מבני נתונים

תרגיל מעשי 2

תיעוד חיצוני

איוב אטרש – 318476611

מוחמד עאמר - 208644963

|  |  |  |
| --- | --- | --- |
| **Complexity** | **Explanation** | **Function** |
| O(1) | The method returns true if and only if the heap is empty. | **public** **boolean** empty() |
| O(1) | Creates a node (of type HeapNode) which contains the given key, and inserts it into the heap. | **public** HeapNode insert(**int** key) |
| Amortized:  O(log(n))  Worst case:  O(n) | Delete the node containing the minimum key. It calls addChildrenToRoots(). | **public** **void** deleteMin() |
| O(1) | It adds the children of the minimum to the root list. | **private** **void** addChildrenToRoots() |
| O(1) | Return the node of the heap whose key is minimal. | **public** HeapNode findMin() |
| O(1) | Meld the heap with heap2. | **public** **void** meld (FibonacciHeap heap2) |
| O(1) | Return the number of elements in the heap. | **public** **int** size() |
| O(log (n)) | Return a counters array, where the value of the i-th entry is the number of trees of order i in the heap. | **public** **int**[] countersRep() |
| O(log (n)) | Deletes the node x from the heap. | **public** **void** delete(HeapNode x) |
| O(1) | The function decreases the key of the node x by delta. The structure of the heap should be updated to reflect this chage (for example, the cascading cuts procedure should be applied if needed). | **public** **void** decreaseKey(HeapNode x, **int** delta) |
| O(1) | This function returns the current potential of the heap, which is:  " **Potential** = #trees + 2\*#marked "  The potential equals to the number of trees in the heap plus twice the number of marked nodes in the heap. | **public** **int** potential() |
| O(1) | This static function returns the total number of cut operations made during the run-time of the program. A cut operation is the operation which diconnects a subtree from its parent (during decreaseKey/delete methods). | **public** **static** **int** totalCuts() |
| O(1) | This static function returns the total number of link operations made during the run-time of the program.  A link operation is the operation which gets as input two trees of the same rank, and generates a tree of rank bigger by one, by hanging the tree which has larger value in its root on the tree which has smaller value in its root. | **public** **static** **int** totalLinks() |
| O(1) | It removes x from y's list of children . | **public void** cut(HeapNode x , HeapNode y) |
|  | Cuts x from y (its parent) , and then does the same for y recursively. | **public void** cascading\_cut(HeapNode x , HeapNode y) |
| O(1) | A key restricted constructor of a node in the Fibonacci heap (everything besides key equals the default value of the parameter). | **public** HeapNode(**int** key) |
| O(1) | A specific constructor of a node in the Fibonacci heap . | **public** HeapNode(**int** key,HeapNode prev,HeapNode parent,HeapNode next,HeapNode child) |
| O(1) | Returns the key of the node. | **public** **int** getKey() |
| O(1) | Returns the rank of the node. | **public** **int** getRank() |
| O(1) | Returns true if the node is marked, otherwise false. | **public** **boolean** getMark() |
| O(1) | Returns the next node of the node. | **public** HeapNode getNext() |
| O(1) | Returns the parent of the node. | **public** HeapNode getParent() |
| O(1) | Returns the previous node of the node. | **public** HeapNode getPrev() |
| O(1) | Returns the child of the node. | **public** HeapNode getChild() |
| Amortized:  O(log(n))  Worst case:  O(n) | Goes over the list of roots and links those with equal ranks. It calls actualConsolidation(), initBuckets() and countRoots(). | **public void** Consolidation() |
| Amortized:  O(log(n))  Worst case:  O(n) | It does the actual consolidation ! | **public void** actualConsolidation() |
| O(1) | Returns the number of roots in the heap. | **private int** countRoots() |
| O(log(n)) | Returns a HeapNode array initialized to null . | **private** **HeapNode[]** initBuckets() |
| O(1) | Gets as input two nodes of the same rank, and generates a tree of rank bigger by one, by hanging the tree which has larger value in its root on the tree which has smaller value in its root(x). | **public void** Link(HeapNode x , HeapNode y) |

**שדות ב – FibonacciHeap :**

|  |  |
| --- | --- |
| **Details** | **Fields** |
| a HeapNode pointer that points to the node with the minimum key value. | **private** HeapNode min |
| Holds the number of marked nodes, we added this parameter to calculate the potential more efficiently. | **private** **int** marked\_nodes=0 |
| Holds the number of total cuts , this static parameter value is updated in decreasekey and returned in totalcuts(). | **private static int** *total\_cuts*=0 |
| Holds the number of linkings , this parameter value is updated in deletemin()\conselidate() and returned in totallinks(). | **private static int** *total\_links*=0 |
| Holds the number of nodes in the heap. | **private int** size=0 |

**שדות ב – HeapNode:**

|  |  |
| --- | --- |
| **Details** | **Fields** |
| Holds true if the node is marked, otherwise false. | **public** **boolean** mark=**false** |
| Is a pointer for the parent of the node. | **public** HeapNode parent |
| Is a pointer for the next node. | **public** HeapNode next |
| Is a pointer for the previous node. | **public** HeapNode prev |
| Is a pointer for the child of the node. | **public** HeapNode child |
| Holds the number of children of the node. | **public** **int** rank |
| Holds the key of the node. | **public** **int** key |

**מדידות :**

* לגבי פעולות הכנסה של m איברים בסדר יורד :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Potential** | **totalCuts** | **totalLinks** | **Run-Time**  **(in miliseconds)** | **M** |
| **1000** | **0** | **0** | **0.275746** | **1000** |
| **2000** | **0** | **0** | **0.407978** | **2000** |
| **3000** | **0** | **0** | **0.705233** | **3000** |

זמן הריצה האסימפטוטי של סדרת פעולה זו כפונקציה של m הוא  **O(m)**

כיוון שפעולת insert אחת לוקחת זמן קבוע , כלומר O(1) ועשינו m פעולות כאלה , לכו בסה"כ O(m) . קיבלנו : totalCuts=totalLinks=0 מכיוון שפעולות אלה אינן מושפעות מפעולת insert כי פעולת ההכנסה היא Lazy, לכן בסכ"ה O(0) . לגבי הפוטינציאל :

Potential = number\_of\_trees + 2\* number\_of\_marked\_nodes = m+ 2\*0 = m --> O(m)

וזה מתאים לתוצאות שקיבלנו בטבלה 😊.

* לגבי פעולות deleteMin() :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Potential** | **totalCuts** | **totalLinks** | **Run-Time**  **(in milliseconds)** | **M** |
| **6** | **0** | 2882 | **7.5347149** | **1000** |
| **6** | **0** | 6773 | **13.366997** | **2000** |
| **7** | **0** | 10038 | **17.604039** | **3000** |

מן הריצה האסימפטוטי של סדרת פעולות **ההכנסה** כפונקציה של m הוא  **O(m)**,

זמן הריצה האסימפטוטי של m/2 פעולות deleteMin() - אחרי ההכנסה – הוא

**במקרה הגרוע** כי בצענו m/2 פעולות שכל פעולה עלולה לקחת O(m) במקרה הגרוע כאשר m הוא מספר האיברים בערימה . עבור זמן אמורטיזד זה יקח **.**

מספר פעולות Cut אינו תלוי בפעולות deleteMin,insert ולכן הוא שווה ל- 0 .

מספר פעולות Link שמתבצעות חסום ע"י m ולכן O(m) . עבור הפוטינציאל , הוא שווה ל- O(1) כי שלמנו אותו בפעולות deleteMin() שביצענו .