Work was done by:

Full name: viktor karsakov (ויקטור קרסקוב)	Asaf Amitai (אסף עמיתי)
I.d. number: 205727183	207688367
university user name: viktork	asaf

Binomial Heap documentation

internal class – HeapNode

variables -

- value: holds the value of the node (no default value)
- HeapNode `theMostLeftChild` : pointer to the node's most left child (default null)
- HeapNode `leftBrother`: pointer to the node's most close left brother (default null)
- HeapNode `rightBrother`: pointer to the node's most close right brother (default null)

methods -

`private HeapNode(int value)` -

builder for the HeapNode class.

main class – BinomialHeap

variables -

-`private boolean empty`: true if the binomial heap is empty
-`private int size`: holds the size of the binomial heap
- `public HeapNode min`: pointer to the minimum heap node
- `public HeapNode[] HeapTreesArray`: array that holds all the trees in the binomial heap

methods -

`public boolean empty()` -

returns true if the heap is empty.
It's done on O(1) since we are using a variable that holds that information.

`public void insert(int value)` -

inserts a value to the heap. If the heap is empty, we just add a new node to the first place.
Otherwise, we create a new (empty) binomial heap, and the new node to that heap. And then, we meld the new and the old heaps.
As it was discussed in class, this operation cost O(1) amortized, and O(logn) in a worst-case analysis, since this operation is analogous to the problem of binary counter. (adding a new node, is similar to adding 1 to the binary counter).

`public void deleteMin()` -

since we have a pointer to the minimum node, we can immediately reach to it. We go through all of the minimum's children, and add them to a new heap (since we know the rank of each child, adding a specific child to the new heap takes only O(1)).
Then, we meld the two heaps. Melding is done in O(logn) by the worst-case analysis.
So in conclusion, deletemin() takes O(logn) time in the worst case.

`public int findMin()-`

returns the value of the minimum node. Done in O(1) since we have a pointer to the minimum.

`public void meld(BinomialHeap heap2) -`

melds to binomial heaps together. It is done in O(logn), as we learned in class. Uses the merge() method.

`public HeapNode merge(HeapNode hN1, HeapNode hN2) -`

doing the linking between two trees. The tree with the smallest root, is the father. Done in O(1).

`public int size() -`

returns the size of the heap. We hold the variable size, so it is done in O(1).

`public int minTreeRank() -`

returns the minimum rank of the binomial heap. The method go through the array of the trees, and returns the index of the first place in the array, that is not null. It is done in O(logn) because the length of the array is log(n).

`public boolean[] binaryRep() -`

returns the binary representations, of the heap, by going over the array that holds the trees. Done in O(logn) since the length of the array is logn.

`public void arrayToHeap(int[] array) -`

inserts all the values in the array to the heap. It is done in O(n), since insert() done in O(1) amortized, and there are n values.

`public boolean isValid()  -`

returns true, if and only if, the heap is valid. Be informed that it is an expensive operation! And it might cause a stack overflow!

`public int countRank(HeapNode h) -`

counts the rank of the node.

`public boolean heapRule(HeapNode h)` -

checks if the heap rule is satisfied in a tree.

`public boolean checkNumChild(HeapNode h, int i)` -

recursive function that check that the number of children of a node, is ok.

The Experiment

answers -

1. we can do insert(), until we get to log (m') inserts, when m' is the closest number to m that is a power of 2, and m'<m. Then, we do (m – log(m') - 1) deletes and inserts one after the other, which cost only O(1) for each operation, since it is only adding a new node to the beginning of the array, and the beginning of the array is empty. At the end, we add to the array two nodes in a row, which ends in an expensive operation of log(m). [going form the binary number 111111...11111 to 10000...00000].

2. at the beginning every insert takes O(1) in average. At the end, we add and delete nodes in O(1). The total amount of inserts and deletes, is m-1. The final operation (insert) takes O(logm). So the total cost is m-1*O(1) + O(logm) = O(m).

3. for 1000, the closest number that is a power of 2, and is less then 1000, is 512. so we do, 511 inserts, and then we 488 deletes and inserts on after another, and then we do on last insert which cost O(logm). So - (i) since log2(512) Is 9 – after the m-1 opertation we get 111111111
(ii) after the m opertation we get 1000000000.

for 2000: the closest number is 1024, which is 2^10. so
(i) after the m-1 operation we get 1111111111
(ii) after the m operation we get 10000000000

for 3000: the closest number is 2048 which is 2^11. so
(I) after the m-1 operation, we get 11111111111
(ii) after the m operation we get 100000000000

running the program -

m	Number of linking	Binary reprsentation after m-1 operations	Binary reprsentation after m operations
1000	519	true true true true true true true true true	false false false false false false false false false true
2000	1032	true true true true true true true true true true	false false false false false false false false false false true
3000	2057	true true true true true true true true true true true	false false false false false false false false false false false true

Explanation of the results:

as you can see in the table above, the binary representation that we got, was exactly what we thought it will be (true denotes, 1, and false denotes 0).

regarding the Number of linking - we also got what we roughly expected. For the first m' operations (m' is defined above) we get O(m') linking, because on the average, every operation is done in O(1). So O(m') = m' * O(1).
For the last m – m' operation we got also O(m-m') linkings. Because each operation is exactly O(1), and O(m-m') = (m-m')*O(1). And at last, the final operation cost us O(logm).

If we sum up all the costs of the operations, we get that the total number of linking that we made, was: O(m) = O(m') + O(m-m') + O(logm).

And indeed, from looking at the table, we see exactly that: #O(1000) = 519;
#O(2000) = 1032;
#O(3000) = 2057;
which are all corect.

side note -
we got the number of linking using a static counter. We marked the static counter, and the use of him as comment (in the code). So just ignore it (when you read the code).