**Practical Exercise 2 – Fibonacci Heap**

**Ido Kessler , Shachar Hirshberg ( 311398499, 305624207)**

**Documentation:**

1. **public** **class** **FibonacciHeap**

An implementation of a Fibonacci Heap.

Class variables:

1. **static** **public** **int** *TotalLinks*
2. **static public int** TotalCuts
3. **public int** NumberOfTrees
4. **public int** NumberOfMarkedNodes
5. **private HeapNode** HeapNode\_Min
6. **private int** size
7. **public boolean empty()**

@complexity :O(1).

@return true if and only if the heap is empty.

@pre none.

@post none.

1. **public HeapNode insert(int key)**

Creates a node (of type HeapNode) which contains the given key, and inserts it into the heap.

@complexity O(1) - add new node as a tree regardless of heap size.

@pre key>=0.

@post the heap now contain a new node with this key.

@return the new heap node created.

1. **public void deleteMin()**

Delete the node containing the minimum key.

@dependencies - delete - Amortized O(log(n))

@complexity - Amortized O(log(n))

@post - tree is valid and doesn't contain the previous minNode

1. **public HeapNode findMin()**

@return the node of the heap whose key is minimal.

@complexity O(1)

@pre - min exists

1. **public void meld (FibonacciHeap heap2)**

Meld the heap with heap2

@complexity O(1) - connect the ends of two double link lists.

@pre none.

@post the two heaps are now combined.

1. **public int size()**

Returns the number of elements in the heap

@complexity O(1) - return a value which updated regularly.

@pre none.

@return number of nodes in tree.

1. **public int[] countersRep()**

@return a counters array, where the value of the i-th entry is the number of trees of order i in

the heap.

@dependencies getMaxRank.

@complexity O(n) - worst case run over all 0-rank tree nodes in the heap.

@pre none.

1. **private int getMaxRank()**

@return the maximum rank in heap.

@pre heap is not empty.

@complexity O(n) - worst case run over all 0-rank tree nodes in the heap.

1. **public void arrayToHeap(int[] array)**

Insert the array to the heap. Delete previous elements in the heap.

@complexity O(n) - see details in arrayToBinomialHeap.

@dependencies nodesArrayToHeap - O(n), arrayToBinomialHeap - O(n).

@pre none.

@post heap now contain all items in array.

1. **public void arrayToBinomialHeap(int[] array)**

Insert the array to the Binomial heap.

@dependencies getBitLength - O(log(n)).

@complexity O(n) - go over each item of the array with amortized time of insert action to

binomial heap of O(1).

@pre array != empty or null.

@return binomial heap with the items in the array.

1. **private void nodesArrayToHeap(HeapNode[] Heaps\_Arr)**

Insert the array to the heap. Delete previous elements in the heap.

@complexity O(n) - go over the array and connect each item to a double linked list.

@pre Heaps\_Arr != empty or null.

@post heap contain all the tree that are in the array. Integers "size" and

"numberOfMakedNodes" doesn't change.

1. **public void getBitLength(int k)**

@return the number of bit takes to represent k.

@complexity O(log(k)) - how many character it takes to represent the number.

@pre none.

1. **private void delete(HeapNode x)**

Deletes the node x from the heap.

@param - HeapNode x an HeapNode.

@param - boolean isMin is this deleteMin.

@complexity - Amortized O(log(n))

@dependencies - delete(HeapNode x, boolean isMin) - Amortized O(log(n))

@pre HeapNode x exists

@post HeapNode doesn't exist in the heap anymore and the heap is valid.

1. **private void delete(HeapNode x, boolean isMin)**

Deletes the node x from the heap.

@param - HeapNode x an HeapNode.

@param - boolean isMin is this deleteMin.

@complexity - Amortized O(log(n))

@dependencies - mergeNodesList - O(1), removeNodeFromNodesList - O(1), cascadingCut –

O(log(n)), consolidate - O(log(n))

@pre HeapNode x exists

@post HeapNode doesn't exist in the heap anymore and the heap is valid.

1. **public void decreaseKey(HeapNode x, int delta)**

The function decreases the key of the node x by delta. The structure of the heap should be

updated to reflect this change (for example, the cascading cuts procedure should be applied if

needed).

