Yoav Ben-Guigui-206570707 Gal Shani-322292210
Runtime analysis, Classes and function documentation:

Class: HeapNode

Description:

Represents the individual nodes contained within the Fibonacci Heap and includes the following attributes:

`Key` - The key of the node

`Info` - A string that represents additional information associated with the node.

`Child` - A pointer to the first child node of the current node.

`Next` - A pointer to the next node in the circular doubly linked list.

`Prev` - A pointer to the previous node in the circular doubly linked list.

`Parent` - A pointer to the parent node (if the node is part of a tree).

`Rank` - The number of children the node has.

`Mark` - Indicates whether the node has been disconnected from its parent once before.

Constructor | O(1):

The constructor initializes attributes such as key, info, child, next, prev, parent, rank, and mark through direct assignments. Since there are no complex computations or loops involved, the time required to execute these actions is independent of the input size or any external parameters. The constructor consistently performs a fixed number of operations for each node, with each initialization or assignment having a complexity of O(1). Therefore, the total complexity of the constructor is O(1).

Class: FibonacciHeap

Description:

Represents the Fibonacci Heap itself and includes the following attributes:

min - A pointer to the node with the minimal key in the heap

sizeVar - The number of nodes in the heap.

totalLinksVar - Counts how many links have been performed in the heap.

totalCutsVar - Counts how many cuts have been performed in the heap.

numTreesVar - The number of trees in the root list of the heap.

Constructor | O(1):

Creates an empty heap and initializes three fields. Each initialization is performed in constant time, so the overall complexity is O(1).

insert(int key, String info) | amortized cmplexity O(1) | worst case complexity O(1):

Description:

The insert function creates a new node using the HeapNode constructor, initializing its pointers to point to itself, forming a circular list. If the heap is empty, the new node becomes both the minimum and the only node in the heap. Otherwise, the new node is added to the root list using the helper function insertNodeToRootList, and the minimum is updated if the key of the new node is smaller. The function then increments the heap's numTreesVar (number of trees) and sizeVar (number of nodes). Finally, the newly added node is returned.

time complexity:

worst case - In the worst case, the function simply creates a new node and performs a fixed number of operations: initializing the node, updating pointers, adding it to the root list, and possibly updating the minimum pointer. These steps are all constant-time operations, independent of the size of the heap.
than the total complexity is O(1).

amortized complexity – remains the same because worst case is O(1).

findMin() | amortized cmplexity O(1) | worst case complexity O(1):

Description:

The findMin method is designed to retrieve the node with the smallest key in the Fibonacci Heap without modifying the heap. It works by simply returning the min pointer, which always points to the root node with the smallest key. Since the min pointer is updated during operations like insert, deleteMin, and decreaseKey, the method is efficient and does not require any traversal or computation. If the heap is empty, the method returns null.

time complexity:

worst case - The findMin method operates with a time complexity of O(1) as it simply returns the pointer to the min node. This involves a single operation with no loops, recursion, or time-dependent calculations. Since the min pointer is actively maintained during other operations, retrieving the minimum element is always efficient and constant in both worst-case and amortized scenarios.

amortized complexity- same as the worst case complexity as we explained.

deleteMin() | amortized complexity O(log n) | wost time complexity O(n):

Description:

The deleteMin function removes the node with the smallest key in the Fibonacci Heap. It first checks if the heap is empty, and if so, exits without performing any operations. If the heap is not empty, the function handles the minimal node's children by detaching them and adding them to the root list of the heap, updating

each child's parent pointer to null. This is done through iteration over the children, merging their circular list into the root list. Afterward, the minimal node is removed from the root list, and the counters are updated. If the heap becomes empty after removing the minimal node, the min pointer is set to null, and the function exits. Otherwise, the function proceeds with the consolidate phase, which merges trees of the same rank to maintain the Fibonacci Heap's structure.

time complexity:

worst case - In the worst case, the deleteMin function has a time complexity of O(logn). This is because the minimal node can have at most O(logn) children, as the number of children is bounded by the logarithm of the total number of nodes due to the properties of the Fibonacci heap. Detaching and integrating these children into the root list requires O(logn) time. Additionally, the consolidate phase processes at most O(logn) root trees, since the number of unique ranks in the heap is logarithmic in the total number of nodes

amortized complexity - The process involves iterating through the children, which are at most O(logn) , as shown in class, with each iteration performing constant-time operations, such as updating pointers. After integrating the children, the counters are updated to reflect the changes, and the method triggers the consolidation phase. This phase reorganizes the heap to ensure that no two trees share the same rank. Since the number of iterations is logarithmic relative to the heap size, the amortized time complexity of deleteMin is O(logn).

