

BIHEAPS, PIVOT SELECTION, AND DOUBLE ENDED PRIORITY QUEUES

MATTHEW GREGORY KRUPA

ABSTRACT. We define and classify a new category of graphs, one for each positive integer, called BiHeap graphs, which we use to define a new data structure called a BiHeap that is then applied to pivot selection for the QuickSelect and QuickSort algorithms. A part of a BiHeap's data lies in a min heap, a part lies in a max heap, and a part lies in both simultaneously. We give a simple to implement $O(N)$ algorithm, BiHeapify(), that forms a BiHeap out of any array of N values. We construct special graphs from BiHeap graphs and use them to implement a double ended queue that has amortized $O(N)$ insertions and amortized $O(N)$ pops. We define an $O(N)$ function that produces a pivot value, pivot_value, which is the middle element of the array of N values to which it was applied, such that for all $N \geq 16$, at least $2^{\lceil \log_3 N \rceil}$ elements have values \leq pivot_value and at least $2^{\lceil \log_3 N \rceil}$ elements have values \geq pivot_value, where $2^{\lceil \log_3 N \rceil} \geq N^{1/\log_2 3}$ with $1/\log_2 3 \approx 0.6309$. Testing on randomly generated arrays shows that on average, approximately $0.48N$ elements have values \leq pivot_value and approximately $0.48N$ elements have values \geq pivot_value. Thus, this pivot represents a substantial improvement above random selection of a pivot for use in the QuickSort and QuickSelect algorithms.


CONTENTS

Part 1. Introduction	3
1. Notation and Terminology	3
Part 2. BiHeaps Graphs	3
2. Definitions of the BiHeap Graph on N Nodes and a BiHeap on N Nodes	3
3. BiHeap Terminology	5
3.1. Pure, Impure, and Impure* Nodes	6
3.2. BiHeap Edges Terminology	6
3.3. Directed BiHeap Graphs	7
4. Example Construction of a Directed BiHeap Graph	7
5. Illustrations of Low Order BiHeaps	8
6. Basic Properties of BiHeaps	12
Part 3. Classification of BiHeaps	14
7. Even or Odd Number Of Nodes	14
7.1. BiHeaps with Interior-Incident Impure* Edges	15
8. Classification by Value of Number of Nodes mod 3	16
8.1. BiHeaps whose Heaps Share an Edge	16
8.2. BiHeaps with Two Distinct Leaf Impure* Edges Having Out-Degree 1	18
8.3. BiHeaps Whose Heaps Lack Interior Nodes With a Single Child	20
9. Corollaries of the Classification of BiHeap Graphs	21
9.1. Computing HeapSize(N)	21
9.2. Other Corollaries	21
9.3. In Nodes	22

Keywords and phrases. Double ended heaps, Heap, BiHeaps, QuickSelect, QuickSort, pivot, selection, double ended priority queue, queues, priority queue.

Part 4. BiHeapification	24
10. Overview of BiHeapification	24
11. Bounded Sifting Up Heaps	24
12. The BiHeapify Algorithm	25
Part 5. Sifting an Element	27
Part 6. Recursively BiHeapifying In Nodes with BiHeapifyInwards()	28
13. Minimum Number of Elements Below and Above an In Value	28
14. The BiHeapifyInwards Algorithm	29
15. The BiHeapifyInwardsNicerMath() Algorithm	30
15.1. The Number of Recursive Calls Made by BiHeapifyInwardsNicerMath()	30
15.2. Pivots	31
15.3. Minimum Number of Elements Below and Above BiHeapifyInwardsNicerMath()-Pivots	31
Part 7. Fused BiHeaps	32
16. FusedBiHeapify with a Range of Permitted Nodes	34
16.1. FusedBiHeapify With $N \bmod 3 \neq 2$	35
17. Sifting an Value in a Fused BiHeap	37
Part 8. Lambdas	38
18. A Short Review of Lambdas	38
19. Using Lambdas to Relabel Nodes	38
Part 9. A Double Ended Queue Based on Fused BiHeaps and Out Nodes	39
19.1. Inserting an Element	40
19.2. Popping the Minimum or the Maximum	42
Part 10. Appendix	43
20. Checking the BiHeap Condition	43
21. Emplacing All Minima and Maxima	44
22. $O(N)$ Functions with Multiple Recursive Calls	44
23. Descendants in Complete Binary Trees	48
23.1. Descendants of a Single Node	48
23.2. Ancestors of a Node	50
23.3. Descendants of a Subset Of an Interval of Nodes	52
23.4. Progenic Intervals	53
24. Flip-Ordered BiHeaps	59

Part 1. Introduction

It is assumed that the reader is familiar with graphs, trees, heaps, the standard $O(N)$ heapify algorithm and the proof that it is $O(N)$, the usual algorithms for sifting elements up and down heaps, and the QuickSelect algorithm. An introduction to these topics may be found in almost any introductory textbook on data structures and algorithms, such as [1]. Working implementations in Perl and C++ of the algorithms presented in this paper may be found at the author's GitHub account  github.com/mgkrupa.

1. Notation and Terminology. Given two integers b and $c > 0$, the non-negative remainder from the division of b by c will be denoted by $b \bmod c := b - c \lfloor \frac{b}{c} \rfloor$, where $\lfloor \cdot \rfloor$ (resp. $\lceil \cdot \rceil$) is the floor (resp. ceiling) function. Throughout this paper, we will use C++ like pseudocode where in particular, the result of a call, $a = b/c$, performing integer division with $c \neq 0$ and $bc \geq 0$, assigns the value $\lfloor \frac{b}{c} \rfloor$ to a . Let $\mathbb{N} = \{0, 1, \dots\}$ and for any N , let $\mathbb{N}_{<N} = \{i \in \mathbb{N} : i < N\}$ and define

$$\text{LeftMiddle}_N = \left\lfloor \frac{N-1}{2} \right\rfloor \quad \text{and} \quad \text{Middle}_N = \text{RightMiddle}_N = \left\lceil \frac{N}{2} \right\rceil$$

Suppose we have a list v_0, \dots, v_{N-1} of $N > 0$ elements. We will call $v_{\text{LeftMiddle}_N}$ (resp. $v_{\text{RightMiddle}_N}$) *the left middle* (resp. *the right middle*) element where if in addition $v_{\text{LeftMiddle}_N} = v_{\text{RightMiddle}_N}$ (which happens, for instance, if N is odd) then we will refer to this common value as *the middle* element. If these elements are associated with total ordered \leq and σ is a permutation of $\mathbb{N}_{<N}$ such that $v_{\sigma(j)} \leq v_{(k)}$ whenever $j \leq k$ with $j, k \in \mathbb{N}_{<N}$ then we will call the left (resp. right) middle element of $v_{\sigma(0)}, \dots, v_{\sigma(N-1)}$ *the left median* (resp. *the right median*) of v_0, \dots, v_{N-1} where if in addition the left median is equal to the right median then we will refer to this common value as *the median*. We will say that v_i *is emplaced* or that *the i^{th} element is emplaced* if $v_i = v_{\sigma(i)}$ or equivalently, if $|\{j \in \mathbb{N}_{<N} : v_j \leq v_i\}| \geq i$ and $|\{j \in \mathbb{N}_{<N} : v_j \geq v_i\}| \geq N - i$. If (1) $v_j \leq v_i$ for all $0 \leq j \leq i$ and (2) $v_j \geq v_i$ for all $i \leq j < N$ then we will say that the list v_0, \dots, v_{N-1} is *partitioned about v_i* or that it is *partitioned about index i* . We will say that v_0, \dots, v_{N-1} is *Dutch partitioned about v_i* or that it is *Dutch partitioned about index i* if it is partitioned about index i and there exist $s, e \in \mathbb{N}_{<N}$ such that for all $j \in \mathbb{N}_{<N}$, $v_j = v_i$ if and only if $s \leq j \leq e$.

We will henceforth assume that N is a positive integer and that we are given an ordered list of N nodes denoted by $V = V+0, \dots, V+(N-1)$. If these nodes are assigned values then we will denote the value of node w by $*w$. Define an involutory bijection $\text{Flip} : \mathbb{N}_{<N} \rightarrow \mathbb{N}_{<N}$ by

$$\text{Flip}_N(i) = N - 1 - i$$

where if N is understood then we will denote this by $\text{Flip}(i)$. By *the i^{th} node* we mean the node $V+i$ and if w is a node then by *the min heap coordinate of w* , *the heap coordinate of w* , or simply *the coordinate of w* we mean the unique integer i such that $w = V+i$ while by its *mirror coordinate* or its *max heap coordinate* we mean the integer $\text{Flip}(i)$. We will say that a node *has coordinates (i, m)* or *$i(m)$* , if (1) i and m are integers in $\mathbb{N}_{<N}$, (2) i is the coordinate of the node, and (3) m is the node's mirror coordinate, in which case we'll necessarily have that $i = \text{Flip}(m)$ and $m = \text{Flip}(i)$. Furthermore, we will identify the node $V+i$ with its coordinates. If w is a node with coordinate i then, assuming that the node is defined, for any integer k , $w+k$ will represent the node $V+(i+k)$. In the pseudocode, any variable that stores the heap coordinate (resp. mirror coordinate) of a node will have `_hc` (resp. `_mc`) as a postfix to its name.

When the values of nodes are stored in an array then the min heap coordinate of a node will correspond to the array index at which the node's value is stored. So if we say that one node v is before or to the left of (resp. is after or to the right of) another node w , then we mean that v 's min heap coordinate is less (resp. greater) than the min heap coordinate of w .

Part 2. BiHeaps Graphs

2. Definitions of the BiHeap Graph on N Nodes and a BiHeap on N Nodes. The concepts in the first two definitions are well known.

Definition and Convention 2.1. Define three endomorphisms on the non-negative integers by

$$\text{Parent}(i) = \begin{cases} \lfloor (i-1)/2 \rfloor & \text{if } i \neq 0 \\ 0 & \text{if } i = 0 \end{cases}, \quad \text{LeftChild}(i) = 2i + 1, \quad \text{and} \quad \text{RightChild}(i) = 2(i + 1)$$

We will identify nodes with their coordinates as well as with their min heap coordinates (but never their max heap coordinates) so that for all $0 < n \leq N$, the sets $\mathbb{N}_{<n} := \{0, \dots, n-1\}$ and $\mathbb{N}^{<n} := \text{Flip}(\mathbb{N}_{<n}) = \{(N-1)-(n-1), \dots, N-1\}$ denote the first (resp. last) n nodes of $V, \dots, V + (N-1)$. Denote the leaves of a tree T by $\Lambda(T)$. ■

Definition 2.2 (Min Heaps and Max Heaps). Let $h \in \{1, \dots, N\}$. By the *directed complete binary tree rooted at 0 on h nodes* we mean the graph $(\mathbb{N}_{<h}, E)$ where $(i, j) \in \mathbb{N}_{<h} \times \mathbb{N}_{<h}$ is an edge in E (going from i to j) if and only if either $j = \text{LeftChild}(i)$ or $j = \text{RightChild}(i)$, where in the former (resp. latter) case we'll say that j is i 's (*min heap*) *left* (resp. *right*) *child*. We may also refer to this graph as the *directed min heap graph of size h* . If these nodes are assigned partially ordered values then we'll say that this graph is a *min heap (rooted at 0 on h nodes)* if the value of a child is always greater than or equal to the value of its parent.

