The Hollow Heap Data Structure: An Introduction, Run-Time Analysis, and Comparison to the Fibonacci Heap

Alisha Sprinkle and Courtney Dixon

December 8, 2019

1 Abstract

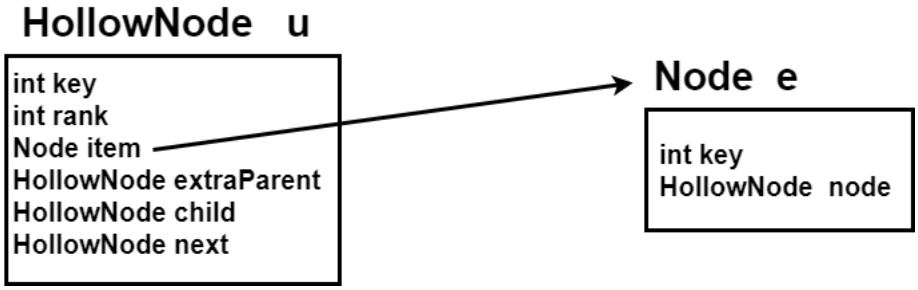
Hollow Heaps were introduced by Hansen, Kaplan, Tarjan, and Zwick in 2015. This data structure performs operations with the same amortized efficiency as the Fibonacci Heap introduced by Tarjan and Fredman. Hollow Heaps perform all operations characteristic of a heap, however, Hollow Heaps use lazy deletion and a directed acyclic graph rather than trees. The goal of this paper is to explain the Hollow Heap data structure precisely, analyze the amortized running time of the data structure, implement the data structure in Java, and compare the efficiency of Hollow Heaps to the renown Fibonacci Heap data structure.

2 Hollow Heaps

A Hollow Heap is a heap-ordered data structure. Therefore, a Hollow Heap maintains the heap property; the key of Node a is less than or equal to the key of Node b for every directed arc(a,b) in the structure where Node a is the parent of Node b. The typical properties of nodes still hold as well. A root is a node that does not have a parent and a leaf is a node that does not have any children. The root of a Hollow Heap has the minimum key in the Hollow Heap.

Hollow Heaps are just as efficient as Fibonacci Heaps. All the operations of a heap, excluding two, take $O(1)$ time in the worst case and amortized with Hollow Heaps. The two heap operations that take longer than constant time are *delete* and *deleteMin*, which each take $O(\log n)$ amortized time with Hollow Heaps. The *decreaseKey* operation in Hollow Heaps uses lazy deletion and reinsertion. Using lazy deletion prevents cascading cuts and this separates Hollow Heaps from Fibonacci Heaps. Other heaps use a tree or set of trees, but Hollow Heaps use a directed acyclic graph (dag) instead.

A Hollow Heap is made up of nodes that hold items and not the typical nodes that are items. HollowNodes have two integer fields, one Node field, and three HollowNode fields. The two integers are key and rank. The key is the value that the item in the HollowNode holds. The rank is the rank of the HollowNode in the HollowHeap. The Node field points to the item that the HollowNode is holding. The three HollowNode fields are child, next, and extra parent. The child field points to the last child of the HollowNode. As a HollowNode gains children they are put at the beginning of the list of children. The next field points to the sibling of the HollowNode. This allows us to traverse the list of children belonging to a HollowNode. The extra parent field points to a HollowNode that is the second parent of the HollowNode pointing to it. A second parent is gained when a HollowNode becomes hollow when the item it is holding is deleted or the key has been decreased causing the item to be stored in a new HollowNode entirely. The item that the HollowNode is holding is of type Node. The Node type has two fields. One integer field to hold the key and one HollowNode field that points to the HollowNode that is holding the item. The node structures are depicted in the image below.



HollowNode u holds an item, Node e.
Node e has a reference to HollowNode u

3 Hollow Heap Implementation

The authors presented three variants of the Hollow Heap data structure: Multi-Root, One-Root, and Two-Parent. The focus of this paper is the Two-Parent Hollow Heap; the final Hollow Heap data structure presented in the original paper. The Two-Parent Hollow Heap uses a directed acyclic graph (dag) that rids the algorithm of having to move children. The Two-Parent Hollow Heap performs the classic heap operations: *makeHeap*, *findMin*, *insert*, *deleteMin*, *meld*, *decreaseKey*, and *delete*.