decreaseKey(HeapNode x, int diff)| amortized complexity O(1) | worst case complexity O(log n):

Description:

The decreaseKey method reduces the key of a given node x by a specified amount diff, assuming the precondition 0 < diff < x.key. If the node is null, the method exits without any action. After reducing the key, the method checks for heap order violations. If the node has a parent and its new key is smaller than the parent's key, the cut method detaches the node from its parent and adds it to the root list. If the parent was already marked, indicating it had previously lost a child, the method triggers a cascading cut using the cascadingCut function, which continues detaching ancestors up the tree as necessary. Finally, if the reduced key is smaller than the current minimum key in the heap, the min pointer is updated to reflect the new minimum node.

time complexity:

worst case- In the worst case, cascadingCut propagates up the tree, cutting all ancestors until it reaches the root. The height of the tree is at most O(logn), so this process takes O(logn). (We will see the detailed complexity analysis of cascadingCut later in the form).

amortized complexity - The amortized complexity of the decreaseKey method in is O(1), explained using the potential function, which measures the heap's state. The potential is defined as the number of trees in the root list plus twice the number of marked nodes. Each cascading cut either reduces the number of marked nodes or increases the number of trees in the root list, leading to a decrease in potential. Although cascading cuts can traverse up to O(logn) levels in the worst case, their cost is offset by the reduction in potential, ensuring the total "amortized cost" remains low. As a result, the average cost of performing decreaseKey over a sequence of operations is O(1), reflecting efficient restructuring based on the potential function.

delete(HeapNode x) | amortized complexity O(log n) | worst time complexity O(log n):

Description:

The delete method removes a specified node x from the Fibonacci Heap. If x is the minimal node, the method calls deleteMin() to handle its removal. Otherwise, it reduces the key of x to a value smaller than the current minimum using decreaseKey , ensuring x becomes the minimal node. If x has children, they are detached, added to the root list, and their parent pointers are set to null .

The node x is then removed from the circular doubly linked list of roots, and the heap counters are updated. Finally, the min pointer is restored to its original value, preserving the heap's structure.

time complexity:

worst case- The worst-case cost is driven by the decreaseKey operation, which may trigger cascading cuts up to O(logn) levels.

amortized complexity- The amortized complexity of the delete method is O(logn), driven by the decreaseKey and deleteMin operations. The potential function ensures that reducing the key of x to make it the minimum has an amortized cost of O(1), as cascading cuts reduce the potential by at least 1 per cut.

Once x becomes the minimal node, deleteMincon solidates the heap, ensuring no two trees share the same rank, which takes O(logn). Detaching and merging children, bounded by the logarithmic height of the tree, also takes O(logn). These steps ensure that, despite occasional expensive operations, the average cost of delete across multiple operations remains logarithmic.

totalLinks() | amortized complexity O(1) | worst case complexity O(1):

Description:

This method returns the total number of linking operations performed in the Fibonacci Heap since its creation

time complexity:

worst case - The method directly returns the value of totalLinksVar, which is a constant-time operation. So worst case complexity will be O(1)

amortized complexity- Since this is a simple lookup, its cost remains constant across all calls So amortized case complexity will be O(1)

totalCuts() | amortized complexity O(1) | worst case complexity O(1):

Description:

This method returns the total number of **cut operations** performed in the Fibonacci Heap since its creation.

time complexity:

worst case - The method directly retrieves the value of totalCutsVar, which is a constant-time operation So worst case complexity will be O(1).

amortized complexity- Since it is a simple lookup operation, the cost remains constant across all calls So amortized case complexity will be O(1).

meld (heap2) | amortized complexity O(1) | worst case complexity O(1):

Description:

The meld method efficiently combines the current Fibonacci Heap with another heap, heap2. If heap2 is null or empty, the method exits immediately. Otherwise, it updates the total number of links and cuts by adding the corresponding values from heap2. If the current heap is empty, it adopts all attributes of heap2, including its minimum node, size, and tree count. If the current heap is not empty, their circular root lists are merged usingmergeCircularLists, and the min pointer is updated to the smaller of the two minimums. Finally, the heap2 is out of use.

time complexity:

worst case - All operations, including updating pointers for the linked lists' start and end, are performed in O(1) .