By the *directed complete binary tree rooted at $N-1$ on h nodes* we mean the graph $(\mathbb{N}^{<h}, E)$ where $(i, j) \in \mathbb{N}^{<h} \times \mathbb{N}^{<h}$ is an edge in E (going from i to j) if and only if either $\text{Flip}(j) = \text{LeftChild}(\text{Flip}(i))$ or $\text{Flip}(j) = \text{RightChild}(\text{Flip}(i))$, where in the former (resp. latter) case we'll say that j is i 's (*max heap*) *left* (resp. *right*) *child*. Note that if $m = \text{Flip}(i)$ and $n = \text{Flip}(j)$ are the max heap coordinates of these nodes then (i, j) is in E if and only if either $n = \text{LeftChild}(m)$ or $n = \text{RightChild}(m)$; this characterization is the reason for defining max heap coordinates. We may also refer to this graph as the *directed max heap graph of size h* . If these nodes are assigned partially ordered values then we'll say that this graph is a *max heap (rooted at $N-1$ on h nodes)* if the value of a child is always less than or equal to the value of its parent.

By the (*undirected*) *complete binary tree rooted at 0* (resp. *at $N-1$*) *on h nodes* we mean the undirected graph induced by the directed complete binary tree rooted at 0 (resp. $N-1$) on h nodes. ■

Notation 2.3. The BiHeap graph on N nodes will be denoted by $B_N = (V(B_N), E(B_N))$, where the vertices are $V(B_N) := \mathbb{N}^{<N}$. Its edges $E(B_N)$ are defined in def. 2.4.

Definition 2.4 (BiHeaps). For any positive integer $n \leq N$, let C_n (resp. C^n) denote the complete binary tree on nodes $\mathbb{N}_{<n}$ (resp. $\mathbb{N}^{<n}$) rooted at node 0 (resp. $N-1$) (def. 2.2). Let U_n denote the graph on $\{0, \dots, N-1\}$ formed by the union of the graphs C_n and C^n (where we only allow at most one edge between any two given nodes). Let R_n (resp. R^n) denote the restriction of U_n to the nodes $\mathbb{N}_{<n}$ (resp. $\mathbb{N}^{<n}$). Observe that due to the symmetry of their construction, R_n and R^n are necessarily isomorphic graphs (via $\text{Flip}()$) and R_n is a binary tree rooted at 0 if and only if R^n is a binary tree rooted at $N-1$, in which case $R_n = C_n$, $R^n = C^n$, and they are isomorphic (via $\text{Flip}()$) as rooted binary trees.

Let $\text{HeapSize}(N)$ denote the unique largest integer $h \leq N$ such that R_h is a tree rooted at 0. Note that by symmetry, $\text{HeapSize}(N)$ is also the unique largest integer $h \leq N$ such that R^h is a tree rooted at $N-1$. We will call $U_{\text{HeapSize}(N)}$ the *BiHeap graph on N nodes* and denote it by B_N .

By the *heap size of B_N* we mean the integer $\text{HeapSize}(N)$. Letting $h = \text{HeapSize}(N)$, by the *heap* or the *min heap* of B_N , denoted by MinH_N , we mean the graph $R_h = C_h$ and by the *mirror heap* or the *max heap* of B_N , denoted by MaxH_N , we mean the graph $R^h = C^h$.

Define $\text{PureHeapSize}(N) = \lfloor N/2 \rfloor$ and $\text{ExtPureHeapSize}(N) = \lceil N/2 \rceil$. Letting $\rho = \text{PureHeapSize}(N)$, we will call $R_\rho = C_\rho$ (resp. $R^\rho = C^\rho$) the *pure heap*, the *pure min heap*, or the PMinH_N (resp. the *pure mirror heap*, the *pure max heap*, or the PMaxH_N) of B_N . Letting $\epsilon = \text{ExtPureHeapSize}(N)$, we will call $R_\epsilon = C_\epsilon$ (resp. $R^\epsilon = C^\epsilon$) the *pure* heap*, the *pure* min heap*, or the *extended pure min heap* (resp. the *pure* mirror heap*, the *pure* max heap*, or the *extended pure max heap*) of B_N and denote it by ExtPMinH_N (resp. ExtPMaxH_N). The subscript N may be omitted if it is understood. ■

Definition 2.5 (Directed BiHeaps). If the nodes $0, \dots, N-1$ are assigned values and if $<$ is a partial order on these values then we will say that B_N is a *BiHeap with respect to $<$* or a *directed BiHeap (with min at 0 and max at $N-1$)* if both of the following conditions hold:

- (1) $R_h = C_h$ is a min heap with respect to $<$ rooted at node 0 (def. 2.2), and
- (2) $R^h = C^h$ is a max heap with respect to $<$ rooted at node $N-1$ (def. 2.2)

where $h = \text{HeapSize}(N)$. If the partial order $<$ is clear from context then we will simply say that B_N is a *BiHeap*. We will henceforth assume without mention that, whenever it is needed, we are given some partial order on the values of the nodes of B_N . Since all of the algorithms that we present require that $<$ be a total order, we will henceforth assume this as well. ■

Included in section 20 of the appendix is C++ like pseudo-code for determining whether or not a sequence of values forms a directed BiHeap.

3. BiHeap Terminology. Note that nodes and edges either belong only to the min heap, only to the max heap, or belong to both simultaneously. In particular, binary tree definitions such as interior node and leaf are now ambiguous. Since the definition of a BiHeap is merely that heap conditions hold on two subgraphs of the BiHeap graph, we adopt to BiHeaps all of the usual terminology used with heaps whenever its meaning is either obvious or else clearly refers to one or both of these two heaps.

Definition 3.1. An edge or a node in a BiHeap graph is said to be *min* (resp. *max*) if it belongs to the min (resp. max) heap while it is *pure min* (resp. *pure max*, *pure* min*, *pure* max*) if it belongs to the pure min (resp. pure max, pure* min, pure* max) heap.

Definition 3.2 (Children and Parents in a BiHeap). Let c and p be two nodes with coordinates $i(m)$ and $j(n)$. We will say that c is p 's *min (heap) left child* (resp. *right child*) or its *heap left* (resp. *right*) *child*, written $c = \text{LeftChild}_{\text{Min}}(p)$ (resp. $c = \text{RightChild}_{\text{Min}}(p)$), if both nodes belong to the min heap and when considered as nodes in the min heap, c is p 's left (resp. right) child. In either case we will call p the *min (heap) parent* of c , written $p = \text{Parent}_{\text{Min}}(c)$, and say that c is a *min (heap) child* of p , which we'll denote by $c = \text{Child}_{\text{Min}}(p)$. In any of these definitions we may replace c and p with i and j , respectively.

We will also say that c is p 's *max (heap) left child* (resp. *right child*) or its *mirror (heap) left child* (resp. *right child*), written $c = \text{LeftChild}_{\text{Max}}(p)$ (resp. $c = \text{RightChild}_{\text{Max}}(p)$), if both nodes belong to the max heap and when considered as nodes in the max heap, c is p 's left (resp. right) child. In either case we will call p the *max (heap) parent* of c , written $p = \text{Parent}_{\text{Max}}(c)$, and say that c is a *max (heap) child* of p , which we'll denote by $c = \text{Child}_{\text{Max}}(p)$. In any of these definitions, if $m = \text{Flip}(i)$ and $n = \text{Flip}(j)$ are understood to be max heap coordinates then we may replace c and p with m and n , respectively.

Remark 3.3. Recalling that $\text{LeftChild}(i) = 2i + 1$ and $\text{RightChild}(i) = 2(i + 1)$, note that $c = \text{LeftChild}_{\text{Min}}(p)$ (resp. $c = \text{RightChild}_{\text{Min}}(p)$) if and only if c and p belong to the min heap and $i = \text{LeftChild}(j)$ (resp. $i = \text{RightChild}(j)$). Also, $c = \text{LeftChild}_{\text{Max}}(p)$ (resp. $c = \text{RightChild}_{\text{Max}}(p)$) if and only if c and p belong to the max heap and $m = \text{LeftChild}(n)$ (resp. $m = \text{RightChild}(n)$) or equivalently, $2j - i = N$ (resp. $2j - i = N + 1$).

Definition 3.4. By the *first leaf of the min* (resp. *max*) *heap*, denoted by $\text{FirstLeaf}_{\text{Min}}(B_N)$ (resp. $\text{FirstLeaf}_{\text{Max}}(B_N)$), we mean the node with min (resp. max) heap coordinate

$$\text{FirstLeaf}(N) := \text{FirstLeaf}_{\text{Min}}(N) := \left\lfloor \frac{\text{HeapSize}(N)}{2} \right\rfloor.$$

Note that the first leaf of the max heap has min heap coordinate $\text{FirstLeaf}_{\text{Max}}(N) := \text{Flip}(\text{FirstLeaf}_{\text{Min}}(N))$. By the *last min* (resp. *max*) *heap leaf* we mean the node with min (resp. max) heap coordinate $\text{HeapSize}(N) - 1$ (resp. $\text{Flip}(\text{HeapSize}(N) - 1)$).

For $N > 1$, by *the last interior node of the min (resp. max) heap*, denoted by $\text{LastInt}_{\text{Min}}(B_N)$ (resp. $\text{LastInt}_{\text{Max}}(B_N)$), we mean the node with min heap coordinate

$$\text{LastInt}(N) := \text{LastInt}_{\text{Min}}(N) := \text{FirstLeaf}_{\text{Min}}(N) - 1$$

(resp. $\text{LastInt}_{\text{Max}}(N) := \text{Flip}(\text{LastInt}_{\text{Min}}(N))$).

The analogous definitions and notation for the pure min heap and the pure max heap are clear. \blacksquare

3.1. Pure, Impure, and Impure* Nodes. Note that nodes and edges either belong only to the min heap, only to the max heap, or belong to both simultaneously. In particular, binary tree definitions such as interior node and leaf are now ambiguous.

Definition 3.5. Say that a node v is:

- (1) a *min* (resp. *max*, *pure min*, *pure max*, *pure* min*, *pure* max*) *leaf* if it is a leaf in the min (resp. max, pure min, pure max, pure* min, pure* max) heap.
- (2) *impure min* (resp. *impure max*) or a *non-pure min* (resp. *non-pure max*) if it belongs to the min (resp. max) heap but not to the pure min (resp. pure max) heap.
- (3) *impure* min* (resp. *impure max*) or a *non-pure* min* (resp. *non-extended pure max*) if it belongs to the min (resp. max) heap but not to the pure* min (resp. pure* max) heap.
- (4) *impure* if it is an impure min node or an impure max node.
- (5) *impure** if it is an impure* min node or an impure* max node.
- (6) an *impure min* (resp. *impure max*, *impure* min*, *impure* max*) *leaf-adjacent* if
 - (a) it is an impure min (resp. impure max, impure* min, impure* max) node, and
 - (b) within the min (resp. max, min, max) heap, there exists an edge to v from a leaf of the pure min (resp. pure max, pure* min, pure* max) heap.
- (7) an *impure min* (resp. *impure max*, *impure* min*, *impure* max*) *interior-adjacent* node if
 - (a) it is an impure min (resp. impure max, impure* min, impure* max) node and
 - (b) within the min (resp. max, min, max) heap, there exists an edge to v from an node that is in the interior of the pure min (resp. pure max, pure* min, pure* max) heap.

