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

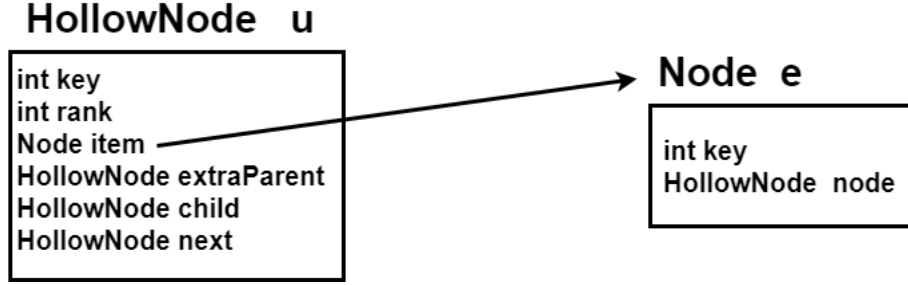Alisha Sprinkle and Courtney Dixon

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

```
int key
int rank
Node item
HollowNode extraParent
HollowNode child
HollowNode next
```

Node e

```
int key
HollowNode  node
```

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 comination 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

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

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].

For the *decreaseKey* and *delete* results in the chart above, 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 which just makes HollowNode hollow instead of getting rid of it to also prevent moving children.

WRITE MORE!!!!

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 UMLs diagram are for our HollowHeap and FibonacciHeap classes.

HollowHeap
- maxRank: int
- numNodes: int
- minH: HollowNode
- arrayA: HollowNode[]
+ HollowHeap()
+ HollowHeap(HollowNode)
+ makeNode(Node, int): HollowNode
+ link(HollowNode, HollowNode): HollowNode
+ addChild(HollowNode, HollowNode): void
+ meld(HollowHeap, HollowHeap): HollowHeap
+ findMin(HollowHeap): Node
+ insert(Node, int, HollowHeap): HollowHeap
+ decreaseKey(Node, int, HollowHeap): HollowHeap
+ deleteMin(HollowHeap): HollowHeap
+ delete(Node, HollowHeap): HollowHeap
+ doRankedLinks(HollowNode): void
+ doUnrankedLinks(HollowNode): void

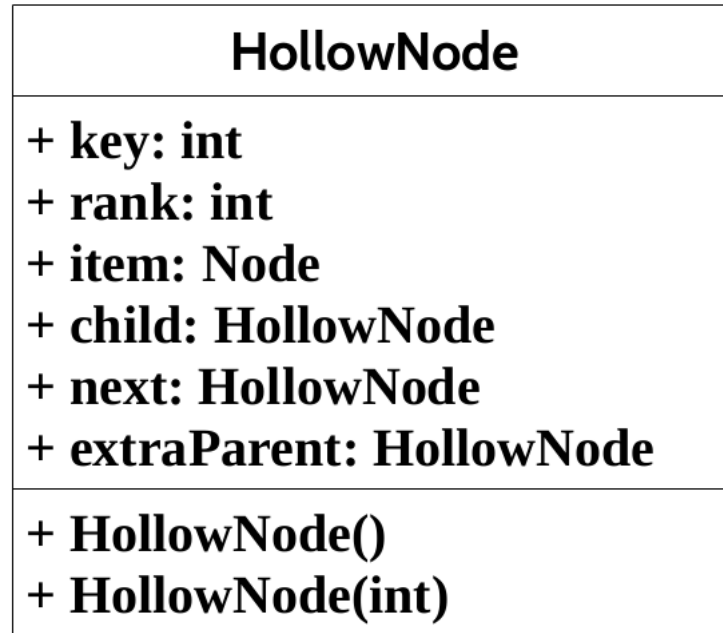| FibonacciHeap |
| --- |
| - minNode: FibonacciNode |
| - nodeCount: int |
| + FibonacciHeap() |
| + insert(FibonacciNode): void |
| + cut(FibonacciNode, FibonacciNode): void |
| + cascadeCut(FibonacciNode): void |
| + makeChild(FibonacciNode, FibonacciNode): void |
| + decreaseKey(FibonacciNode, int): void |
| + findMin(): FibonacciNode |
| + deleteMin(): void |
| + delete(FibonacciNode): void |
| + consolidate(): void |

5.1 Similarities

1. heap operations in $O(1)$ and $O(\log n)$

2. simple to implement

3.

5.2 Dissimilarities

1. Tree(s) versus Dag

2. no cascading cuts

3.

