Name1: Tal Sofer

Id1: 323869842

Username1: talsofer1

Name2: Jonathan Klier

Id2: 323916395

Username2: klier

BinomialHeap Project Documentation

חלק מעשי:

***Class BinomialHeap():***

Fields:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| Size | Int | Heap size |
| Last | HeapNode | Pointing to the root of the largest tree |
| Min | HeapNode | Pointing to the node with the minimal key |

Methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Role** | **Description** | **Complexity** | **Complexity Analysis** |
| public BinomialHeap()  public BinomialHeap(HeapNode last, HeapNode min, int size) | Class Constructor | Initializes BinomialHeap’s fields |  | - |
| public HeapItem insert(int key, String info) | Inserts a new item to the Heap | Operates according to 3 different cases:   1. If the heap is empty, create a new node who is the last and min and size = 1. 2. If the heap’s size is even, connect a new node to the heap by updating its .next field to be the last’s next field, and update the last’s next field to be the new node 3. If the heap’s size is odd, create a new Binomial Heap whose only node is the new node we’re inserting. Meld this new Binomial Heap with the original Binomial Heap using Meld() function   The key and info will be updated in the node’s item. |  | Worst case calls the Meld() function which operates in . All other operations in the function operate in time. |
| public void deleteMin() | Delete’s the minimal key from the heap | Operates according to 2 different cases:   1. If the deleted node is the root of the only tree in the heap, it initializes the heap’s minimum and last fields according to the deleted node’s children’s minimum and last. 2. Otherwise, change the .next field of the last node before the deleted minimum to be the deleted minimum’s next (therefore disconnecting the minimum from the heap). Update the last and min fields accordingly, create a new Binomial Heap from the deleted minimum’s children, and meld this new heap with the original heap using the meldChildrenWithMainHeap(HeapNode n) function. |  | Worst case calls the Meld() function which operates in . Searching the deleted minimum’s children also operates WC in All other operation’s in the function operate in time. |
| private void meldChildrenWithMainHeap(HeapNode n) | For the case when the deleted node isn’t ‘alone’ in the main heap row, meld its children with the main heap | Called from the DeleteMin() function. Operates according to 2 different cases:   1. If the deleted minimum’s child’s .next field is itself, create a new BinomialHeap with the min and last fields as the deleted minimum’s child, size = 1, and meld with the original heap. 2. Otherwise, create a new Binomial Heap which its last field is the deleted minimum’s child, the min field is the minimum between the deleted minimum’s children, and the size of the new heap is , and meld this new heap with the original heap. |  | Worst case calls the Meld() function which operates in . All other operations in the function operate in time. |
| public HeapItem findMin() |  |  |  |  |
| public void decreaseKey(HeapItem item, int diff) |  |  |  |  |
| public void delete(HeapItem item) |  |  |  |  |
| public void meld(BinomialHeap heap2) |  |  |  |  |
| private int which\_case(HeapNode b\_node, HeapNode s\_node, HeapNode remainder , int exp) |  |  |  |  |
| private HeapNode[] check\_which\_nodes\_to\_concatenate(HeapNode b\_node, HeapNode s\_node, HeapNode remainder) |  |  |  |  |
| private void one\_node(HeapNode node, HeapNode ret\_node) |  |  |  |  |
| private HeapNode two\_nodes(HeapNode node1,HeapNode node2) |  |  |  |  |
| private HeapNode[] three\_nodes(HeapNode b\_node, HeapNode s\_node, HeapNode remainder) |  |  |  |  |
| private void merge\_remainder(HeapNode this\_node, HeapNode other\_node, boolean are\_we\_on\_heap\_2) |  |  |  |  |
| public int size() |  |  |  |  |
| public boolean empty() |  |  |  |  |
| public int numTrees() |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

***Class HeapNode():***

Fields:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| key | int | the numerical value of the node |
| value | string | the word inside the node |
| left | AVL\_Node | the node that's directly to the left of our node |
| right | AVL\_Node | the node that's directly to the right of our node |
| parent | AVL\_Node | the parent node |
| height | int | the maximum depth between our node and a leaf (0 if you're a leaf) |
| size | int | the size (number of sons) of our sub-tree |

Methods:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Description** | **Detailed Description** | **Complexity** |
| is\_real\_node() | returns True if the node is "real" (meaning it's not a node without a key and a value since each real node has two sons) | Checks whether the node's key is None. If so, return false, otherwise return true. |  |