The *makeHeap* method returns a new Hollow Heap that is empty. The *findMin* method returns the item pointed to by the minimum HollowNode in the Hollow Heap. The *insert* method returns a new Hollow Heap that contains all the old elements of the original Hollow Heap and the new element that was to be inserted. The *deleteMin* method removes the minimum element from the Hollow Heap. The *meld* method takes two Hollow Heaps and returns a Hollow Heap that is the combination of the two original Hollow Heaps. The *decreaseKey* method changes the key of the element to the provided key. A call to *decreaseKey* makes an entirely new HollowNode with a new Node both with the new key and the old HollowNode is made hollow. The *delete* method removes an element.

In addition to the classic heap methods, the Two-Parent Hollow Heap also has the following methods: *makeNode*, *link*, *addChild*, *doRankedLinks*, *doUnrankedLinks*. The *makeNode* method takes an item of type node and an integer key and makes a new HollowNode that holds the Node item with the key specified. When a new HollowNode is created it is considered full. A HollowNode becomes hollow after a call to *decreaseKey* or a call to *delete*. A HollowNode can never become full again and remains hollow until it is destroyed. The *link* method takes two HollowNodes and determines which one has the smaller key. The HollowNode with the smaller key becomes the parent of the HollowNode whose key is larger. The paper calls the parent HollowNode the "winner" and the child HollowNode the "loser". The *link* method returns the parent HollowNode, the "winner". The *addChild* method takes two HollowNodes and makes the first the sibling of the child of the second. The *doRankedLinks* method is used by the *delete* method and calls the *link* method based on the ranks of the HollowNodes in the HollowHeap. The *doUnrankedLinks* method is also used by the *delete* method and calls the *link* method regardless of the ranks of the HollowNodes in the HollowHeap.

4 Amortized Analysis via the Accounting Method

The *insert* operation is responsible for the creation of HollowNodes and therefore will be overcharged by $O(1)$ to account for the future *decreaseKey* operation that results in a rebuilding of the Hollow Heap. The rebuilding occurs when the number of *insert* operations plus the number of *decreaseKey* operations exceeds the number of full nodes multiplied by a constant greater than one. In formulaic form, rebuilding occurs when $N > cn : c > 1$, where N is the number of *insert* operations plus the number of *decreaseKey* operations, n is the number of full HollowNodes in the Hollow Heap, and c is a constant larger than one that makes the

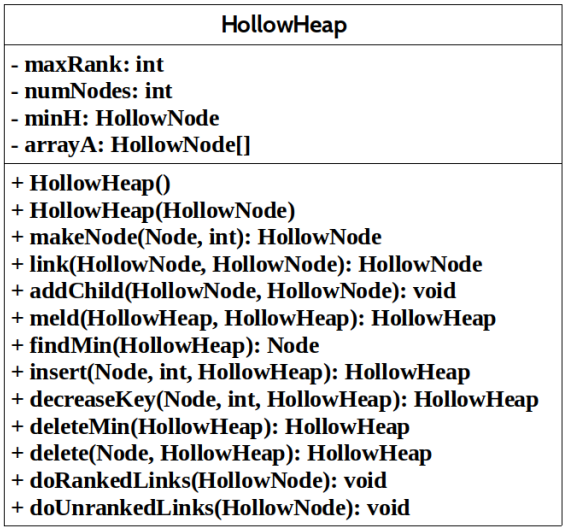
algorithm most efficient for some n sized Hollow Heap [1]. The same formula is true for a rebuild triggered by a call to the *delete* method.

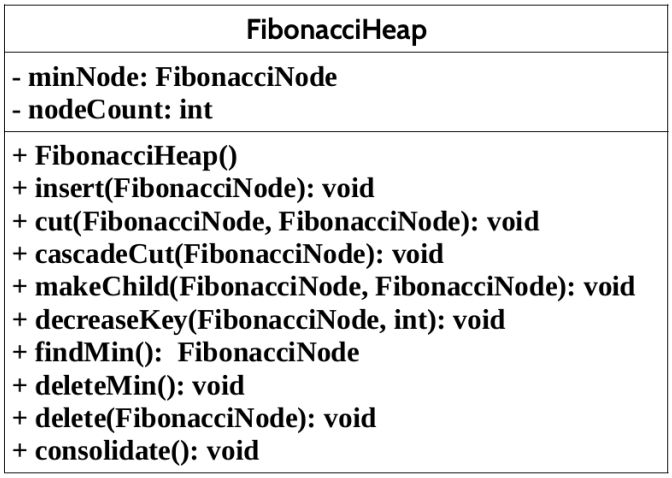
For the *decreaseKey* and *delete* results in the chart below, we will assume 17 *insert* operations have already occurred and our $c = 2$. The two charts are the same because *decreaseKey*, *delete*, and *deleteMin* all use some form of lazy deletion. The *decreaseKey* method makes a new HollowNode with the new key and makes the old HollowNode hollow instead of getting rid of it to prevent moving children. The *deleteMin* method calls the *delete* method, so the analysis is treated the same. The *delete* method just makes HollowNode hollow instead of getting rid of it to also prevent moving children. The *decreaseKey* and *delete* methods each take $O(\log n)$ amortized time because of the potential of a rebuilding that traverses the entire dag of height $O(\log n)$ getting rid of HollowNodes and linking full HollowNodes possibly using ranks depending on the event that triggered the rebuild.