\blacksquare

3.1.1. In Nodes and Edge Terminology.

Definition and Notation 3.6. Let $\text{In}(B_N)$ denote the set of nodes belonging to the *intersection* of the set of nodes in the min heap with the set of nodes in the max heap and call any such node an *In node*. Let $\text{In}(N)$ denote $|\text{In}(B_N)|$, the number of nodes in $\text{In}(B_N)$, and let $\text{In}_{\text{PMin}}(B_N)$ (resp. $\text{In}_{\text{PMax}}(B_N)$) denote the nodes that belong to both the pure min (resp. pure max) heap and $\text{In}(B_N)$. Let $\text{In}_{\text{PMin}}(N) = |\text{In}_{\text{PMin}}(B_N)|$ and let $\text{In}_{\text{PMax}}(N) = |\text{In}_{\text{PMax}}(B_N)|$. By the *first* (resp. *last*) *In node* we mean the *In node* with the smallest (resp. largest) min heap coordinate. We will denote the first *In node* by $\text{FirstIn}(N)$.

Any node in B_N that is not an *In node* will be called an *Out node*. The set of *Out nodes* will be denoted by $\text{Out}(B_N)$ and the number of *Out nodes* will be denoted by $\text{Out}(N)$. Let $\text{Out}_{\text{P*Min}}(B_N)$ ($\text{Out}_{\text{P*Max}}(B_N)$) denote the nodes that belong to both the pure* min (resp. pure* max) heap and $\text{Out}(B_N)$ and let $\text{Out}_{\text{P*Min}}(N) = |\text{Out}_{\text{P*Min}}(B_N)|$ (resp. $\text{Out}_{\text{P*Max}}(N) = |\text{Out}_{\text{P*Max}}(B_N)|$) \blacksquare

3.2. BiHeap Edges Terminology. For edges, in addition to definition 3.1, we introduce the following terminology, where in reference to an edge, “impure” (resp. “pure”) refers to the dashed (resp. solid) arrows in figures 2, 3, 4, and 5.

Definition 3.7. Say that an edge e that is incident to nodes v and w is

- (1) a *min* (resp. *max*, *pure min*, *pure max*, *pure* min*, *pure* max*) *edge* if it is a edge in the min (resp. max, pure min, pure max, pure* min, pure* max) heap.

- (2) *impure min* (resp. *impure max*, *impure** *min*, *impure** *max*) if it is an edge that belongs to the min (resp. max, min, max) heap but not to the pure min (resp. pure max, pure* min, pure* max) heap.
 - Note that being an impure min edge is equivalent to belonging to the min heap with one incident node belonging to the pure min heap with the other belonging to the pure* max heap.
 - (3) *impure* if it is a impure min edge or a impure max edge.
 - (4) *impure** if it is a *impure** min edge or a *impure** max edge.
 - Note that if N is odd then no *impure** edge is incident to the middle node.
 - (5) *extended impure min* (resp. *extended impure max*) if it is an edge that belongs to the min (resp. max) heap but not to the pure min (resp. pure max) heap.
 - This is equivalent to requiring that it belong to the min (resp. max) heap with one of its incident nodes belonging to the extended pure min heap while the other belongs to the extended pure max heap.
 - (6) *extended impure* if it is an extended impure min edge or an extended impure max edge.
 - Note that if N is odd then an extended impure may be incident to the middle node.
 - (7) *leaf-incident impure min* (resp. *impure max*, *impure** *min*, *impure** *max*) (a) if it is an impure min (resp. impure max, *impure** min, *impure** max) edge and (b) it is incident to an impure min (resp. impure max, *impure** min, *impure** max) leaf-adjacent node (def. 3.5(6)).
 - That is, an edge is leaf *impure** min (resp. leaf *impure** max) if one of its incident nodes is a leaf in the pure* min (resp. pure* max) heap while the other incident node does not belong to the pure* min (resp. pure* max) heap.
- It is *leaf impure* if it is a leaf impure min edge or a leaf impure max edge. It is *leaf impure** if it is a leaf *impure** min edge or a leaf *impure** max edge.
- (8) *interior-incident impure min* (resp. *impure max*, *impure** *min*, *impure** *max*) if
 - (a) it is an impure min (resp. impure max, *impure** min, *impure** max) edge and
 - (b) it is incident to an impure min (resp. impure max, *impure** min, *impure** max) interior-adjacent node (def. 3.5(7)).
- It is *interior-incident impure* if it is an interior-incident impure min edge or an interior-incident impure max edge. It is *interior-incident impure** if it is an interior-incident *impure** min edge or an interior-incident *impure** max edge.
- (9) a *double (headed) arrow* if the two nodes adjacent to the edge both (a) belong to the min heap as well as to the max heap, and (b) are incident to each other in both the min heap and the max heap.

■

3.3. Directed BiHeap Graphs. As we shall see, there is at most one edge that belongs simultaneously to the min heap and to the max heap. Consequently, we can unambiguously define a direction for all but possibly one edge in B_N .

Definition 3.8. Define the *directed BiHeap graph* on N nodes, denoted by \vec{B}_N , to be graph B_N with each edge directed so as to go from its parent to its child, where if an edge (v_1, v_2) simultaneously belongs to both the min heap and the max then this edge is to be interpreted as a bi-directional edge (or if more appropriate for the situation, it may instead be replaced by two directed edges $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_1$). Explicitly, an edge in the min heap (resp. max heap) goes from its parent to its child. ■

There is thus a canonical one-to-one correspondence between BiHeap graphs and directed BiHeap graphs, which we will henceforth use without mention to identify the two.

Remark 3.9 (Directed BiHeaps should not be confused with directed BiHeap graphs). While the definition of a *directed BiHeap* (def. 2.5) requires that every node in B_N be assigned some value from some partially ordered set, no such partial order or valuation is needed in the definition of a *directed BiHeap graph* (def. 3.8).

4. Example Construction of a Directed BiHeap Graph. To become comfortable with BiHeaps, the author recommends that the reader construct a several BiHeap graphs for any 6 consecutive values of N by

hand by going through the construction in definition 2.4 and (symmetrically) adding two edges at a time. We now illustrate this by constructing the directed BiHeap graph on $N = 14$ nodes

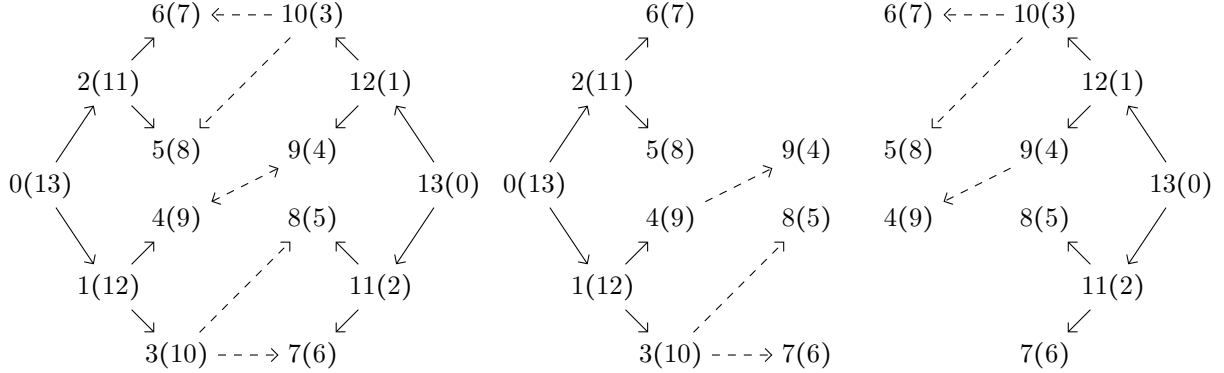


FIGURE 1. On the left is the directed (def. 3.8) BiHeap graph on 14 nodes, B_{14} , which has $\text{HeapSize}(14) = 10$. For the BiHeap graph, the double headed arrow should be interpreted as being a single edge in the graph. In the center is this BiHeap's directed (def. 2.2) min heap graph, whose restriction to the solid arrows forms the (directed) pure* min heap graph. At the right is this BiHeap's directed max heap graph, whose restriction to the solid arrows forms the (directed) pure* max heap graph.

Note that B_{14} is the union of the min heap graph with the max heap graph, where of course when a min heap node and a max heap node have the same coordinates then they are considered to be the same node. Visually, one obtains the BiHeap graph by “rigidly sliding without rotation” the min heap graph (pictured in the center) to the right until all nodes with equal coordinates overlap one another.

To construct B_{14} , we will now go through the construction in definition 2.4 and (symmetrically) adding two edges at a time. For instance first add edges $0(13) \rightarrow 1(12)$ and $13(0) \rightarrow 12(1)$ (i.e. from a min/max root to its left child) and then check the defining condition (which holds). Follow this by adding edges $0(13) \rightarrow 2(11)$ and $13(0) \rightarrow 11(2)$ (i.e. from a min/max root to its right child) and then checking the defining condition (which holds). Now add edges $1(12) \rightarrow 3(10)$ and $12(1) \rightarrow 10(3)$ and check the defining condition (which again holds). This process continues until eventually (after 7 more iterations in this case) the defining condition fails to hold, which implies that the last two edges should not have been added (i.e. only 6 more iterations should have been done rather than 7).

5. Illustrations of Low Order BiHeaps. We now illustrate the directed BiHeap graphs on N nodes for $1 < N < 28$ where $N = 1$ is omitted since it is just a point. The binary tree formed by restricting the BiHeap graph to all arrows (together with their incident nodes) going from left to right (resp. right to left) form the min (resp. max) heap of the BiHeap graph. The two connected components of the subgraph consisting of only the solid arrows and their nodes form the two pure* subheaps of these BiHeap graphs. Note that some of the BiHeap graphs in the following illustrations have an arrow that is double headed, which should be interpreted in the obvious way as a bi-directional arrow. That is, an arrow is drawn as double headed if and only if there is an edge between those two nodes that belongs to both the min heap and the max heap.

If one were to draw a line straight down the middle of one of these illustrated BiHeap graphs then the subgraph on the left (resp. right), which excludes the middle node if N is odd, would be the pure* min heap (resp. pure* max heap). The pure* min heap and the pure* max heap are always isomorphic to each other via the isomorphism $(i, m) \mapsto (m, i)$. Note that if one were to rotate the min heap (resp. pure* min heap) by 180 degrees around the center of the illustration then, ignoring all nodes' labels, one obtains the max heap (resp. pure* max heap).

