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() | Finds the item with the minimal value | Goes around all of the roots (since they hold the smallest values of their respective tree) from last and sees which one of them has the minimal value | | O(log(n)) | There are at most log(n) (base 2) roots to go around |
| public void decreaseKey(HeapItem item, int diff) | Decreases the value of an item by given amount of diff | Lowers the value of the given item’s key, and while the item’s parent is larger than his switches their location, compares the lower value of the item with min and if it’s smaller then the item becomes the new min. | | O(log(n)) | Each tree is atmost log(n) (n is the total amount of items) deep so at the worst case that how much switches we have to do |
| public void delete(HeapItem item) | Deletes the given item from the heap | Does decrease key to make sure that the item goes to the root and then we call deletemin | | O(log(n)) | Decrease key and deletemin are both O(log(n)) |
| public void meld(BinomialHeap heap2) | Melds two heaps to a legal heap | First we divide the heaps to smaller one and a bigger one based on the largest tree’s size. If both are single trees with the same size we will connect them based on the root with the smallest value (with the help of two\_nodes and HeapNode.concatenate) . Otherwise, we create a new array of nodes when each cell i represents the i’th rank tree in the new heap, the nodes are being put in the array like adding binary numbers (with remainder and etc).  After were done building the array we create a new node chain based on the array and assign it to this.last. finally we update the heap size and min | | O(log(n)) | We only move O(1) times on on the roots which are with O(log(n)) proportions to the total amount of nodes/ |
| private int which\_case(HeapNode b\_node, HeapNode s\_node, HeapNode remainder , int exp) | Helps the meld function decide what to do with a numerical value | Gives a number based on how many trees at rank exp we currently have on bigger\_heap, smaller\_heap, and remainder | | O(1) | Does a finite amount of checks |
| private HeapNode[] check\_which\_nodes\_to\_concatenate(HeapNode b\_node, HeapNode s\_node, HeapNode remainder) | Return an array of the nodes sorted based which one we should move to the final heap and which one we should concatenate and put on the remainder | Creates an array of 3 nodes and sorts in so that the minimal valued node at the first index | | O(1) | Does a finite amount of checks. |
| private HeapNode two\_nodes(HeapNode node1,HeapNode node2) | Meld two trees and returns the melded tree | Check which one has the smallest root key and concatenates that one with the other using HeapNode concatenate | | O(1) | Does a finite amount of checks, concatenate is O(1) |
| private HeapNode[] three\_nodes(HeapNode b\_node, HeapNode s\_node, HeapNode remainder) | Returns a melded tree and the remainder | Gets 3 nodes, calls check\_which\_nodes\_to\_concatenate and figures out which node should it put in ret\_array and which ones to meld and put in the remainder and does | |  |  |
| 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. |  |