Op #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
decreaseKey	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
rebuild																		17		
total	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	18	1	1
cumulative	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	35	36	37
delete	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
rebuild																		17		
total	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	18	1	1
cumulative	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	35	36	37

5 Hollow versus Fibonacci: A Comparison

Our implementation is of the Two-Parent Hollow Heap. We implemented the classic Fibonacci Heap so that we could compare the two data structures in terms of practical efficiency. Our programs are written in the Java programming language. Our source code can be found at *https://github.com/dixoncs/CS5110-Fall2019.git*. The following UML diagrams are for our HollowHeap and FibonacciHeap classes.

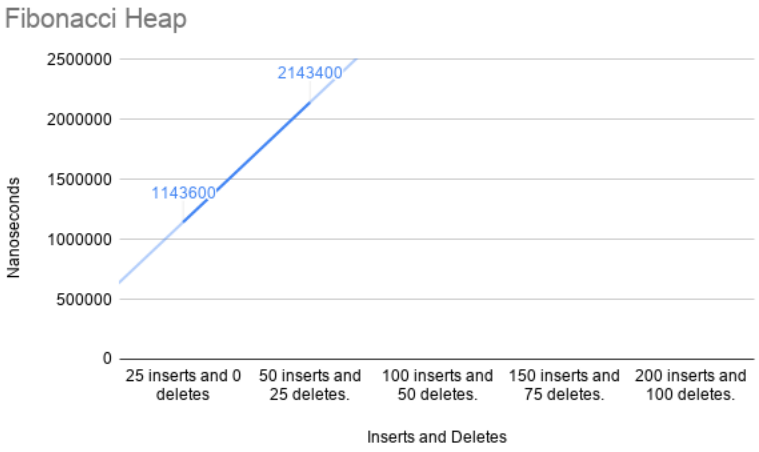




As you can see in the diagrams above, the methods for the two implementations are quite different. The Fibonacci Heap handles cascading cuts whereas the Hollow Heap avoids these cuts by using HollowNodes and lazy deletion. The Fibonacci Heap methods almost all have a void return type, but the Hollow Heap methods return either a Hollow Heap or a HollowNode. This difference lead to some creativity in method calls. There are differences in the node structures as well. The FibonacciNode has four pointers to keep track of the parent, child, left, and right FibonacciNodes as well as two integer fields and a boolean member. The first integer is for the key, the second is for the degree, and the boolean is to keep track of marked-ness. The HollowNode also has four pointers to node structures but only three are of type HollowNode. The item field is a pointer to a Node data type. There is still an integer type for the key and a rank fields which is consistent with the Fibonacci Heap fields. The Hollow Heap has a couple more methods than the Fibonacci Heap and this is because of the difference in the rebuilding process. Two-Parent Hollow Heaps are represented as a directed acyclic graph instead of a tree or set of trees like the Fibonacci Heap. Fibonacci Heaps and Hollow Heaps both have the same amortized runtime analysis. All of the classic heap methods run in $O(1)$ amortized time except *decreaseKey* and *delete*, which run in $O(\log n)$ amortized time for a n element heap of height $O(\log n)$.

