**מבני נתונים – פרוייקט מספר 2 – ערמת פיבונאצ'י**

מגיש 1:

יעקב אודסר

iakovodesser

209860188

מגיש 2:

איל ספיר

eyalsapir

206405417

להלן תיעוד חיצוני של הפונקציות במחלקה FibonacciHeap. התיעוד כתוב באנגלית באישור המתרגל, כי ככה היה לנו נוח יותר לכתוב אותו תוך כדי המימוש:

**insert(int key, String info):**

Inserting a new node to a heap, we check whether the heap is empty. If so, we make our new and only node min, and create a circular linked list with only the one node. Otherwise, we insert it as a new root in the preexisting list, pushing it between the min and the following root. Lastly, we increment the size of the heap. The only operations we've used are the simplest ones - initializing new objects, reassigning pointers and incrementing an integer.

The insert is indeed lazy, with time complexity of O(1).

**findMin()**

The method does little to none - returning the pointer to a field which is maintained through all other methods.

Time complexity is self-evidently O(1).

**deleteMin()**

deleteMin removes the global minimum node from the heap and performs the following steps:

1) If the heap is empty, there is nothing to delete and the method returns.

2) Otherwise, it detaches the children (if any) of the minimum node and adds them to the root list.

3) It removes the minimum node from the root list and decreases the total size by one.

4) If the heap is not empty after removal, it calls consolidate() to restore the Fibonacci Heap structure:  
consolidate first counts how many roots are in the circular list and stores them in a temporary array (rootArray). This involves traversing the root list twice: once to count and once to populate the array. It then merges any roots of the same rank (via link operations) and rebuilds a new root list, updating the min pointer.

Time complexity:

- detachChildren costs O(r) = O(log n), where r is the rank (number of children) of the deleted node.

- consolidate involves creating and iterating over the rootArray of size O(log n) (since a Fibonacci Heap has O(log n) roots), merging them in O(1) per link, and finally constructing the new root list and finding the min in another O(log n) pass.

- Thus, deleteMin overall runs in O(log n) time.

**decreaseKey(x, d)**

takes a node x whose key is reduced by d. After decreasing its key, if x violates the heap property (its key is now smaller than its parent's key), x is cut from its parent and placed into the root list. This may trigger cascading cuts: if the parent was already marked, it is also cut from its parent, and so on up the tree.

Time complexity:

- Decreasing x.key itself is O(1).

- Checking x against its parent is O(1).

- Cutting x from its parent is O(1) amortized, and placing x in the root list is also O(1).

- Cascading cuts can propagate up the tree, but each node can only be cut once before it must be reattached and marked again. This ensures the operation is amortized O(1).

- Hence, the amortized time complexity for decreaseKey is O(1).

**delete(x)**

This method performs the following steps:

1. If the node is the minimum, it calls deleteMin() to remove it.

2. Otherwise, it decreases the node's key to make it the global minimum (using decreaseKey), and then removes it using delteMinNocons.

Runtime Analysis:

If the node is the minimum, deleteMin is O(log n)

If the node is not the minimum:

- decreaseKey: is O(1).

- delteMinNocons is dominated by the number of leaves detached from the minimum which is worst case O(log n) but in average O(1).

Hence ovreall run time is worst case O(log n) but in average O(1)

**totalLinks(), totalCuts(), size(), numTrees()**

Having maintained proper fields all along, we just return them all in O(1).

**meld(heap2)**

Melds the current heap (this) with another FibonacciHeap (heap2). After calling meld, heap2 becomes redundant (effectively empty), and 'this' contains all elements from both heaps.

The method does nothing but simple if-tests for edgecases and reassigning pointers consecutively. O(1).

**החלק הניסויי**

שאלה 1: להלן תוצאות הניסויים

ניסוי 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **מספר סידורי i** | **זמן ריצה (מילישניות)** | **גודל הערמה בסיום** | **מספר חיבורים** | **מספר חיתוכים** | **מספר עצים בסיום** |
| 1 | 2.60 | 6559 | 6550 | 0 | 9 |
| 2 | 4.20 | 19681 | 19674 | 0 | 7 |
| 3 | 6.50 | 59047 | 59037 | 0 | 10 |
| 4 | 21.15 | 177145 | 177133 | 0 | 12 |
| 5 | 104.55 | 531439 | 531427 | 0 | 12 |