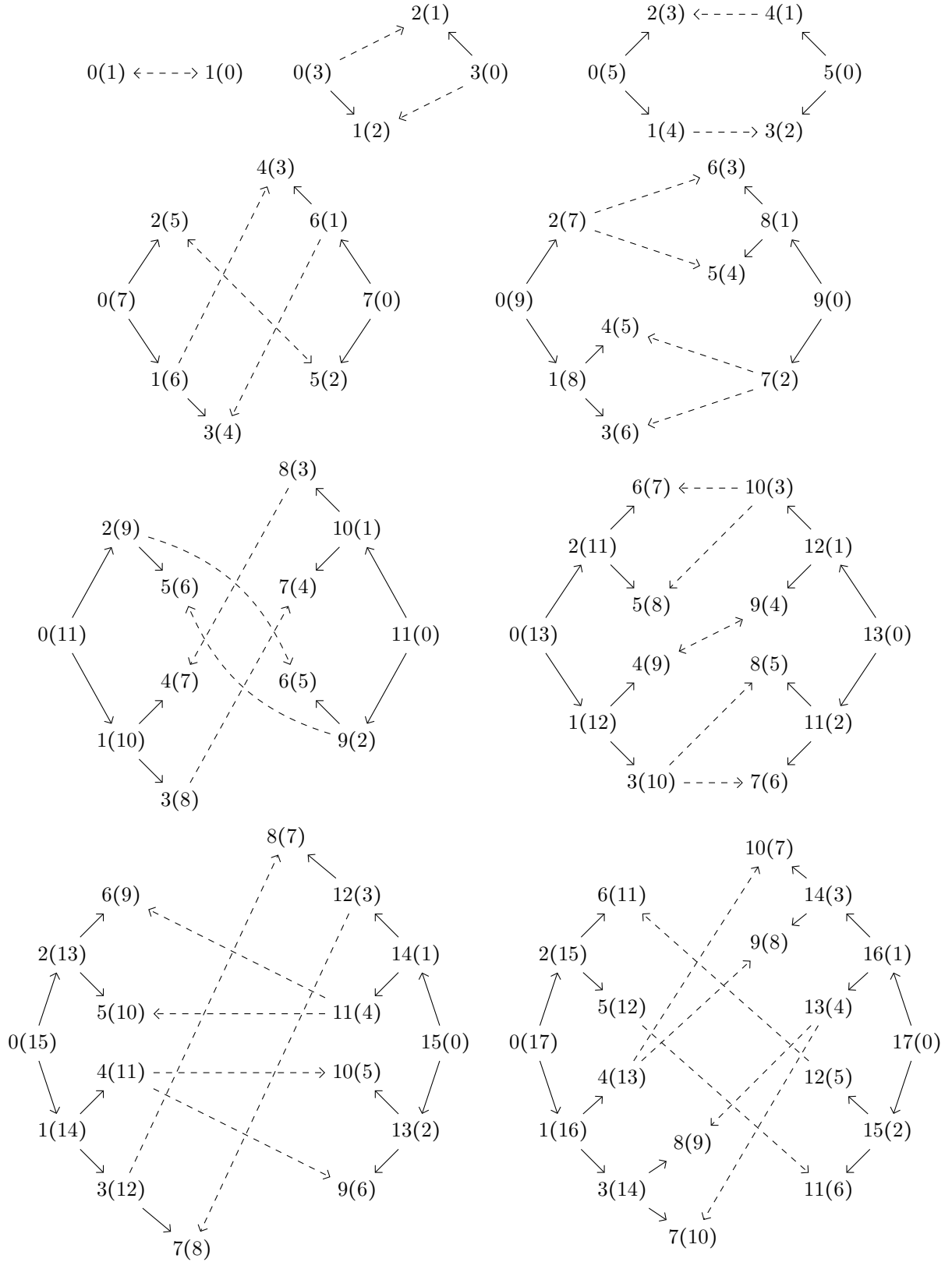


FIGURE 2. From left to right and from the top down, these are the BiHeap graphs on 2, 4, 6, 8, 10, 12, 14, 16, and 18 nodes, respectively.

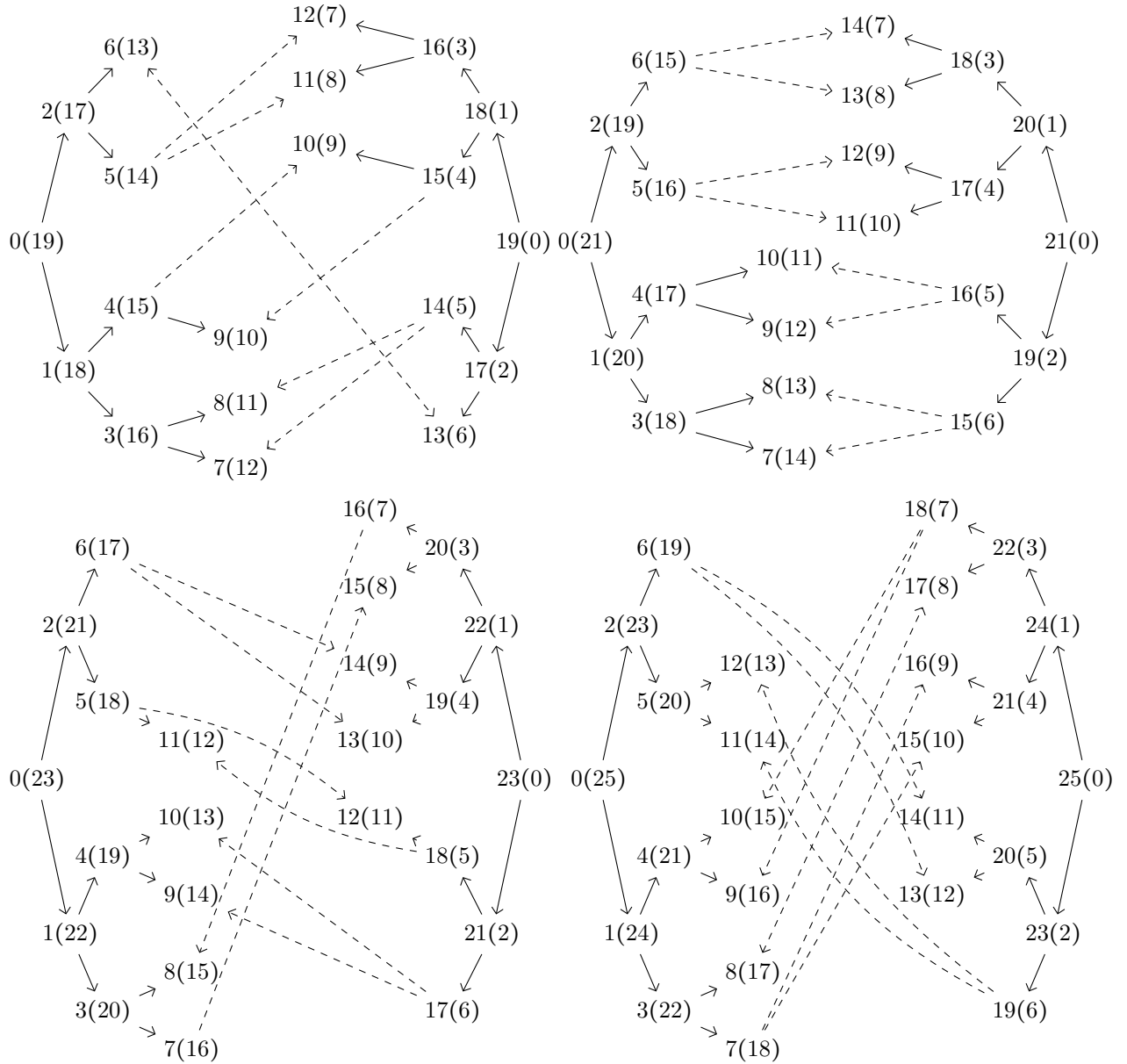


FIGURE 3. From left to right and from the top down, these are the BiHeap graphs on 20, 22, 24, and 26 nodes, respectively.

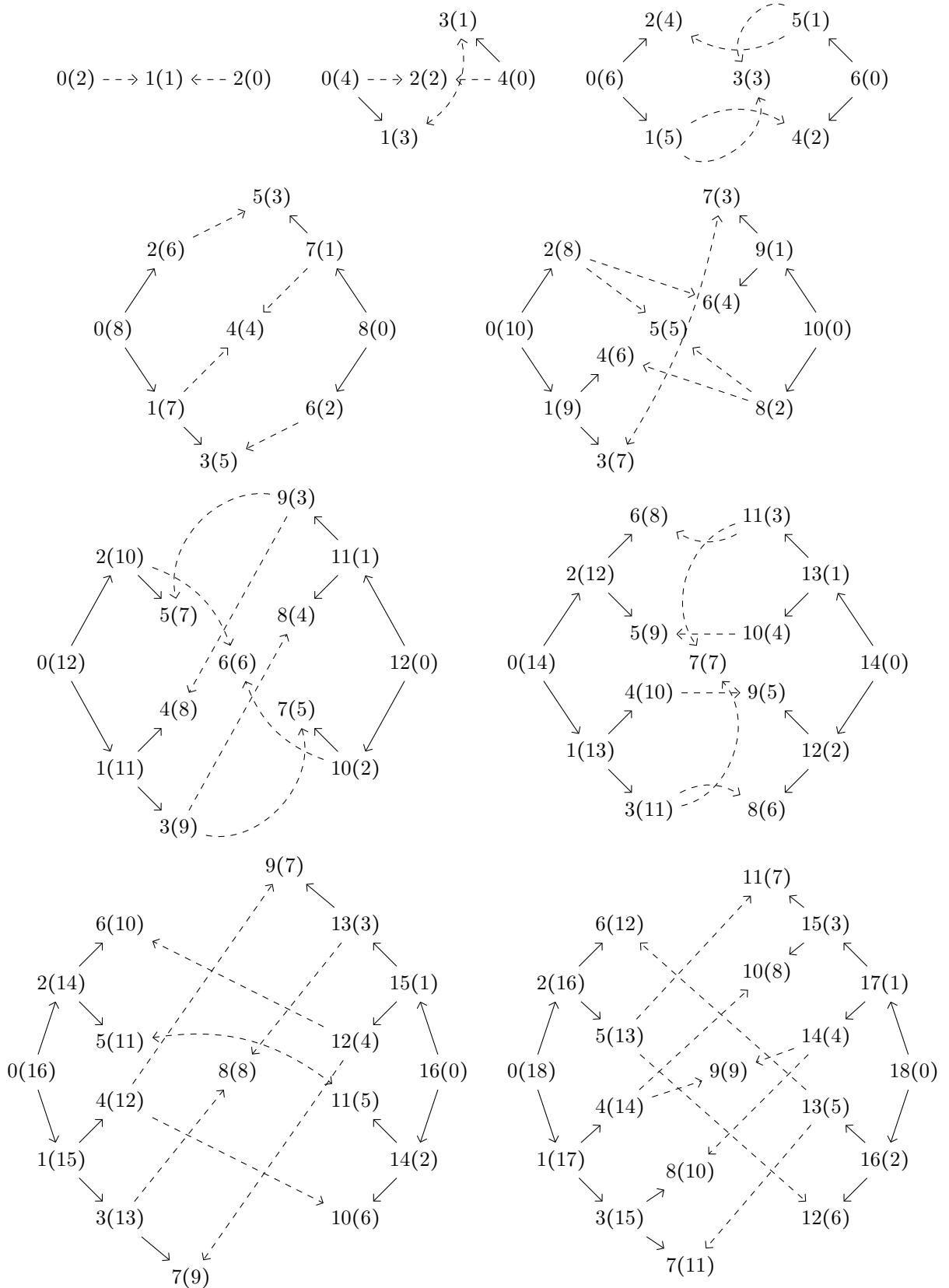


FIGURE 4. From left to right and from the top down, these are the BiHeap graphs on 3, 5, 7, 9, 11, 13, 15, 17, and 19 nodes, respectively. The subgraph consisting of the middle node and all nodes on the left (resp. right) half of the graph together with the solid arrows between these nodes form the “pure” min (resp. “pure” max) heaps of these BiHeap graphs.