6   Figures

HollowNode

+ key: int
+ rank: int
+ item: Node
+ child: HollowNode
+ next: HollowNode
+ extraParent: HollowNode

+ HollowNode()
+ HollowNode(int)

HollowNode Class UML Diagram

FibonacciNode

+ parent: FibonacciNode
+ child: FibonacciNode
+ left: FibonacciNode
+ right: FibonacciNode
+ key: int
+ degree: int
+ marked: boolean

+ FibonacciNode()
+ FibonacciNode(int)
+ FibonacciNode(FibonacciNode, FibonacciNode,
                FibonacciNode, FibonacciNode,
                int, int, boolean)
+ getKey(): int
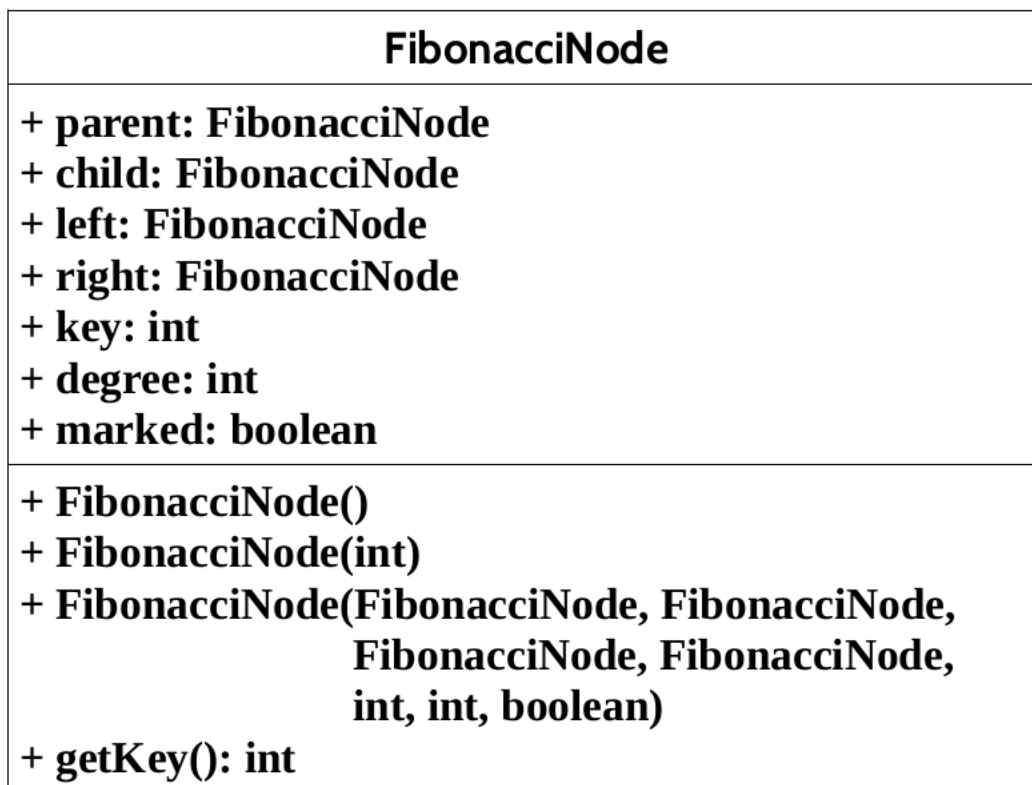
FibonacciNode Class UML Diagram

Node
+ key: int + node: HollowNode
+ Node() + Node(int) + Node(int, HollowNode)

Node Class UML Diagram

```
-bash-4.2$ tree
.
├── checkstyle-5.5-all.jar
├── FibonacciHeapDriver.java
├── FibonacciHeap.java
├── FibonacciNode.java
├── HeapDriver.java
├── HeapDriverTest.java
├── HollowHeapDriver.java
├── HollowHeap.java
├── HollowNode.java
├── junit-platform-console-standalone-1.2.0.jar
├── makefile
├── Node.java
└── style.xml
```

Directory Tree

```
-bash-4.2$ make
7 available targets:
_____ check - runs checkstyle
_____ compile - compiles all .java files
_____ test - builds JUnit5 tests
_____ fib - runs the Fibonacci Heap driver program
_____ hollow - runs the Hollow Heap driver program
_____ run - executes the combined driver program
_____ clean - removes editor tmpfiles and .class files
```

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.