**תרגיל מעשי 2 – מבני נתונים**

ניתוח סיבוכיות זמן הריצה:

Fibonacci Heap Methods:

**Built-in Methods:**

* **public boolean isEmpty() –** the method checks if the heap is empty by checking if the number of nodes is 0, using the field "this.n". Therefore, complexity = O(1).
* **public HeapNode insert(int key) –** the method inserts a new key into the heap. We saw in the lecture that inserting a new key to a Fibonacci heap is done using the "lazy" method. Therefore in the method we just concatenate the new node to be the first on the left in the heap. Total complexity = O(1).
* **public void deleteMin() –** the method deletes the minimum key in the heap. First, the method checks if the heap is empty or contains only one node – if so, the fields are initialized with zeros/nulls and in this case it is O(1).

If the heap contains more than one node, we go over the heap with a loop – which cuts the min from its children. Afterwards we use a "buckets" array to which we insert each of the roots. Once a bucket slot contains two trees – we concatenate them in the binomial-form, using the method "meld\_array" of complexity O(1) (will be elaborated) and insert the joint tree into the next slot. The process is done until the bucket array has no slot with more than 1 tree, and after finishing going over all the trees in the heap. Finally we go over the bucket, using "retrieve\_from\_buckes" method (will be elaborated) with a complexity of O(log(n)), to restore the heap and set the "first" and "min" values. As we saw in the lecture, the amortized cost of the operation is O(log(n)) and could also be O(n) in worst-case (like when we have all singles nodes before deleting).

* **public HeapNode findMin() –** the function returns the node whose key is minimal. We use the field "this.min", therefore a complexity of O(1).
* **public void meld (FibonacciHeap heap2) –**  the method melds the current heap with another. First we check edge-cases when one of the heaps is empty – if so, the other is the joint. Else – we concatenate "heap2" with current using the "next" and "prev" pointers. We also check the updated min-value. Therefore a total complexity of O(1).
* **public int size() –** we return the field "this.n" which maintains the number of nodes in the heap. Therefore a complexity of O(1).
* **public int[] countersRep() –** the method maintains an array in which the value of the ith-entry is the number of trees of order i in the heap.first, we check whether the heap is empty – if so we return empty array. Otherwise, we find the max-rank using a loop over the trees in the heap, and we create an array in that size (O(max-rank)). Later, we go over the trees in a loop and add each tree's rank to its location in the array. At last – we return the array. Therefore a total complexity of O(n).
* **public void delete(HeapNode x) –** the method deletes the node x from the heap. in order to use this method, we use the method "decrease\_key" (will be elaborated) which decreases the key to be the min. this method has a complexity of O(1) amortized. After – we use the "delete\_min" method. Therefore a total complexity of O(log(n)) amortized, and O(n) worst-case.
* **public void decreaseKey(HeapNode x, int delta) –** the function decreases the key of the node x by delta. First we check if the decrease didn't "harm" the heap **–** which means the heap-rule (that each node is smaller than its children) wasn't break – if so we stop at that point. Else – we use the method "cascading-cut" (will be elaborated) which has amortized complexity of O(1) and worst-case of O(log(n)).Therefore a total complexity of O(1) amortized and O(log(n)) worst-case.

**Added Methods:**

* **public void retrieve\_from\_buckets(HeapNode[] array\_of\_buckets)** – the method is a helper method used for "delete\_min". in this method we go over the bucket-array generated in delete-min, and concatenate the trees in the bucket – back to a heap.

we have a complexity of O(log(n)) due to the fact that in the original heap there were n trees, and after committing the delete\_min and processing the "meld\_array" method for joining them – we have approximately log(n) trees after the process, as proved in class.

* **public void cascading\_cut(HeapNode node) –** the function cuts (using "cut" method which will be elaborated) x from the tree and checks if its parent need to be cut too (and so on recursively) due to marking.if the process is done in a single tree from the bottom up to the top – we did a log(n) in worst-case. So we have a total complexity of O(1) amortized (as taught in the lecture) and O(log(n)) worst-case.
* **public void cut(HeapNode node) –** themethod cuts the given node. First we check if the node was marked – if so, we reduce the number of marks and change its mark to false. After checking if the node has neighbors/not, we update the pointers of the node's neighbors, parent, and update first to be node. Therefore a total complexity of O(1).
* **public HeapNode meldArray(HeapNode node\_1, HeapNode node\_2) –** the function checks which of the given nodes is higher (by key) that the other. By the definitions we made using "high" and "low" – we connect the two nodes forming a new joint tree, and concatenate the children. Therefore a total complexity of O(1).
* **public int potential() –** the method returns the current potential of the heap using #trees + 2\*#marked. Therefore we use the fields "this.number\_of\_trees" and "this.number\_of\_marked". So a total complexity of O(1).
* **public static int totalLinks() –** this static function returns thetotal number of link operations made during the run-time of the program. We use the static field "total\_links" so a total complexity of O(1).
* **public static int totalCuts() –** this static function returns thetotal number of cuts operations made during the run-time of the program. We use the static field "total\_cuts" so a total complexity of O(1).
* **public static int[] kMin(FibonacciHeap H, int k) –** this static function returns the k minimal elements in a binomial tree H.

First, we check if k is 0 or 1 – if so we return an empty array (case 0) or an array with min-key (case 1). Otherwise, we initialize an array of size k and also create a new heap (k\_min\_heap). By generating the new heap we insert H's min to it, and then we go over a loop k-times and at each iteration we delete-min in the new heap. we also verify edge-case for the children of each node. The process is done when the loop ends.

The complexity is O(k\*deg(H)) due to the fact that we go over a loop k times – each time checking for children, (rank/deg) and if so inserting them too.

HeapNode methods

**Built-in methods:**

* **public int getKey()** – the method returns the key of the current node using the field "key". Therefore a total complexity of O(1).