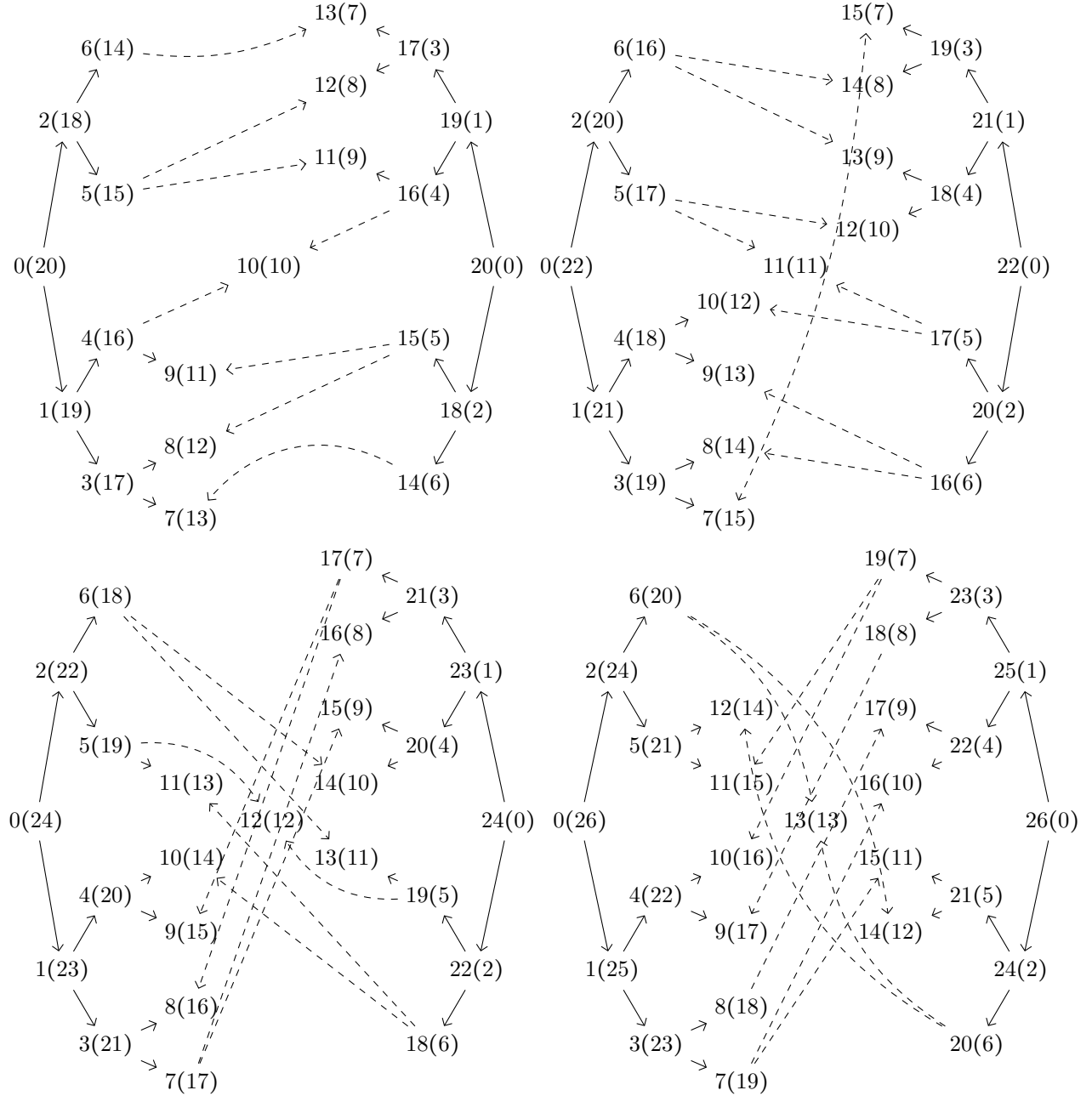


FIGURE 5. From left to right and from the top down, these are the BiHeap graphs on 21, 23, 25, and 27 nodes, respectively.

6. Basic Properties of BiHeaps. We now make some observations that are clear from the definition of a BiHeap. These observations and equations will henceforth be assumed to be known and used without comment.

Lemma 6.1. Let v and w be two nodes in B_N with coordinates (i, m) and (j, n) , respectively (so that by definition, $m = \text{Flip}(i)$ and $n = \text{Flip}(j)$) and recall that we identify a node with its coordinates. The following statements are true.:

- (1) v belongs to the min heap (resp. pure min heap, pure* min heap) if and only if the node with coordinate (m, i) belongs to the max heap (resp. pure max heap, pure* max heap).
- (2) $j - i = m - n$.
- (3) All equalities in the left column are equivalent and all equalities in the right column are equivalent.

$w = \text{LeftChild}_{\text{Min}}(v)$	$w = \text{RightChild}_{\text{Min}}(v)$
$(j, n) = \text{LeftChild}_{\text{Min}}(i, m)$	$(j, n) = \text{RightChild}_{\text{Min}}(i, m)$
$(n, j) = \text{LeftChild}_{\text{Max}}(m, i)$	$(n, j) = \text{RightChild}_{\text{Max}}(m, i)$
$j = \text{LeftChild}(i)$	$j = \text{RightChild}(i)$
$N = n + \text{RightChild}(i)$	$N = n + \text{RightChild}(i) + 1$
$j + \text{LeftChild}(m) = 2N$	$j + \text{LeftChild}(m) = \text{LeftChild}(N)$
$2m = n + N$	$2m = n + N + 1$
$m - n = i + 1$	$m - n = i + 2$
$m + j = N + i$	$m + j = N + i + 1$

- (4) There is at most one pure* min heap interior node that has a impure min heap edge leaving it along with a impure max heap edge entering it (since otherwise the max heap would have a cycle). Hence, if such a min heap edge goes from v to w then the min heap coordinate of v , call it i , satisfies $i \geq \lfloor ([N/2] - 1)/2 \rfloor$, where the right hand side is the coordinate of the last min heap interior node.
- (5) If v belongs to the min heap, $i \leq j$, and there exists a impure edge in the BiHeap that is incident to both v and w then, so as not to violate the min heap condition, the following statements must hold:
 - (a) v must belong to the pure min heap and w to the pure max heap.
 - (b) v is either a leaf of the pure min heap or otherwise there exists a unique pure min heap interior node with exactly 1 child in the pure min heap and v is this interior node. Consequently, letting $\rho = \text{PureHeapSize}(N)$ and $f = \lfloor (\rho - 1)/2 \rfloor$, we have,
 - (i) The first pure min heap node (meaning the node with the smallest min heap coordinate) to be adjacent in the BiHeap to a pure max heap node is the node with coordinates $(f, \text{Flip}(f))$.
 - (ii) If $k > \text{Flip}(f)$ then node $(k, \text{Flip}(k))$ belongs to the interior of the pure max heap and it is not adjacent to any pure min heap node.
 - (iii) $i \geq f$, $j > \rho$, and $j \leq \text{Flip}(f)$.
 - (iv) If v has exactly one child in the pure min heap then $i = f$ and $j = \rho$.
 - (v) If v is a pure min heap leaf node then $i \geq f$, where this inequality is strict if ρ is even.
 - (c) If (k, p) are the coordinates of any min heap node that has an edge incident to w then $i = k$ and $m = p$. That is, there is at most one impure min heap edge incident to any give pure max heap node.

Lemma 6.2. Let v be the first leaf of the pure min heap (i.e. the pure min heap leaf with the smallest min heap coordinate) and let w be the first leaf of the pure max heap. Let (i, m) and (j, n) be the coordinates of v and w , respectively, and let $\rho = \text{PureHeapSize}(N) = \lfloor \frac{N}{2} \rfloor$ be the size of the pure min heap. Then

$$i = n = \lfloor \rho/2 \rfloor = \left\lfloor \frac{N}{4} \right\rfloor = \left\lceil \frac{N-3}{4} \right\rceil$$

and

$$j = m = \text{Flip}(\lfloor \rho/2 \rfloor) = \left\lfloor \frac{3N-1}{4} \right\rfloor = \frac{3N + (N \bmod 4) - 4}{4}$$

If (i', m') (resp. (j', n')) is the coordinate of the first leaf of the pure* min (resp. pure* max) heap then

$$i' = n' = \lfloor \epsilon/2 \rfloor = \left\lfloor \frac{N+1}{4} \right\rfloor = \left\lceil \frac{N-2}{4} \right\rceil = \frac{1}{4} (N-2 + ((3N+2) \bmod 4))$$

$$j' = m' = \text{Flip}(\lfloor \epsilon/2 \rfloor) = \frac{1}{4} (3N-2 - ((3N+2) \bmod 4))$$

where $\epsilon := \text{ExtPureHeapSize}(N) = \lceil \frac{N}{2} \rceil$.

Proof. Recall that a complete binary tree with ρ nodes has exactly $\lfloor \rho/2 \rfloor$ interior nodes, from which it follows that the first leaf in the pure min heap has min heap coordinate $\lfloor \rho/2 \rfloor$. Consequently, the first leaf in the pure max heap has min heap coordinate $j = \text{Flip}(\lfloor \rho/2 \rfloor)$. One can verify that $\lfloor \rho/2 \rfloor = \lfloor \frac{N}{4} \rfloor = \frac{1}{4}(N - (N \bmod 4))$ by going through four cases based on the value of $N \bmod 4$. Since $j = \text{Flip}(i)$, it follows that $j = N - 1 - \frac{1}{4}(N - (N \bmod 4)) = \frac{1}{4}(3N + (N \bmod 4) - 4)$. By the symmetry in the construction of the BiHeap graph, we have that $i = n$ and $j = m$. The proof of the statement regarding the pure* heaps is similar. ■

Corollary 6.3. Let (i, m) be the last interior node of the pure min heap, let (j, n) be the last interior node in the pure max heap, and let $\rho = \text{PureHeapSize}(N)$ be the size of the pure heap. Then

$$i = n = \lfloor \rho/2 \rfloor - 1 = \left\lfloor \frac{N}{4} \right\rfloor - 1 = \frac{1}{4}(N - (N \bmod 4) - 4)$$

and

$$j = m = \text{Flip}(\lfloor \rho/2 \rfloor - 1) = \left\lceil \frac{3N}{4} \right\rceil = \frac{1}{4}(3N + (N \bmod 4)).$$

If (i', m') (resp. (j', n')) is the coordinate of the first leaf of the pure* min (resp. pure* max) heap and if $\epsilon = \text{ExtPureHeapSize}(N)$ then

$$i' = n' = \lfloor \epsilon/2 \rfloor - 1 = \frac{1}{4}(N - 6 + ((3N + 2) \bmod 4))$$

and

$$j' = m' = \text{Flip}(\lfloor \epsilon/2 \rfloor - 1) = \frac{1}{4}(3N + 2 - ((3N + 2) \bmod 4)).$$

Proof. The first leaf in the pure min heap comes immediately after the pure min heap's last interior node. ■

Part 3. Classification of BiHeaps

One way to classify BiHeaps is by whether or not they have a “middle node” (i.e. based on whether the BiHeap's contains an even or odd number of nodes), which is the content of proposition 7.2. Another way to classify BiHeaps is to group them depending on whether or not they have interior impure edges (def. 3.5(7)), which is the content of proposition 7.7. And yet a third more complicated way to classify them is by the value of their number of nodes moduloed by 3. These three classifications broadly describe the most important distinguishing features of BiHeap graphs and they are independent of each other so that each BiHeap falls into exactly one of twelve categories.