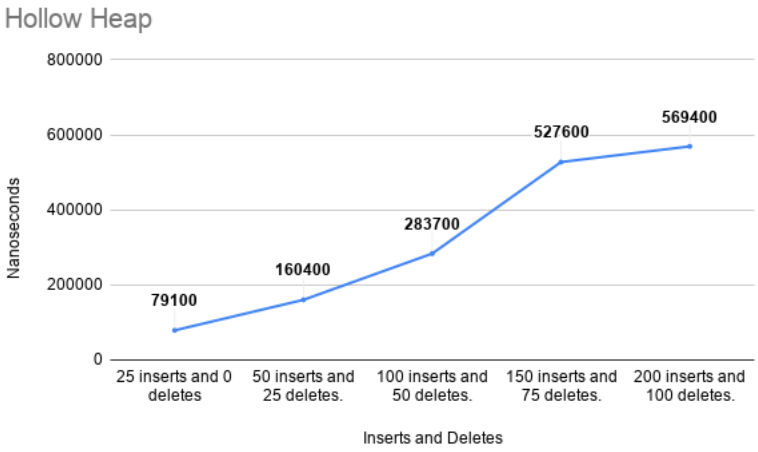
The implementation of Hollow Heap based on the findings in the paper proved to be challenging to near impossible. This is due in most part to a few inconsistencies within the language used within the document. A majority of the confusion is in the pseudocode itself on pages 16 and 17. There are two h variables being used throughout. The first is in relation to an actual hollow heap itself. The second is to the minimum node in the heap called in implementation, minH. It takes serious reading of the paper to determine which is being referenced. We were unsure most of the time which to use. This lead to trouble in *delete* and *decreaseKey*. Second was the format of the pseudocode which lead to a failure when implementing *delete* when the node to delete was the minimum. In the nested while loop, two closing brackets appear within an if statement that effectively closes off the logic. It is extremely unclear what is meant to occur. Our assumption of an if-else chain proved not to work.

The two graphs demonstrate our success however. We implemented a Fibonacci Heap to run in comparison. We performed a range of insertions and deletions, each increasing in difficulty. The Hollow Heap Implementation easily outperformed the Fibonacci Heap implementation after the third round of insertions. Java forced a timeout and refused to run the insertions for one hundred and above. Fibonacci Heaps are well implemented too so we are sure our implementation is sound. Future work would include contacting the authors if the paper to determine if they implemented the structure themselves based on their paper. It is our determination that relying solely on the paper will not grant the results needed to implement the data structure successfully and soundly. The images below are results of testing using our programs.



```
c:\Users\supah\CS5110-Fall2019>cd c:\Users\supah\CS5110-Fall2019 && C:\Users\supah\.vscode\extensions\vsc
java.vscode-java-debug-0.23.0\scripts\launcher.bat "C:\Program Files\Java\jdk1.8.0_201\bin\java" -Dfile.e
ncoding=UTF-8 -cp C:\Users\supah\AppData\Roaming\Code\User\workspaceStorage\829eac7eeb5674d2fa3d9024d403d
5cb\redhat.java\jdt_ws\jdt.ls-java-project\bin HeapDriver

FH took 1143600 nanoseconds for 25 inserts and 0 deletes.
FH took 2143400 nanoseconds for 50 inserts and 25 deletes.
```

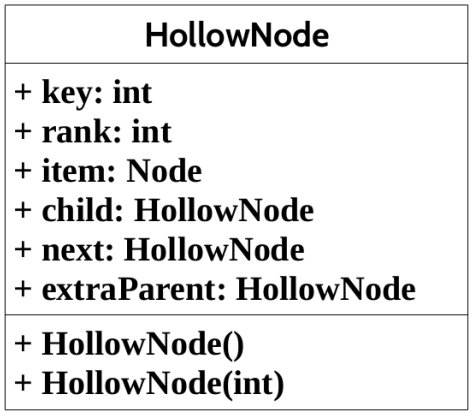



```
5cb\redhat.java\jdt_ws\jdt.ls-java-project\bin HeapDriver
HH took 79100 nanoseconds for 25 inserts and 0 deletes.
HH took 160400 nanoseconds for 50 inserts and 25 deletes.
HH took 283700 nanoseconds for 100 inserts and 50 deletes.
HH took 527600 nanoseconds for 150 inserts and 75 deletes.
HH took 569400 nanoseconds for 200 inserts and 100 deletes.
```

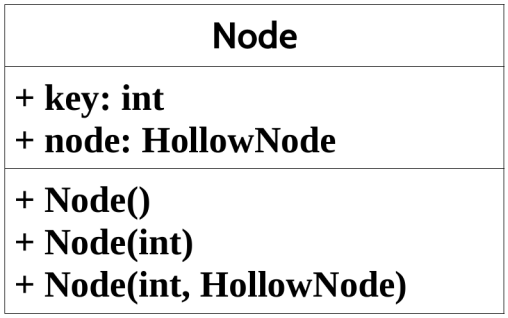
6 Conclusion

Hollow Heaps are directed acyclic graphs that use lazy deletion and reinsertion to avoid cascading cuts and having to move children. Hollow Heaps use a HollowNode data structure that holds an item of a Node data type. Hollow Heaps have all the classic heap methods and these methods run in the same amortized time as the same methods in Fibonacci Heaps. We did not find a significant difference between Fibonacci Heaps and Hollow Heaps through side by side programmatic testing. In the future we would like to complete the implementation of Hollow Heaps so that we can run more extensive tests with larger data sets to determine if Hollow Heaps are in fact an improvement of Fibonacci Heaps as claimed by the authors.