amortized complexity – same as worst case complexity.

size() | amortized complexity O(1) | worst case complexity O(1):

Description:

This method retrieves the total number of nodes currently stored in the Fibonacci Heap.

time complexity:

worst case - The method simply returns the value of an integer variable (`sizeVar`), which is a constant-time operation. So worst case complexity will be O(1).

amortized complexity- Since this is a direct lookup operation, its cost remains constant across all calls So amortized case complexity will be O(1).

numTrees() | amortized complexity O(1) | worst case complexity O(1):

Description:

This method retrieves the number of trees currently in the Fibonacci Heap's root list.

time complexity:

worst case - The method simply returns the value of numTreesVar, a straight forward integer variable lookup, which is a constant-time operation So worst case complexity will be O(1).

amortized complexity- Since this is a direct lookup operation, its cost remains constant across all calls So amortized case complexity will be O(1).

consolidate () | amortized complexity O(log n) | worst case complexity O(n):

Description:

The consolidate method ensures that the Fibonacci Heap contains at most one tree of each rank, as proven in class, where the maximum rank is bounded by O(logn).

To achieve this, the method uses an array (rankTable) with a size proportional to the logarithm of the heap size, representing an upper bound for the number of final trees. It iterates through the root nodes of the heap, detaching each root from its siblings and checking for any previously processed root of the same rank in the array. If such a tree exists, it is removed from the array and linked with the current tree, where the tree with the smaller key becomes the parent, and the resulting tree's rank increases by 1. This process continues until the rank slot in the array is empty, at which point the resulting tree is placed in the array. Once all roots have been processed, the heap is reconstructed from the trees in the array by iterating through it and adding the non-null trees to the new root list.

The min pointer is updated to reflect the smallest key in the heap.

time complexity:

worst case - The root list contains all n nodes as separate trees, resulting in O(n) operations.

amortized complexity - The amortized complexity of the consolidate method is O(logn), explained through the potential method. The potential function, defined as the number of trees (T) plus twice the number of marked nodes (M), serves as a non-negative measure of the heap's structure. This function is valid because it starts at 0 for an empty heap and increases as the structure becomes more complex. During consolidation, the focus is on reducing the number of trees, making T the central parameter, while cuts are not relevant in this context. The actual cost of the operation is determined by the number of links performed, which is proportional to + log n, where T0 is the initial count of trees. The change in the potential is given by the difference in the number of trees before and after consolidation, (T1−T0). Combining these, the amortized cost is: Amortized Cost = Actual Cost +Change in Potential ≤ T0 + log n + (T1 − T0)≤ 2log n.

This demonstrates that consolidation effectively reduces the root list size, ensuring logarithmic complexity relative to the heap size. (even when structural adjustments are required)(?).

link (HeapNode y, HeapNode x) | amortized complexity O(1) | worst case complexity O(1):

Description:

The link method combines two trees of the same rank into one, where the tree with the smaller key (x) becomes the parent and the other tree (y) becomes its child. If x has no children, y is added as its sole child; otherwise, it is added to x's child list using the insertNodeToRootList helper. The method increments x's rank, decreases the total number of trees, unmarks y, and increments the total link counter.

time complexity:

worst case - The method involves constant-time operations: updating pointers, assigning parent-child relationships, and maintaining ranks and counts. So total complexity will be O(1)

amortized complexity – same as worst case complexity, So total complexity will be O(1)

cut (HeapNode y, HeapNode x) | amortized complexity O(1) | worst case complexity O(1):

Description:

The cut method removes a node x from its parent y and promotes x to the root list of the Fibonacci Heap, typically when x violates the heap property. It detaches x from its siblings using removeNodeFromCircularList and updates y.child to point to another child or null if x was the only child. The rank of y is decremented, and x's parent pointer is set to null, with its mark flag reset to false. The node x is then added to the root list via the insertNodeToRootList helper function.

