיד 7. חד כיוונית לעיתים  * פותר התנגשויות   **לדוגמא:**   * Open Addressing (מציאת מקום חדש לאובייקט) * Separate Chaining (מערך של רשימות מקשורות, עצי חיפוש, וכו׳) | טבלת גיבוב |