Runtime analysis, Classes, and functions' documentation:

Class: BinomialHeap

Description**:**

Represents a Binomial Heap. Every element has the following fields:

> **subTreeArray -** an arraylist containing the subtrees of the heap. Every root of every subtree will be the smallest key in the subtree.
>
> **Min -** a pointer to the node with the smallest key in the heap. This node is the root of one of the heap's subtrees.
>
> **Last -** a pointer to the root of the largest subtree of the heap which is the last element in the subtrees list.
>
> **Size -** total amount of nodes in the heap.
>
> **Treescount -** the total amount of subtrees in the heap. We will notice that not all of the subtrees of a heap are always existent, and therefore the size of the array won't necessarily represent the total amount of subtrees.

Functions:

public HeapItem insert*(int key, String info)* **|** $\boldsymbol{O(log(n))}$

First, we will create a HeapItem and HeapNode with the input elements. Then, we will check the current state of the heap, and act accordingly:

1. If the heap is an empty heap, or there is no subtree or rank 0 then we will add the HeapNode as the first element in the subTreesArray.
2. If the heap is not an empty heap, and exists a subtree of rank 0 then we will call the helper function **singleMeld** with this new HeapNode and the index 0.

Next, we'll increase the size of the heap by one, and update the min pointer if needed.

In the worst case, the heap is not empty, and all of it's subtrees are full, and therefore we will have to call the singleMeld function. In this case we have $\lceil Logn \rceil$ subtrees, so the runtime of singleMeld will be $O(\log(n))$. All of the other actions in this function are conditions and pointers update, which takes $O(1)$ time. In conclusion, summing up all the actions will give us the runtime analysis of $\boldsymbol{O(log(n))}$**.**

public void deleteMin*()* | $O(log(n))$

First, we will create a new subtreesArray which consists of the children of the minimal node of the current heap. Next, we'll create a new heap element using this array, while noticing this is not a real heap representation, therefore we won't update all of the pointers.

Then, we'll check if the minimum node is the root of the largest subtree. If so, we'll remove this node. Otherwise, we'll set the element in this index to be null.

After disconnecting the minimum node, we'll call we'll and **updateLast** in order to update the heap's pointers. In the end we'll meld the original heap without the minimum node with the new heap that we created from its children.

In the worst case, the minimum node is the root of the largest subtree, the new minimum will be the second largest subtree, and all of the ranks are full. In this case, updateMin will take $O(log(n))$ because we will iterate over all the roots of the other subtrees until we will reach the minimum.

Then, we will call the function **meld** with two heaps, which are the same size. Therefore, we will have to meld all of the subtrees. We will meld $O(log(n))$ subtrees, and each meld takes $O(1)$**.** These two parts happen one after the other, so we will sum it up, and will get $O(log(n))$ runtime.

public void updateLast*()* | $O(1)$

We will check if the heap is an empty tree or not. If not, we will return the last root in the subTreesArray.

public void updateMin*()* | $O(log(n))$

We will iterate over all of the roots of the subtrees in the subTreeArray and compare their value until we find the minimal root.

If the minimal node is the root of the largest tree, then we'll iterate over all of the subtrees. If all of the subtrees of the heap exist then, we will perform $log(n)$ iterations, so the runtime of the function will take $O(log(n))$**.**

public HeapItem findMin*()* | $O(1)$

Returning the value of the min pointer.

public void decreaseKey*(HeapItem item, int diff)* **|** $\boldsymbol{O(log(n))}$

First, we'll calculate the new key of the Item. Then, we'll iterate up in the tree as long as there exists a parent, and that the parent's key is bigger than the new key we calculated. Every step we take, we'll switch between the item of the current parent we are looking at, and the item of the current node we updated.

After arriving to the top or to a parent with a smaller key, we'll call the functions **updateMin** and **updateLast**, to make sure that these two pointers are up to date.

In the worst case, the runtime will be $\boldsymbol{O(log(n))}$**.** In this case, we will get a leaf of the largest tree in the heap, and we'll need to iterate all the way to its root. The largest tree is $\boldsymbol{O(log(n))}$'s height, so we'll do $\boldsymbol{O(log(n))}$ iterations and pointers update.

In addition, as described earlier, the function **updateMin** might take up to $\boldsymbol{O(log(n))}$ as well. These two parts are occurring one after the other, so the total runtime will be $\boldsymbol{O(log(n))}$**.**

public void delete*(HeapItem item)* **|** $\boldsymbol{O(log(n))}$

In order to delete a given node, we'll calculate the difference between this key and the minimum key and will enlarge it to make sure that the new key after the decreaseKey function will be the new minimum. After calculating the required difference, we'll call the function **decreaseKey**. After this function we know that this item is the new minimum of the heap, so we'll now call the function **deleteMin**.

As mentioned earlier, the functions **decreaseKey** and **deleteMin** can take up to $\boldsymbol{O(log(n))}$ each, so the total runtime of delete will be $\boldsymbol{O(log(n))}$ as well.

public void meld*(BinomialHeap heap2)* **|** $\boldsymbol{O(log(n))}$

First, we'll check the new heap's size and will act accordingly.