7. Even or Odd Number Of Nodes.

Proposition 7.1. N is even if and only if the pure* min heap and the pure* max heap are disjoint.

Proof. Immediate from the definitions of the pure* min and pure* max heaps. ■

Proposition 7.2. Suppose that there exists a node v in the min heap having two min heap children that belong to the max heap (such a v exists if and only if $N > 3$ and $N \neq 6$). Then N is even if and only if $\text{Parent}_{\text{Max}}(\text{LeftChild}_{\text{Min}}(v)) = \text{Parent}_{\text{Max}}(\text{RightChild}_{\text{Min}}(v))$.

Proof. Let (i, m) be the coordinates of v . Note that $\text{Parent}(\text{Flip}(\text{LeftChild}(i))) = \lfloor \frac{N-2i-3}{2} \rfloor$ while $\text{Parent}(\text{Flip}(\text{RightChild}(i))) = \lfloor \frac{N-2i-4}{2} \rfloor$ so that if N is even then these are both equal to $\frac{N}{2} - i - 2$ while if N is odd then they are not equal. ■

Corollary 7.3. Suppose that there exists a node belonging to both the min heap and the max heap having a sibling in the min heap and having a sibling in the max heap. Then N is odd if and only if the sibling in the min heap is different from the sibling in the max heap.

Definition 7.4. A graph is a *rectangle* (resp. *hexagon*) if it is a simple cycle on four (resp. six) nodes with four (resp. six) edges. A graph is said to *contain a rectangle* if there exists a subgraph that is a rectangle.

Using the illustrations of the BiHeap graphs given above, we obtain the following corollary to proposition 7.2.

Corollary 7.5. N is even if and only if the BiHeap graph on N nodes is a single edge, a hexagon, or contains a rectangle.

7.1. BiHeaps with Interior-Incident Impure* Edges. Another way to classify BiHeaps is to group them depending on whether or not they have an interior-incident impure* edges (def. 3.7(8)), which is the content of proposition 7.7.

Proposition 7.6. Let $\rho = \text{PureHeapSize}(N) = \lfloor N/2 \rfloor$ be the size of the pure heap. Then the following are equivalent:

- (1) ρ is even.
- (2) 4 divides N or 4 divides $N+3$.
- (3) there exists an interior-incident impure edge (def. 3.7(8)).
- (4) there exists exactly one interior-incident impure min edge and/or there exists exactly one interior-incident impure max edge.

If this is the case and if the unique interior-incident impure min edge goes from the pure* min heap interior node (i, m) to the max heap node (j, n) then these coordinates are determined as follows:

$$\begin{aligned}
 i &= \left\lfloor \frac{N}{4} \right\rfloor - 1 &= \frac{1}{4} (N - (N \bmod 4) - 4) \\
 m &= \left\lceil \frac{3N}{4} \right\rceil &= \frac{1}{4} (3N + (N \bmod 4)) \\
 j &= 2 \left\lfloor \frac{N}{4} \right\rfloor &= \frac{1}{2} (N - (N \bmod 4)) = \text{RightChild}(\rho/2 - 1) \\
 n &= 2 \left\lfloor \frac{N}{4} \right\rfloor - 1 + (N \bmod 4) &= \frac{1}{2} (N + (N \bmod 4)) - 1
 \end{aligned}$$

Proof. The equivalence of (1), (2), and (3) is obvious from the definitions of the BiHeap graph, the heaps, the pure heaps, and from the fact that a complete binary tree with ρ nodes has exactly $\lfloor \rho/2 \rfloor$ interior nodes. Everything else follows from lemma 6.2 or the fact that $j = \text{RightChild}_{\text{Min}}(i)$ and $n = \text{Flip}(j)$. ■

Proposition 7.7. Let $\epsilon = \text{ExtPureHeapSize}(N) = \lceil N/2 \rceil$ be the size of the pure* heap. Then the following are equivalent:

- (1) ϵ is even.
- (2) 4 divides N or 4 divides $N+1$.
- (3) $N = 3$ or else there exists an interior-incident impure* edge (def. 3.7(8)).
- (4) $N = 3$ or else there exists exactly one interior-incident impure* min edge and/or there exists exactly one interior-incident impure* max edge.

If this is the case and if the unique interior-incident impure* min edge goes from the pure* min heap interior node (i, m) to the max heap node (j, n) then these coordinates are determined as follows:

$$\begin{aligned}
 i &= \left\lfloor \frac{N+1}{4} \right\rfloor - 1 &= \frac{1}{4} (N - 6 + ((3N+2) \bmod 4)) \\
 m &= \left\lfloor \frac{3N}{4} \right\rfloor &= \frac{1}{4} (3N + 2 - ((3N+2) \bmod 4)) \\
 j &= 2 \left\lfloor \frac{N+1}{4} \right\rfloor &= \frac{1}{2} (N - 2 + ((3N+2) \bmod 4)) = \text{RightChild}(\epsilon/2 - 1) \\
 n &= 2 \left\lfloor \frac{N+1}{4} \right\rfloor - 2 &= \frac{1}{2} (N - ((3N+2) \bmod 4)) - 1
 \end{aligned}$$

Proof. The equivalence of (1), (2), and (3) is obvious from the definitions of the BiHeap graph, the heaps, the pure heaps, and from the fact that a complete binary tree with ϵ nodes has exactly $\lfloor \epsilon/2 \rfloor$ interior nodes. Everything else follows from lemma 6.2 or the fact that $j = \text{RightChild}_{\text{Min}}(i)$ and $n = \text{Flip}(j)$. ■

8. Classification by Value of Number of Nodes mod 3.

8.1. BiHeaps whose Heaps Share an Edge.

Proposition 8.1. The following are equivalent:

- (1) There exist nodes w and x such that $x \in \text{Child}_{\text{Min}}(w)$ in the min heap and $w \in \text{Child}_{\text{Max}}(x)$ in the max heap.
- (2) 3 divides $N - 2$ (or equivalently, $N \bmod 3 = 2$).

If this is the case and if w and x have coordinates (i, m) and (j, n) , respectively, then

$$i = n = \frac{N-2}{3} \quad \text{and} \quad j = m = \frac{2N-1}{3} = 2i + 1$$

so that, in particular, i, n, j, m, w , and x are uniquely determined and furthermore,

- (a) $x = \text{LeftChild}_{\text{Min}}(w)$ and $w = \text{LeftChild}_{\text{Max}}(x)$.
- (b) The pure* min heap node w is the last node in the *max* heap and the pure* max heap node x is the last node in the *min* heap.
- (c) A node (k, p) is in the max heap if and only if $k \geq \frac{N-2}{3}$.
- (d) A node (k, p) is in the min heap if and only if $k \leq \frac{2N-1}{3}$.
- (e) The number of pure* min heap leaves with exactly two children in the min heap is

$$\left\lfloor \frac{N+1}{12} \right\rfloor - 1 = \frac{N-2}{3} - \left\lfloor \frac{N+1}{4} \right\rfloor = \frac{1}{3} \left(\left\lfloor \frac{N}{4} \right\rfloor + (\epsilon \bmod 2) - 2 - (N \bmod 2) \right) = \left\lfloor \frac{N-2}{12} \right\rfloor = \left\lfloor \frac{N-1}{12} \right\rfloor = \left\lfloor \frac{N}{12} \right\rfloor$$

where $\epsilon = \left\lfloor \frac{N}{2} \right\rfloor$ is the size of the pure* min heap.

- (f) The number of pure min heap leaves with exactly two children in the min heap is

$$\left\lfloor \frac{N+4}{12} \right\rfloor - 1 = \frac{N-2}{3} - \left\lfloor \frac{N}{4} \right\rfloor = \frac{1}{3} \left(\left\lfloor \frac{N+2}{4} \right\rfloor + (\rho \bmod 2) - 1 \right) = \left\lfloor \frac{N+1}{12} \right\rfloor = \left\lfloor \frac{N+2}{12} \right\rfloor = \left\lfloor \frac{N+3}{12} \right\rfloor$$

where $\rho = \left\lfloor \frac{N}{2} \right\rfloor$ is the size of the pure min heap.

- (g) If $N > 2$ then N is even (resp. odd) if and only if $w = \text{RightChild}_{\text{Min}}(\text{Parent}_{\text{Min}}(w))$ (resp. $w = \text{LeftChild}_{\text{Min}}(\text{Parent}_{\text{Min}}(w))$).

Proof. Suppose for the sake of contradiction that $j = \text{LeftChild}(i)$ and $m = \text{RightChild}(n)$. Then $m - n = i + 1$ and $n + 2 = j - i$ so that $j = m + 1 = \text{Flip}(i) + 1 = N - i$. Now, $2i + 1 = \text{LeftChild}(i) = j = N - i$ so that $i = \frac{N-1}{3}$ and $j = \text{LeftChild}(i) = \frac{2(N-1)}{3} + 1$, from which we conclude that $N = \frac{3(j-1)}{2} + 1$. By symmetry, there also

exist nodes w' and x' , with coordinates (i', m') and (j', n') , respectively, such that $x' = \text{LeftChild}_{\text{Max}}(w')$ and $w' = \text{RightChild}_{\text{Min}}(x')$. This implies that $n' = \text{LeftChild}(m')$ and $i' = \text{RightChild}(j')$. Proceeding as before, we can obtain $n' + m' = N$, $m' = \frac{N-1}{3}$, and $n' = \frac{2(N-1)}{3} + 1$, thus showing that $m' = i$ and $n' = j$. Now $j = \text{Flip}(i) = \text{Flip}(m') = \text{RightChild}(j') = 2(j' + 1)$, which shows that j is even. But $j = \text{LeftChild}(i) = 2i + 1$ so that j is also odd. An analogous argument obtains a contradiction from the assumption that $j = \text{RightChild}(i)$ and $m = \text{LeftChild}(n)$.