@dependencies cut - O(1) , cascadingCut - O(log(n))

@complexity O(log(n))

@pre HeapNode x exists, delta >= 0.

@post the heap is now updated according to the new key.

1. **public void cut(HeapNode node)**

The function cuts the node node from its location.

@dependencies removeNodeFromNodesList - O(1), mergeNodesList - O(1)

@complexity O(1)

@pre none.

@post the node is no longer linked to its parent.

1. **public void cascadingCut(HeapNode node)**

The function performs a cascading cut.

@dependencies cut - O(1) , cascadingCut - O(log(n))

@complexity - O(log(n))

@pre HeapNode node exists

@post cascading cut was performed, if needed.

1. **public void mergeNodesList(HeapNode node, HeapNode node2)**

The function links the node node to another node, node2.

@dependencies - none

@complexity - O(1)

@pre - HeapNode node and node2 exists.

@post - nodes are now connected.

1. **public void mergeTree(HeapNode tree1, HeapNode tree2)**

Adds a new child for an HeapNode object.

@param HeapNode newChild - the new child.

@complexity O(1)

@pre none.

@post the new child is added now to the father node.

1. **public void removeNodeFromNodesList(HeapNode node)**

The function removes the node from siblings linked list and from parent.

@dependencies

@complexity - O(1)

@pre HeapNode node exists

@post HeapNode node is no longer linked to its siblings and parent.

1. **public void consolidate()**

Combine every tree of the same size.

@dependencies nodesArrayToHeap - O(n), getMaxRank - O(n). getBigLength - O(1).

mergeTree - O(1)

@complexity O(n) - worst case go over all 0-rank trees in the heap.

@pre NumberOfTrees is correct.

@post the heap now doesn't contain 2 tree of the same rank.

1. **public int potential()**

The potential equals to the number of trees in the heap plus twice the number of marked

nodes in the heap.

@returns the current potential of the heap, which is: Potential = #trees + 2#marked

@complexity O(1) - use value which are regularly being updated.

@pre none.

1. **public static int totalLinks()**

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.

@returns the total number of link operations made during the run-time of the program.

@complexity O(1) - use value which are regularly being updated.

@pre none.

1. **public static int totalCuts()**

A cut operation is the operation which disconnects a subtree from its parent (during

decreaseKey/delete methods).

@returns the total number of cut operations made during the run-time of the program.

@complexity O(1) - use value which are regularly being updated.

@pre none.

1. **public class HeapNode**

An implementation of a Fibonacci Heap Node.

Class variables:

1. public int key
2. public boolean isMarked
3. public HeapNode nextNode
4. public HeapNode prevNode
5. public HeapNode parentNode
6. public HeapNode child
7. private int rank
8. **public HeapNode(int k)**

Constructor for a HeapNode object.

@param k - key of the node.

@complexity O(1)

@pre none.

**Measurements:**

**Part 1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *m* | *Run Time(ms)* | *totalLinks* | *totalCuts* | *Potential* |
| 1000 | 6 | 0 | 0 | 1000 |
| 2000 | 10 | 0 | 0 | 2000 |
| 3000 | 11 | 0 | 0 | 3000 |

**Explanations:**

**Run time:**

**Total Links:** Since we only inserted into the heap we would have a heap with m HeapNodes each of rank 0. We never connected anything since nothing was deleted, and accordingly no links were made.

**Total Cuts:** Since we only inserted into the heap we would have a heap with m HeapNodes each of rank 0. We didn’t delete anything, and accordingly no cuts were made.

**Potential:** Since we only inserted into the heap we would have a heap with m HeapNodes each of rank 0. No cuts were made so we would expect to get potential of m trees, and this is indeed what we get.

**Part 2:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *m* | *Run Time(ms)* | *totalLinks* | *totalCuts* | *Potential* |
| 1000 | 8 | 1891 | 0 | 6 |
| 2000 | 14 | 5780 | 0 | 6 |
| 3000 | 20 | 11552 | 0 | 7 |

**Explanations:**

**Run time:**

**Total Links:**

**Total Cuts:** Since we only deleted the minimum Node of the heap, which is always a root, no cuts were made. And indeed we get zero total cuts.

**Potential:**