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

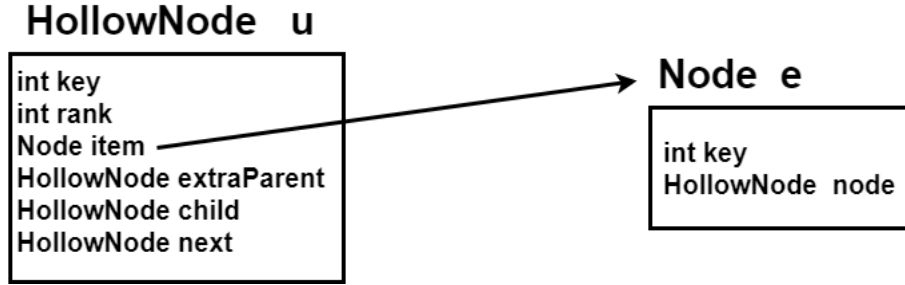Alisha Sprinkle and Courtney Dixon

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

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* methtod returns a new Hollow Heap that is empty. The *findMin* methtod returns the item pointed to by the minimum HollowNode in the Hollow Heap. The *insert* methtod returns a new Hollow Heap that contains all the old elements of the original Hollow Heap and the new HollowNode element that was to be inserted. The *deleteMin* methtod makes a call to the *findMin* methtod and removes the HollowNode returned. The *meld* methtod . The *decreaseKey* methtod . The *delete* methtod .
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. 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.

3   Amortized Analysis via the Accounting Method

"Amortized Analysis via Accounting Method"

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

4   Hollow Heap Implementation

Our implementation is of Two-Parent Hollow Heaps. Our programs are written in the Java programming language. The UML diagram below is for the Hollow Heap class. All other UML diagrams can be found in

the figures section of this paper.

5   Hollow versus Fibonacci: A Comparison

"Compare the two data structures in terms of practical efficiency"

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

```
-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
```

Figure : 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
```

Figure : 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.