Now suppose for the sake of contradiction that $j = \text{RightChild}(i)$ and $m = \text{RightChild}(n)$. The former happens if and only if $m - i = n + 2$ while the latter happens if and only if $n + 2 = j - i$, giving us $m = j$ from which $i = n$ follows. Now $2i + 2 = \text{RightChild}(i) = m = \text{Flip}(i) = N - 1 - i$ so that $N = 3(i + 1)$. But we also have that $j = m = \text{RightChild}(n) = \text{RightChild}(\text{Flip}(j))$, which gives us $j = 2N - 2j$. It follows that $3j = 2N$, $j = 2(i + 1)$ so that j is even and $j = \frac{2N}{3}$. Now $i = \text{Parent}(j) = j/2 = i + 1$, giving us a contradiction.

Now suppose that $j = \text{LeftChild}(i)$ and $m = \text{LeftChild}(n)$. As before, we can obtain $m = j$ and $i = n$ from which we conclude that $N - 1 - i = \text{Flip}(i) = j = m = \text{LeftChild}(n) = \text{LeftChild}(i) = 2i + 1$. This shows that $N = 3i + 2$ and $i = \frac{N-2}{3}$. Using $m = \text{LeftChild}(n) = \text{LeftChild}(\frac{N-2}{3})$ we obtain $m = \frac{2N-1}{3}$.

We've thus proven both (a) and (1) \implies (2). To prove (2) \implies (1), let $i = \frac{N-2}{2}$ and $j = \text{LeftChild}(i)$, where of course their mirror coordinates are $m = \text{Flip}(i)$ and $n = \text{Flip}(j)$. We must show that $m = \text{LeftChild}(n)$. Expanding $\text{LeftChild}(n) = \text{LeftChild}(\text{Flip}(\text{LeftChild}(i)))$ gives us $\text{LeftChild}(n) = \frac{2N-1}{3}$ while expanding $m = \text{Flip}(i) = \text{Flip}(\frac{N-2}{3})$ also gives us $\frac{2N-1}{3}$, as desired.

Suppose for the sake of contradiction that x has a max heap sibling s that was also in the min heap. Since x is its max heap parent's left child, the min heap coordinate of s is necessarily $\text{RightChild}(\text{Parent}(m)) = \text{RightChild}(n) = \text{LeftChild}(n) + 1 = j + 1$. Let $k = j + 1$ and $p = \text{Flip}(k)$ so that s 's coordinates are (k, p) and $p = N - 1 - (j + 1) = \text{Flip}(j) - 1 = n - 1$. Note that (b) is clearly true for $N = 2$ and $N = 5$ so assume that $N > 5$. Since $N > 5$ we have $p < n$ and since there exists a max heap edge leaving (j, n) , by the symmetry in the construction of the BiHeap graph, there must also be a max heap edge leaving node (k, p) and going into a min heap node. Thus the restriction of the BiHeap graph, B_N , to node $\{0, \dots, j + 1\}$ contains the cycle:

$$w \rightarrow (k, p) \rightarrow \text{RightChild}_{\text{Max}}(k, p) \rightarrow (0, N - 1) \rightarrow w$$

contradicting the definition of the min heap of the BiHeap graph. Since x has no max heap sibling that is also in the min heap, it is therefore the last node in the min heap, which proves (b). Parts (c) and (d) follow immediately from what has already been shown.

To prove (e), let χ equal 1 if $\epsilon := \lceil \frac{N}{2} \rceil$ is even and 0 otherwise, let x be the desired quantity, and note that the number of nodes that are either pure* min heap leaves or pure* max heap leaves is equal to both $6x + 2 + 2\chi + (N \bmod 2)$ and $2\lceil \frac{\epsilon}{2} \rceil - (N \bmod 2)$. Solving for x and using the formula $\lceil \frac{\epsilon}{2} \rceil = \lceil \frac{N}{4} \rceil$ gives us (e). Now note that the min heap's last node (i.e. the node with the largest min heap coordinate) that could possibly have two children has min heap coordinate $\frac{N-2}{3} - 1$ whereas the first leaf of the pure* min heap has min heap coordinate $\text{Parent}(\epsilon - 1) + 1 = \lfloor \frac{\epsilon}{2} \rfloor = \lfloor \frac{N+1}{4} \rfloor$. From this it follows that the pure* min heap has $\frac{N-2}{3} - \lfloor \frac{N+1}{4} \rfloor = \lceil \frac{N+1}{12} \rceil - 1$ leaves that have exactly two children in the min heap. Since $N \bmod 3 = 2$, we must have $N \bmod 12 = 2, 5, 8$, or 11 and going through each of these cases allows us to conclude that $x = \lfloor \frac{N-2}{12} \rfloor = \lfloor \frac{N-1}{12} \rfloor = \lfloor \frac{N}{12} \rfloor$.

The proof of (f) is similar where we find that the number of nodes that neither belong to the interior of the pure min heap nor to the interior of the pure max heap is equal to both $6x + 2 + (N \bmod 2) - 2(\rho \bmod 2)$ and $2\lceil \frac{\rho}{2} \rceil + (N \bmod 2)$. Solving for x and using the formula $\lceil \frac{\rho}{2} \rceil = \lfloor \frac{N+2}{4} \rfloor$ gives us $x = \frac{1}{3} (\lfloor \frac{N+2}{4} \rfloor + (\rho \bmod 2) - 1)$. Since $N \bmod 3 = 2$, we must have $N \bmod 12 = 2, 5, 8$, or 11 and going through each of these cases allows us to conclude that $x = \lfloor \frac{N+1}{12} \rfloor = \lfloor \frac{N+2}{12} \rfloor = \lfloor \frac{N+3}{12} \rfloor$.

Now note that the first leaf of the pure min heap has min heap coordinate $\text{Parent}(\rho - 1) + 1 = \lfloor \frac{\rho}{2} \rfloor = \lfloor \frac{N}{4} \rfloor$, where $\rho = \lfloor \frac{N}{2} \rfloor$, so that as in the proof of (e), it follows that the pure min heap has $\frac{N-2}{3} - \lfloor \frac{N}{4} \rfloor = \lceil \frac{N+4}{12} \rceil - 1$ leaves that have exactly two children in the min heap. Now note that $\text{Parent}(i) = \text{Parent}(\frac{N-2}{3}) = \lfloor (\frac{N-2}{3} - 1)/2 \rfloor$ so that if N is even (resp. odd) then $\text{Parent}(i) = \frac{N-8}{6}$ (resp. $\text{Parent}(i) = (\frac{N-2}{3} - 1)/2$) from which we immediately obtain that $i = \text{RightChild}(\text{Parent}(i))$ (resp. $i = \text{LeftChild}(\text{Parent}(i))$). Since a node can

either be a left or a right child of its parent, this proves (g). ■

8.2. BiHeaps with Two Distinct Leaf Impure* Edges Having Out-Degree 1.

Proposition 8.2. The following are equivalent:

- (1) There exist a node v in the min heap with exactly one min heap child w and the edge formed by any such pair does not belong to the max heap.
- (2) 3 divides N .
- (3) There exists a pure min heap leaf node v that does not belong to the max heap and that has a unique min heap child w .

For $N \neq 3$, we can add the following to the above list:

- (4) There exists a pure* min heap leaf node v such that v has a unique min heap child w such that v is not a max heap child of w .

In this case, the v and w given in (4) are the same v and w given in (1). If (1) holds and if such v and w have coordinates (i, m) and (j, n) , respectively, then

$$i = \frac{N}{3} - 1, \quad j = \frac{2}{3}N - 1, \quad m = \frac{2}{3}N, \quad \text{and} \quad n = \frac{N}{3}$$

so that, in particular, i, j, m, n, v , and w are uniquely determined and furthermore,

- (a) $w = \text{LeftChild}_{\text{Min}}(v)$ and w has no sibling in the min heap.
- (b) N is even if and only if $v = \text{LeftChild}_{\text{Min}}(\text{Parent}_{\text{Min}}(v))$.
- (c) If $N > 3$ then N is odd if and only if $v = \text{RightChild}_{\text{Min}}(\text{Parent}_{\text{Min}}(v))$.
- (d) The pure* min heap node v is the last node in the min heap with any outgoing edges (i.e. for any node (k, p) , if $k > i$ then (k, p) does not have any children in the min heap).
- (e) A node (k, p) is in the max heap if and only if $k \geq \frac{N}{3}$.
- (f) A node (k, p) is in the min heap if and only if $k < \frac{2N}{3}$.
- (g) If $N \neq 3$, then the number of pure* min heap leaves with exactly two children in the min heap is

$$\left\lfloor \frac{N-3}{12} \right\rfloor - 1 = \frac{N}{3} - \left\lfloor \frac{N+1}{4} \right\rfloor - 1 = \frac{1}{3} \left(\left\lfloor \frac{N}{4} \right\rfloor - 3 + (\epsilon \bmod 2) - (N \bmod 2) \right) = \left\lfloor \frac{N-4}{12} \right\rfloor = \left\lfloor \frac{N-5}{12} \right\rfloor = \left\lfloor \frac{N-6}{12} \right\rfloor$$

where $\epsilon = \left\lfloor \frac{N}{2} \right\rfloor$ is the size of the pure* min heap.

- (h) The number of pure min heap leaves with exactly two children in the min heap is

$$\left\lfloor \frac{N}{12} \right\rfloor - 1 = \frac{N}{3} - \left\lfloor \frac{N}{4} \right\rfloor - 1 = \left\lfloor \frac{N-1}{12} \right\rfloor = \left\lfloor \frac{N-2}{12} \right\rfloor = \left\lfloor \frac{N-3}{12} \right\rfloor$$

Remark 8.3. Note that statement 4 does *not* say that v has only *one* min heap child.

Proof. Note that when $N = 3$ then (1), (2), (3), and (a) - (h) are all true and the formulas for i, j, m , and n hold while for $N = 1, 2, 4$, and 5 , statements (1) - (4) are all false. We may now henceforth assume that $N \geq 6$, so that the pure* min heap has at least one interior node and the min heap has at least one impure min node. Since $N \geq 6$, once we prove that (1) and (4) are equivalent and that the formulas for i, j, m , and n hold, it will then easily follow that (1), (2), (3), and (4) are all equivalent.

Suppose that v is a pure* min heap leaf node having exactly one min heap child w such that v is not a max heap child of w . Let (i, m) and (j, n) be the coordinates of v and w , respectively.

Assume (4) and suppose for the sake of contradiction that v has two min heap children, where we will call this second child z . Were v not a child of z in the max heap then the uniqueness property of w stated in (4) would be violated so that v must be a max heap child of z . But now condition (1) of proposition 8.1 is satisfied so by part (b) of that proposition, z must be both the last node of the min heap and the left child of v . These two properties are impossible to have due to w being z 's sibling in the min heap. Thus w is v 's

only child in the min heap. This implies that w is necessarily v 's left child in the min heap, which proves part (a). Since v has no right child, part (d) now follows immediately from the definition of the BiHeap graph on N nodes where note that the statement of part (d) implies that v is uniquely determined so that there can be at most one node in the min heap having v 's defining property.