Finally, totalCutsVar and numTreesVar are incremented to reflect the performed cut and the addition of a new tree to the root list.

time complexity:

worst case - Each step in the cut method involves constant-time operations, regardless of the size or depth of the Fibonacci Heap.

amortized complexity- same as worst case complexity.

cascadingCut (HeapNode y) | amortized complexity O(1) | worst case complexity O(log n):

Description:

The cascadingCut method maintains the Fibonacci Heap's structural properties when a node y is removed from its parent z due to a heap property violation. If y has no parent (z == null), the method exits as no further action is needed. If y is not marked, it is marked to indicate it has lost one child but its parent remains intact. If y is already marked, it is cut from its parent using the cut method and added to the root list, and the cascadingCut method is called recursively on its parent z. This process continues up the tree until an unmarked node or the root is reached, ensuring lazy structural adjustments while preserving efficiency

time complexity:

worst case - In the worst case, cascading cuts propagate up a path from a leaf node to the root. Since the height of the tree in a Fibonacci Heap is bounded by O(logn), the cost of cascading cuts is O(logn) in the worst case.

amortized complexity - The amortized complexity of the cascadingCut method is O(1), explained using the potential function that we saw akready, $\Phi = T + 2M$, where T is the number of trees in the root list and M is the number of marked nodes. When a cascading cut is performed, M decreases by 1 (as the node being cut is unmarked), and T increases by 1 (as a new root is added). This reduces the potential, effectively offsetting the cost of the operation. While the actual cost of propagating cascading cuts can reach O(logn) in the worst case due to recursive calls, the change in potential ensures that the amortized cost remains O(1).

mergeCircularLists(HeapNode a, HeapNode b)| amortized complexity O(1) | worst case complexity O(1):

Description:

The mergeCircularLists method combines two circular doubly linked lists represented by their starting nodes, a and b. If either list is null, the method exits without changes. It updates pointers by linking a and b together, ensuring that their respective next and prev nodes connect seamlessly, maintaining the circular structure.

time complexity:

worst case - The method involves a fixed number of pointer updates (8), regardless of the size of the lists we can also notice that there are no traversal or recursion occurs, so the time taken is constant. So the omplexity will be O(1)

amortized complexity- same as worst case complexity.

insertNodeToRootList(HeapNode node, HeapNode root)| amortized complexity O(1) | worst case complexity O(1):

Description:

The insertNodeToRootList method inserts a node into a circular doubly linked list, placing it immediately after the specified root node.

time complexity:

worst case - The method performs a fixed number of pointer updates (4), regardless of the size of the list, ensuring constant time complexity. So worst case complexity will be O(1).

amortized complexity- same as worst time complexity

removeNodeFromCircularList(HeapNode node)| amortized complexity O(1) | worst case complexity O(1):

Description:

The removeNodeFromCircularList method removes a node from a circular doubly linked list while maintaining the list's circular structure.

time complexity:

worst case - The method performs a fixed number of pointer updates (4), making it a constant-time operation regardless of the list size. So worst case complexity will be O(1).

amortized complexity- same as worst time complexity

Part 2– Experiments

Experiment 1:

1)

Serial Number (i)	***N***	***Run Time (milliseconds)***	***Heap's size at the end***	***Total number of links***	***Total number of cuts***	***Number of trees at the end***
1	6560	2	6559	6550	0	9
2	19682	3	19681	19674	0	7
3	59048	7	59047	59037	0	10
4	177146	24	177145	177133	0	12
5	531440	115	531439	531427	0	12

2) *time complexity analysis:*
We note that the experiment involves inserting n elements into the heap in random order, followed by deleting the minimum element from the heap. To achieve this, we perform the insert operation n times and the deleteMin operation once.
As we observed earlier, the time complexity of the insert function is constant. Therefore, performing n insert operations results in a total time complexity that is linear in the number of elements.
Similarly, we analyzed the deleteMin function and determined that its worst-case time complexity is logarithmic in the number of elements. Since we only need to perform the deleteMin operation once, the time complexity for this operation remains logarithmic.

The insert and deleteMin operations are independent and executed separately. Hence, for the overall time complexity analysis, we add the complexities of these operations. Thus, the total time complexity of the experiment, in the worst case, as a function of the input size, is:

$$n \cdot O(1) + O(\log n) = O(n) + O(\log n) = O(n)$$

3) *In this experiment, none of the measurements depend on the insertion order, so it will not affect them.*
Total number of links and total number of cuts wont be affected because we have n trees (each one contains just one node) and then we delete the minimum node (calling deleteMin()) and that is when we consolidate the trees- an algorithm that is not affected

by the order of the trees but just by the size of the heap. Therefore number of links is constant. We wont make any cuts in this experiment, regardless of the insertion order. The size of the heap surely wont change because in this experiment we are told just to delete one node so regardless of the insertion order the size will be n-1. Number of trees at the end- as we said, we call once the deleteMin() function and therefore we consolidate the trees in the heap. The number of trees at the end is determined by the consolidation and the consolidation is not affected by insertion order. Running time wont be affected because non of the measurements are.