ניסוי 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **מספר סידורי i** | **זמן ריצה (מילישניות)** | **גודל הערמה בסיום** | **מספר חיבורים** | **מספר חיתוכים** | **מספר עצים בסיום** |
| 1 | 6.25 | 3280 | 38224 | 0 | 5 |
| 2 | 14.15 | 9841 | 130157 | 0 | 7 |
| 3 | 35.20 | 29524 | 434847 | 0 | 8 |
| 4 | 104.35 | 88573 | 1449086 | 0 | 12 |
| 5 | 518.45 | 265720 | 4734265 | 0 | 9 |

ניסוי 3:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **מספר סידורי i** | **זמן ריצה (מילישניות)** | **גודל הערמה בסיום** | **מספר חיבורים** | **מספר חיתוכים** | **מספר עצים בסיום** |
| 1 | 3.95 | 31 | 6984 | 3296 | 5 |
| 2 | 5.60 | 31 | 22313 | 11140 | 1 |
| 3 | 14.25 | 31 | 63656 | 30734 | 3 |
| 4 | 31.95 | 31 | 206186 | 102421 | 1 |
| 5 | 132.80 | 31 | 575119 | 269926 | 4 |

שאלה 2:

עבור כל הניסויים, הכנסה בודדת לערימת פיבונאצ'י עולה O(1) בזמן אמורטייזד ולכן עבור n צמתים תעלה ההכנסה בסך הכול O(n).

עבור הניסוי הראשון, נבצע deleteMin פעם אחת בלבד. עלות הפעולה היא O(logn) ולכן זמן הריצה הכללי הוא O(n)+O(logn)=O(n).

בניסוי השני נבצע n/2 פעולות deleteMin שיעלו O(2n​⋅logn)=O(nlogn), לכן זמן הריצה הכללי הוא O(n)+O(nlogn)=O(nlogn).

בניסוי השלישי אנחנו מבצעים פעולות מחיקה עד שהערימה מגיעה לגודל 31, כלומר /2(n−31) פעולות deleteMin ו-/2(n−31) פעולות delete. פעולות deleteMin עולות O(logn) בעוד פעולות delete עולות בממוצע O(1). מכאן שהעלות האסימפטוטית היא O(nlogn)+O(n)=O(nlogn) למרות שבפועל היא תהיה נמוכה משמעותית בהשוואה לזמן שהיה מתקבל מביצוע מספר דומה של פעולות deleteMin בלבד.

שאלה 3:

עבור הניסוי הראשון והניסוי השני, מספר החיבורים ישתנה משום שמספר החיבורים תלוי בדרגות של השורשים, שמושפעות מסדר ההכנסה. באופן דומה, מספר העצים בערימה תלוי בפעולות ה- consolidation שמושפעות מדרגות הצמתים, שמושפעות מסדר ההכנסה. בהתאם לכך גם זמן הריצה עשוי להשתנות, אך לא במונחים אסימפטוטיים. פעולות חיתוך לא מתרחשות כחלק מ- deleteMinולכן לא יושפעו, וכמובן גודל הערימה לא יושפע.

עבור הניסוי השלישי האמירות על מספר החיבורים ומספר העצים תקפות, ובנוסף מבנה העץ משפיע על מספר הפעמים שצריך לעשות Cascading cuts כחלק מפעולת ה-decreaeKey הדרושה לפעולת ה-delete הרגילה, ולכן בעקיפין גם על מספר החיתוכים.

שאלה 4:

הקשר בין מספר פעולות החיבור ומספר פעולות החיתוך הוא שפעולות חיבור מקטינות את מספר העצים בערימה בעוד פעולות חיתוך מגדילות את מספר העצים בערימה, לכן אם נסמן ב- ΔT את השינוי במספר העצים בעץ, מתקיים ΔT=Cuts−Connections, כאשר עבור ΔT>0 מספר החיבורים גדול ממספר החיתוכים ולהפך.

לפיכך, אם נסמן ב-ti את מספר העצים בעץ לפני הפעולה וב-tf את מספר העצים אחרי הפעולה נקבל את המשוואה: Connections=Cuts− (tf−ti​).

נבחין כעת שמתקיים = n ti בפעולה הראשונה עבור ערימה בגודל n, משום שלפני פעולת ה-consolidation הראשונה מספר העצים בעץ שווה למספר האיברים בו. מנגד מתקיים tf​=D(n)+1≈logϕ​(n)+1, משום שהדרגה המקסימלית של צומת בערימת פיבונאצ'י תקינה היא D(n)≈logϕ​(n), ולאחר פעולת consolidation רשימת השורשים כוללת לכל היותר צומת אחד מכל דרגה בין 0 ל-D(n).

לסיכום מתקיים: Connections = Cuts− (tf−ti​) = Cuts− (n−(logϕ​(n)+1)).