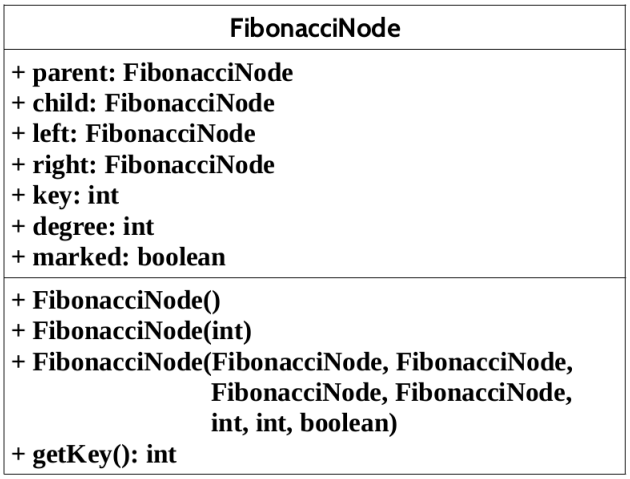
7 Figures



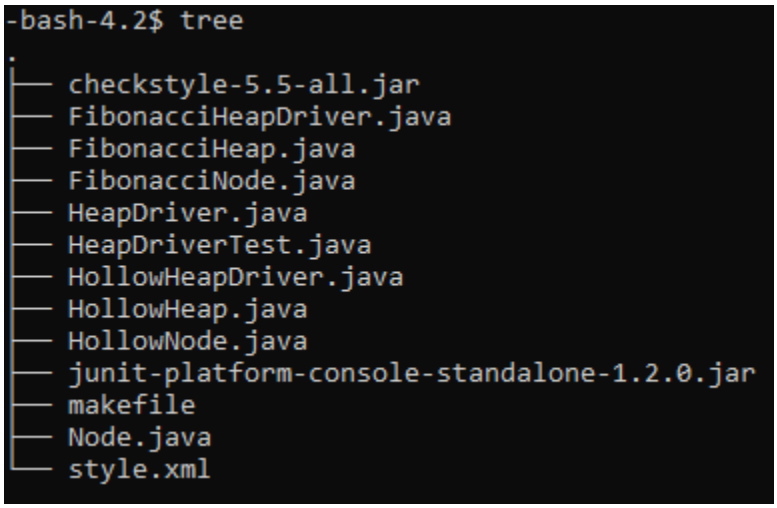
HollowNode Class UML Diagram



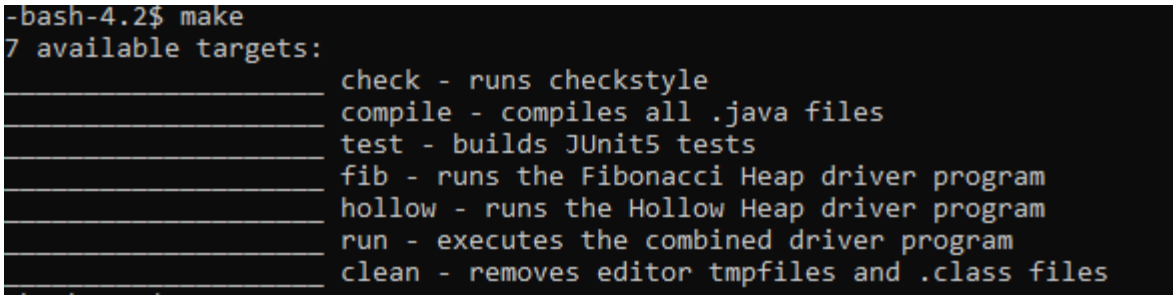
Node Class UML Diagram



FibonacciNode Class UML Diagram



Directory Tree



Makefile Targets

References

[1] Thomas Dueholm Hansen et al. "Hollow Heaps". In: *CoRR* abs/1510.06535 (2015). arXiv: 1510.06535. URL: http://arxiv.org/abs/1510.06535.