Experiment 2:

Serial Number (i)	n	Runtime (milliseconds)	Heap's size at the end	Total number of links	Total number of cuts	Number of trees at the end
1	6560	9	3280	39149	35874	5
2	19682	13	9841	130199	120365	7
3	59048	48	29524	434931	405415	8
4	177146	140	88573	1448567	1360006	12
5	531440	984	265720	4733966	4468255	9

2)*time complexity analysis:*
For the next experiment, we need to perform n insertions in random order, followed by $\frac{n}{2}$ *deleteMin operations.*
We have already seen that the time complexity of the insert operation is O(1) in the worst case. Therefore, performing n insertions results in a total time complexity of O(n).
Next, we need to perform $\frac{n}{2}$ *deleteMin operations.*
As previously analyzed, the deleteMin operation has a worst-case time complexity of O(log n), where n is the number of elements in the heap. Thus, the time complexity for performing $\frac{n}{2}$ *deleteMin operations is derived by summing the logarithms of the remaining elements in the heap:*

$$\log n + \log(n-1) + \log(n-2) + \cdots + \log\left(\left(\frac{n}{2}\right)+1\right) + \log(\frac{n}{2})$$

By the properties of logarithms, this can be simplified as follows:

$$\log\left(n * (n-1) * (n-2) * \ldots * \left(\left(\frac{n}{2}\right)+1\right) * \left(\frac{n}{2}\right)\right)$$

The product above represents the factorial-like product of n down to $\frac{n}{2}$, which is bounded as follows:

Upper bound:

$$\boldsymbol{log}\left(\boldsymbol{n}*(\boldsymbol{n}-\boldsymbol{1})*(\boldsymbol{n}-\boldsymbol{2})*\ldots..*\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}+\boldsymbol{1}\right)*\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right)\right)\leq \boldsymbol{log}\left(\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right)^{\frac{\boldsymbol{n}}{\boldsymbol{2}}}\right)=\frac{\boldsymbol{n}}{\boldsymbol{2}}\boldsymbol{log}\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right)\rightarrow \boldsymbol{O}(\boldsymbol{n}\,\boldsymbol{log}\,\boldsymbol{n})$$

***lower bound**:*

$$\boldsymbol{log}\left(\boldsymbol{n}*(\boldsymbol{n}-\boldsymbol{1})*(\boldsymbol{n}-\boldsymbol{2})*\ldots..*\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}+\boldsymbol{1}\right)*\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right)\right)\geq \boldsymbol{log}\left((\boldsymbol{n})^{\frac{\boldsymbol{n}}{\boldsymbol{2}}}\right)=\frac{\boldsymbol{n}}{\boldsymbol{2}}\boldsymbol{log}(\boldsymbol{n})\rightarrow \boldsymbol{O}(\boldsymbol{n}\,\boldsymbol{log}\,\boldsymbol{n})$$

Thus, the total complexity of the $\frac{n}{2}$ deleteMin operations is O (n log n).
Finally, since the insert and deleteMin operations are performed independently, we sum their respective complexities to calculate the total time complexity for the experiment:

$$n*O(1)+O\left(\sum_{i=0}^{\frac{n}{2}}\log(n-i)\right)=O(n)+O(\log\left(\prod_{\{i=0\}}^{\frac{n}{2}}(n-i)\right))$$
$$=O(n)+O(n\log n)=O(n\log\ n)$$

***3)** In this experiment some of the measurements are affected by the insertion order and some are not. We will specify:*
*The size of the heap at the end will not be affected because regardless of the insertion order we will still delete the same number of nodes. The number of trees at the end will not be affected either. After n/2 deletions of the minimum node, we will have n/2 nodes in the heap regardless of the insertion order. The last operation in this experiment is deleting the minimum node (deleteMin()). The final number of trees after this operation is determined by the number of remaining nodes in the heap (n/2 as mentioned) and is calculated by the consolidate method that occurs in the last deletion. This method relies only on the number of nodes and therefore the final number of trees wont be **affected** by the insertion order.*
On the other hand, running time, the total number of links and the total number of cuts will be affected by the insertion order.

Insertion order affects the number of children of a node. The earlier a node was inserted the greater the number of his children after the first deleteMin() operation. When we delete the minimum node, we add his children to the roots list. The more children there are, the more links we make. That's why the total number of links is affected by the insertion order. Symmetrically, the total number of cuts will increase as the minimum node has more children (every time we add the children to the roots list we make number of children times cuts).