1. If the heap is empty, then we do nothing, because we have nothing to meld with.
2. If the heap is only one node, then we insert this node by using the insert function. In the insert function we know that all of the relevant pointers will be updated.
3. Else, the heap has two or more nodes. In this case we'll iterate over all of the subtrees of the new heap, and will meld each with the current heap we have. After melding, we will update the minimum of the heap if needed.

   At the end of the function, we will call the function **updateLast** in order to make sure that this pointer is updated.

In the worst case we are at situation number 3, and we have to meld between two heaps of the same rank. A full chain of melding will occur only once due to the design of the binomial heaps. Therefore, in the end we will have executed one full chain meld that

crossed all ranks. All of the other inserts will cost us $O(1)$. This can happen for example when both trees are full at all ranks. Melding the first rank 0 tree with the heap will cause a chain of melds at the cost of $O(log(n))$. After that inserting all other ranks will be $O(1)$ since those ranks at the original heap are now empty.

public void singleMeld*(HeapNode newNode, int indexStart)* **| $\boldsymbol{O(log(n))}$**

This is a helper function, that will take a subtree and an index, and will merge this subtree with the current heap, starting from the index that represents the required location of this subtree.

In the worst case, we have a full heap, and we'll merge a single node into it. In this case, every meld will create a new subtree in a size that already exists, so we'll have to go over all of the subtrees and meld them, and create a new subtree. After going over all of the subtrees, we'll check if the current index is a valid index using the function **isIndexLegal**, which checks if the index is in the middle of the subarray or not. According to the function result, we'll decide if we have to insert or update the relevant index in the array.

In this case we will iterate $\boldsymbol{O(log(n))}$ times, and each iteration we'll call the function **uniteNodes** which updates the pointers. So, in each iteration we will do $O(1)$ actions, and the total runtime of this function will be $\boldsymbol{O(log(n))}$**.**

public boolean isIndexLegal*(int index)* **| $\boldsymbol{O(1)}$**

We will compare the input index to the subarray's size.

public void uniteNodes*(HeapNode minNode, HeapNode otherNode)* **| $\boldsymbol{O(1)}$**

Helper function updates all of the pointers of the two nodes and creates a new subtree out of it. This function will make sure that all of the children of the node will be connected in a circular way. By calling this function, we make sure that the parent will be the node with the smaller key.

public int size*()* **| $\boldsymbol{O(1)}$**

Returns the size of the heap by using the size element.

public Boolean empty*()* **| $\boldsymbol{O(1)}$**

Returns True if the size of the heap is 0, False otherwise.

public int numTrees*()* **| $\boldsymbol{O(1)}$**

Returns the total count of the subtrees using an element that is updated every time we create or delete a subtree (during melding).

Class: HeapNode

Description**:**

This class is representing the HeapNode, which is a node in the subtree. Each node has the following fields:

Item - a HeapItem element that contains the key and the info of the node. By the item's key we can determine the node's position in the subtree.

Child - the last inserted child of the node. The child's key is larger than the node's key.

Next - other child of the node's parent. We can't assume anything about the key's order.

Parent - a HeapNode with a smaller key that represents the parent in the subtree.

Rank - represents how many children the node has.

Functions:

public int getKey*()* | $O(1)$

Returns the item's key.

public String getInfo*()* | $O(1)$

Returns the item's info.

public HeapItem getItem*()* | $O(1)$

Returns the item.

public HeapNode getChild*()* | $O(1)$

Return's the node's child using the pointer.

public HeapNode getNext*()* | $O(1)$

Returns the node's next using the pointer.

public HeapNode getParent*()* | $O(1)$

Returns the node's parent using the pointer.

public int getRank*()* | $O(1)$

Return the node's rank using the pointer.

public void setItem*(HeapItem myItem)* **|** $\boldsymbol{O(1)}$

Set's the item pointer of the node to be the input item.

public void setChild*(HeapNode myChild)* **|** $\boldsymbol{O(1)}$

Set's the child pointer of the node to be the input node.

public void setNext*(HeapNode myNext)* **|** $\boldsymbol{O(1)}$

Set's the next pointer of the node to be the input node.

public void setParent*(HeapNode myParent)* **|** $\boldsymbol{O(1)}$

Set's the parent pointer of the node to be the input node.

public void setRank*(int myRank)* **|** $\boldsymbol{O(1)}$

Set's the node's rank to be the input.

Class: HeapItem

Description**:**

This class represents HeapItem, which is the element that contains the key and the value we want to keep. Each Item has the following fields:

Key - comparable type, allows to determine the position of the item.

Info - the data the item contains.

Node - a pointer to the node that contains this specific item.

Functions:

public int getItemKey*()* **|**$\boldsymbol{O(1)}$

Returns the item's key.

public String getItemInfo*()* **|**$\boldsymbol{O(1)}$

Returns the item's info.

public void setNode*(HeapNode myNode)* **|**$\boldsymbol{O(1)}$

Sets the node's pointer to be the input.

public HeapNode getNode*()* **|**$\boldsymbol{O(1)}$

Returns the item's node.