**class FibonacciHeap:**

public HeapNode getFirst()

Return the First (most left and newest) node in the heap.

Complexity: O(1)

public int getSize()

Return the number of nodes in the heap

Complexity: O(1)

public int getNumTrees()

Return the number of trees in the heap.

Complexity: O(1)

public int getCountMarkNodes()

Return the number of mark nodes.

Complexity: O(1)

public boolean isEmpty()

Returns true if and only if the heap is empty.

Complexity: O(1)

private replaceMin(HeapNode node)

If argument node has minimal key, update min attr in Heap.

@pre: node is in heap, node.getParent() == null (node is root node).

Returns true if min got replaced, false otherwise.

public HeapNode insert(int key)

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

The added key is assumed not to already belong to the heap.

Help functions: replaceMin

Complexity: O(1)

Returns the newly created node.

private void removeMinNode()

Only used for deleteMin. Remove the node with minimal key, add and connect its children as heap roots by nullifying parent connection and connecting them to the main root chain; reset their marked status.

@pre: there is more than 1 node in Heap.

Help functions: HeapNode.resetMarkedInChain, HeapNode.nulifyParentInChain, HeapNode.insertBefore

Complexity: O(m), m - number of subtrees of root with minimum key (m = O(logn))

private HeapNode consolidateConnect(HeapNode node1, HeapNode node2)

Only used for consolidate. Add the node with the bigger key as left-most child of node with the smaller key.

Complexity: O(1)

Returns the HeapNode with the smaller key

public void consolidate()

Consolidate trees by linking them with so that we will have O(log(n)) trees, each with a different rank, using the "buckets" method we saw in lecture. First, we create an array of buckets that each will hold a tree with a different rank. We go over every tree and insert it by it's rank to the appropriate bucket, if there is a tree in that bucket we connect them using consolidateConnect and move it to the next bucket. After consolidation process is finished, we go over the bucket array, collect and connect the remaining trees while determining which root has the minimum value.

Help functions: HeapNode.resetMarkedInChain, HeapNode.nulifyParentInChain, HeapNode.insertBefore

Complexity: O(k-1+m), k - number of trees in heap (before deletion of min used prior) / m - number of subtrees of root with minimum key (m = O(logn))

public void deleteMin()

Deletes the node containing the minimum key.

First, we remove the minimum key with removeMinNode;

Then we consolidate and connect all trees using the method we saw in lecture with consolidate function, while determining which key has the minimum value.

Help functions: removeMinNode, consolidate

Complexity: O(k-1+m), k - number of trees in heap / m - number of subtrees of root with minimum key (m = O(logn))

public HeapNode findMin()

Returns the node of the heap whose key is minimal, or null if the heap is empty.

Complexity: O(1)

public void meld (FibonacciHeap heap2)

Melds heap2 with the current heap.

Help functions: FibonacciHeap.insertBefore, replaceMin

Complexity: O(1)

public int size()

Returns the number of elements in the heap.

Complexity: O(1)

public int[] countersRep()

Return an array of counters. The i-th entry contains the number of trees of order i in the heap.

(Note: The size of of the array depends on the maximum order of a tree.)

Help functions:findMaxRank()

Complexity: O(n)

public int findMaxRank()

Return the max rank of the all trees in the heap

Complexity: O(n)

public void delete(HeapNode x)

Deletes the node x from the heap.

It is assumed that x indeed belongs to the heap.

Complexity: O(n)

public void decreaseKey(HeapNode x, int delta)

Decreases the key of the node x by a non-negative value delta. The structure of the heap should be updated to reflect this change (for example, the cascading cuts procedure should be applied if needed).

Help functions:cascadingCut()

Complexity:O(log(n))

public int nonMarked()

This function returns the current number of non-marked items in the heap

Complexity:O(1)

public int potential()

This function returns the current potential of the heap, which is:

Potential = #trees + 2#marked

In words: The potential equals to the number of trees in the heap

plus twice the number of marked nodes in the heap.

Complexity:O(1)

public static int totalLinks()

This static function returns the total number of link operations made during the

run-time of the program. 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 under the other tree.

Complexity:O(log(n))

public static int totalCuts()

This static function returns the total number of cut operations made during the

run-time of the program. A cut operation is the operation which disconnects a subtree

from its parent (during decreaseKey/delete methods).

Complexity:O(1)

public void cascadingCut(HeapNode x,HeapNode xParent)

This is recursive function!

This function continues to make cascading cuts as long as the parent tree is marked.

Help functions:cut()

public void cut(HeapNode x,HeapNode xParent)

Cuts node x from xParent , xParent its x's parent and adds it as a new tree.

Help functions:replaceMin()

Complexity:O(1)

public static int[] kMin(FibonacciHeap H, int k)

This static function returns the k smallest elements in a Fibonacci heap that contains a single tree.

The function should run in O(kdeg(H)). (deg(H) is the degree of the only tree in H.)

Help functions:isEmpty(),deleteMin(),findMin(),insert(),

Complexity:O(kdeg(H))

**class HeapNode:**

public HeapNode(int key)

Initializing heap node

Complexity: O(1)

public int getKey()

Return key of the node

Complexity: O(1)

public void setKey(int k)

Set k as the node's key

Complexity: O(1)

public int getRank()

Return the rank of the node

Complexity: O(1)

public void setRank(int k)

Set k as the node's rank

Complexity: O(1)

public boolean getMarked()

Return true if the current node is marked else return false.

Complexity: O(1)

public void setMarked(boolean TF)

Set true if the node is marked else set false.

Complexity: O(1)

public HeapNode getChild()

Return the child of the node.

Complexity: O(1)

public void setChild(HeapNode node)

Set node as the child of the current node.

Complexity: O(1)

public HeapNode getParent()

Return the parent of the node.

Complexity: O(1)

public void setParent(HeapNode node)

Set node as the parent of the current node.

Complexity: O(1)

public HeapNode getNext()

Return the next node of the current.

Complexity: O(1)

public HeapNode getFirst()

Set node as the next node of the current.

Complexity: O(1)

public HeapNode getPrev()

Return the previous node of the current.

Complexity: O(1)

public void setPrev(HeapNode node)

Set node as the next node of the current.

Complexity: O(1)

public HeapNode getKMinPointer()

Return the KMinPointer. We use this in the function kMin(FibonacciHeap H, int k).

Complexity: O(1)

public void setKMinPointer(HeapNode node)

Set node as the KMinPointer of the current.

Complexity: O(1)

private void insertBefore(HeapNode node)

Add node x as a left sibling to instance node.

Complexity: O(1)

private void insertBefore(HeapNode node1, HeapNode node2)

Add a chain of nodes and connect them to instance node. The last node in the chain (node2) will be the left sibling of instance node.

Complexity: O(1)

private int nulifyParentInChain()

Go over instance node and all it's sibling and change attribute 'parent' to null.

Returns the number of nodes in chain (k+1).

Complexity: O(k+1), k - number of siblings instance node has.

private int ResetMarkedInChain()

Go over instance node and all it's sibling and change attribute 'mark' to 'false'.

Returns the number of nodes in chain in which 'mark' was set to 'true'.

Complexity: O(k+1), k - number of siblings instance node has.