Experiment 3:

Serial Number (i)	n	***Run Time (milliseconds)***	***Heap's size at the end***	***Total number of links***	***Total number of cuts***	***Number of trees at the end***
1	6560	1	31	6550	6549	30
2	19682	5	31	19674	19672	29
3	59048	13	31	59037	59036	30
4	177146	42	31	177133	177131	29
5	531440	246	31	531427	531426	30

2) *time complexity analysis:*
In this experiment, we need to insert n elements into the heap in random order, delete the minimum element, and then, from the remaining n−1 elements, repeatedly delete the maximum until only 31 elements remain in the heap (which is $\mathbf{2^5 - 1}$).
For the first part, inserting n elements and deleting the minimum, we already analyzed the complexity and determined that it is linear in the number of elements, O(n)
Now, given that we have a pointer to the maximum element in the heap, and based on the analysis of the functions in the first part of the project, we know that the delete function has a logarithmic time complexity relative to the number of ***elements*** *in the heap. Therefore, analyzing the complexity of this part after the insertion and deletion of the minimum gives us the following cost:*

$$\boldsymbol{log}(\boldsymbol{n-1}) + \boldsymbol{log}(\boldsymbol{n-2}) + \cdots + \boldsymbol{log}(\mathbf{33}) + \boldsymbol{log}(\mathbf{32})$$

This can be expressed as:

$$\sum_{\{i=31\}}^{n-1} log(i)$$

We can now estimate this sum using integrals. Since all the elements are non-negative and the logarithmic function is monotonically increasing, we can use integral bounds for the approximation:

$$\sum_{\{k=31\}}^{n-1} log(k) \geq \int_{31}^{n-1} log(k)dk = (k\,log\,(k) - k)_{\{31\}}^{n-1} = O(n\,log\,n)$$

$$\sum_{\{k=31\}}^{n-1} log(k) \leq \int_{32}^{n} log(k)dk = (k\,log(k) - k)_{\{32\}}^{n} = O(n\,log\,n)$$

Combining these, the total complexity of the experiment is:

$$O(n) + O(\,n\,log\,n) = O(\,n\,log\,n)$$

3) *In this experiment some of the measurements are affected by the insertion order and some are not. We will specify:*
This experiment determines the heaps size at the end so it is not affected by the insertion order. Also the total number of links wont be affected because we link only in the deleteMin() method that happens here just once (first operation) and therefore as we mentioned in experiment 1, number of links is not affected by the insertion order.
As we said in experiment 2, the insertion order affects the number of children a node will have (after the deleteMin() operation, that here is the first operation we make).
The number of cuts is affected by the insertion order. Every time we delete the maximum node, its key is decreased and then he becomes a root by cutting him from his parent. Its children are then cut. The arrangement of nodes, which is influenced by the insertion order, determines the number of cuts, thus affecting the total number of cuts.
The number of trees at the end will be also affected by the insertion order because after every deletion of the maximum we wont consolidate as in the deleteMin(), and we wont make any operation that contains linking nodes. Therefore the number of trees at the end is affected by the number of cuts, which, as mentioned, is affected by the insertion order; hence, the number of trees at the end is affected by the insertion order. Running time will be affected too.

4)

for convenience let:
N – heap's size in the end
L- total number of links
T – number of trees in the end

C - total number of cuts

Firstly, note that for all three experiments and for every heap size in the table, the following equation holds:

$$L - C = N - T$$

This equation reflects the relationship between the number of links (L), the number of cuts C), the heap's size in the end (N), and the number of trees (T) in the heap at the end of the experiment.
It's important to observe that with every cut operation, a new tree is added to the root list, while every link operation reduces the number of trees by 1. From our analysis, at the end of the series of operations, the number of trees in the heap corresponds to the number of 1's in the binary representation of the number of elements remaining in the heap.
Experiment 1: *No cuts are performed, so the number of links corresponds to the difference between the heap size at the end of the experiment and the number of trees. Additionally, based on the implementation, this is also the minimum number of links required to create a heap with the smallest possible number of trees.*

Experiments 2 and 3: *The number of links increases in correlation with the number of cuts. This happens because every cut generates a new tree that will eventually need to be linked back into the heap during the algorithm.*

Thus, the equation described at the beginning of the paragraph accurately represents the relationship between these four variables. It holds true for all experiments and for every heap size given in the data.