By the symmetric construction of the BiHeap graph on N nodes (def. 2.4), there must also exist a (now known to be unique) pure* max heap leaf v' having exactly one max heap child w' such that v' is not a min heap child of w' . Let (i', m') and (j', n') be the coordinates of v' and w' , respectively. Since the node w' is the last node in the max heap, the symmetry of the construction of the BiHeap graph on N nodes implies that $n' = j$ and $j' = n$ so that $m' = \text{Parent}(n') = \lfloor \frac{j-1}{2} \rfloor = \lfloor \frac{(2i+1)-1}{2} \rfloor = i$, which implies $i' = \text{Flip}(m') = \text{Flip}(i) = m$. Note that if $j' \leq i$ then w' being in the max heap would necessitate that v , and hence also w (v 's child), belong to the max heap as well, which would cause the max heap to have a cycle. Thus $j' > i$.

Now suppose for the sake of contradiction that $j' > i + 1$. Let $k = i + 1$, $p = \text{Flip}(k)$, $p' = m + 1$, and $k' = \text{Flip}(p')$. Let z be the node whose coordinate is (k, p) and let z' be the node whose coordinate is (k', p') . Since j' is the min heap coordinate of w' , the last leaf of the max heap, and since $j' > i + 1 = k$, it follows that z does not belong to the max heap. Similarly, since n is the max heap coordinate of w , the last leaf of the min heap, and since $n = j' > i + 1 = m' + 1 = p'$, it follows that z' does not belong to the min heap. So by adding an edge between v and z' to the min heap and another edge from v' to z to the max heap, we could extend both the min heap and the max heap by one edge, thereby contradicting the maximality of the heaps in the definition (def. 2.4) of the BiHeap graph on N nodes. Thus $j' \leq i + 1$ so that $j' = i + 1$. From $i + 1 = j' = n = \text{Flip}(j) = \text{Flip}(\text{LeftChild}(i))$, we conclude that $i = \frac{N}{3} - 1$ so that in particular, 3 divides N , proving (2). From $j = \text{LeftChild}(i)$, $m = \text{Flip}(i)$, and $n = \text{Flip}(j)$ we obtain the formulas for j, m , and n .

Now suppose that 3 divides N and define i, j, m , and n by the formulas given in this proposition's statement. Let v be the node whose coordinates are (i, m) and let w be the node whose coordinates are (j, n) . It is straightforward to show that $j = \text{LeftChild}(i)$ and that $\text{Parent}(m) = \lfloor \frac{N}{3} - \frac{1}{2} \rfloor = \frac{N}{3} - 1 \neq \frac{N}{3} = n$, which shows that w is not the max heap parent of v .

Suppose that v has a min heap right child, call it z , whose coordinates would then necessarily be (k, p) where $k := j + 1$ and $p := n - 1$. Note that $\text{Parent}(m) = \lfloor \frac{N}{3} - \frac{1}{2} \rfloor = \frac{N}{3} - 1 = n - 1 = p$ so that v is z 's max heap child. Thus either v has no min heap right child or else its min heap right child is v 's max heap parent. Either way, it follows that w is the unique child of v such that v is not a max heap child of w , which finishes the proof that (2) implies (4).

To prove (b), note that since 3 divides N , $\text{Parent}(i) = \lfloor \frac{i-1}{2} \rfloor = \lfloor \frac{N}{6} \rfloor - 1$ so if N is even then $\text{Parent}(i) = \frac{N}{6} - 1$, which implies that $\text{LeftChild}(\text{Parent}(i)) = 2(\frac{N}{6} - 1) + 1 = \frac{N}{3} - 1 = i$. Conversely, $i = \text{Parent}(\text{LeftChild}(\text{Parent}(i)))$ implies $\frac{N}{3} - 1 = 2\lfloor \frac{N}{6} \rfloor - 1$ so that $\frac{N}{6} = \lfloor \frac{N}{6} \rfloor$, which shows that N is divisible by 6 and thus also by 2.

To prove (c), suppose first that N is odd. Since 2 does not divide $\frac{N}{3}$, $\lfloor \frac{N}{6} \rfloor = \lfloor \frac{N/3}{2} \rfloor = \frac{N/3}{2} - \frac{1}{2}$ so that $\text{RightChild}(\text{Parent}(i)) = \frac{N}{3} - 1 = i$. Conversely, if $i = \text{RightChild}(\text{Parent}(i))$ then $\frac{N}{3} - 1 = 2(\lfloor \frac{N}{6} \rfloor - 1) - 2$ gives us $\frac{N}{6} = \lfloor \frac{N}{6} \rfloor - \frac{1}{2}$ where since the right hand side is not an integer, neither is $\frac{N}{6}$ so that at least one of 2 and 3 does not divide N . Since 3 divides N , it follows that 2 does not divide N , so that N is odd.

To prove (g), let χ equal 1 if $\epsilon := \lfloor \frac{N}{2} \rfloor$ is even and 0 otherwise (so $\chi = 1 - (\epsilon \bmod 2)$), let x be the desired quantity, and note that the number of nodes that are either pure* min heap leaves or pure* max heap leaves is equal to both $6x + 4 + 2\chi + (N \bmod 2)$ and $2\lfloor \frac{\epsilon}{2} \rfloor - (N \bmod 2)$. Solving for x and using the formula $\lfloor \frac{\epsilon}{2} \rfloor = \lfloor \frac{N}{4} \rfloor$ gives us $x = \frac{1}{3}(\lfloor \frac{N}{4} \rfloor - 3 + (\epsilon \bmod 2) - (N \bmod 2))$. Now note that the min heap's last node (i.e. the node with the largest min heap coordinate) that could possibly have two children has min heap coordinate $\frac{N}{3} - 2$ whereas the first leaf of the pure* min heap has min heap coordinate $\text{Parent}(\epsilon - 1) + 1 = \lfloor \frac{\epsilon}{2} \rfloor = \lfloor \frac{N+1}{4} \rfloor$. From this it follows that the pure* min heap has $\frac{N}{3} - 1 - \lfloor \frac{N+1}{4} \rfloor = \lfloor \frac{N-3}{12} \rfloor - 1$ leaves that have exactly two children in the min heap. Since $N \bmod 3 = 0$, we must have $N \bmod 12 = 0, 3, 6$, or 9 and going through each of these cases allows us to conclude that $x = \lfloor \frac{N-4}{12} \rfloor = \lfloor \frac{N-5}{12} \rfloor = \lfloor \frac{N-6}{12} \rfloor$.

To prove (h), note that the first leaf of the pure min heap has min heap coordinate $\text{Parent}(\rho - 1) + 1 = \lfloor \frac{\rho}{2} \rfloor = \lfloor \frac{N}{4} \rfloor$, where $\rho = \lfloor \frac{N}{2} \rfloor$, so that as in the proof of (g), it follows that the pure min heap has $\frac{N}{3} - 1 - \lfloor \frac{N}{4} \rfloor = \lfloor \frac{N}{12} \rfloor - 1$

leaves that have exactly two children in the min heap. Since $N \bmod 3 = 0$, we must have $N \bmod 12 = 0, 3, 6$, or 9 and going through each of these cases allows us to conclude that $x = \lfloor \frac{N-1}{12} \rfloor = \lfloor \frac{N-2}{12} \rfloor = \lfloor \frac{N-3}{12} \rfloor$. ■

8.3. BiHeaps Whose Heaps Lack Interior Nodes With a Single Child.

Proposition 8.4. The following are equivalent:

- (1) There is no min heap node with exactly one child.
- (2) 3 divides $N-1$ (or equivalently, $N \bmod 3 = 1$).

For $N > 1$, we can add the following to the above list:

- (3) If v is the last node of the min heap then $v = \text{RightChild}_{\text{Min}}(\text{Parent}_{\text{Min}}(v))$.

Suppose that (2) holds, let v be the last node in the min heap, and let (i, m) be its coordinates. Then

$$i = \frac{2(N-1)}{3} \quad \text{and} \quad m = \frac{N-1}{3}.$$

so that, in particular, i, m , and v are uniquely determined and furthermore,

- (a) A node (k, p) is in the max heap if and only if $k \geq \frac{N-1}{3}$.
- (b) A node (k, p) is in the min heap if and only if $k \leq \frac{2(N-1)}{3} = 2m$.
- (c) The number of pure* min heap leaves with exactly two children in the min heap is

$$\left\lfloor \frac{N+5}{12} \right\rfloor - 1 = \frac{N-1}{3} - \left\lfloor \frac{N+1}{4} \right\rfloor = \frac{1}{3} \left(\left\lfloor \frac{N}{4} \right\rfloor - 1 + (\epsilon \bmod 2) - (N \bmod 2) \right) = \left\lfloor \frac{N+2}{12} \right\rfloor = \left\lfloor \frac{N+3}{12} \right\rfloor = \left\lfloor \frac{N+4}{12} \right\rfloor$$

where $\epsilon = \left\lceil \frac{N}{2} \right\rceil$ is the size of the pure* min heap.

- (d) The number of pure min heap leaves with exactly two children in the min heap is

$$\left\lfloor \frac{N+8}{12} \right\rfloor - 1 = \frac{N-1}{3} - \left\lfloor \frac{N}{4} \right\rfloor = \left\lfloor \frac{N+5}{12} \right\rfloor = \left\lfloor \frac{N+6}{12} \right\rfloor = \left\lfloor \frac{N+7}{12} \right\rfloor$$

Proof. If (1) hold then propositions 8.1 and 8.2 imply that $N \bmod 3 \neq 2$ and $N \bmod 3 \neq 0$, respectively, so that (2) necessarily holds. The converse also follows immediately from these two propositions. Similarly, these same propositions prove that (2) and (3) are equivalent.

To prove (c), let χ equal 1 if $\epsilon := \left\lceil \frac{N}{2} \right\rceil$ is even and 0 otherwise, let x be the desired quantity, and note that the number of BiHeap leaves is equal to both $6x + 2\chi + (N \bmod 2)$ and $2\left\lceil \frac{\epsilon}{2} \right\rceil - (N \bmod 2)$. This gives us $x = \frac{1}{3} \left(\left\lfloor \frac{N}{4} \right\rfloor - 1 + (\epsilon \bmod 2) - (N \bmod 2) \right)$. Since $N \bmod 3 = 1$, we must have $N \bmod 12 = 1, 4, 7$, or 10 and going through each of these cases allows us to conclude that $x = \left\lfloor \frac{N+4}{12} \right\rfloor = \left\lfloor \frac{N+3}{12} \right\rfloor = \left\lfloor \frac{N+2}{12} \right\rfloor$. Now note that the min heap's last node (i.e. the node with the largest min heap coordinate) that could possibly have two children has min heap coordinate $\frac{N-1}{3} - 1$ whereas the first leaf of the pure* min heap has min heap coordinate $\text{Parent}(\epsilon - 1) + 1 = \left\lfloor \frac{\epsilon}{2} \right\rfloor = \left\lfloor \frac{N+1}{4} \right\rfloor$. From this it follows that the pure* min heap has $\frac{N-1}{3} - \left\lfloor \frac{N+1}{4} \right\rfloor = \left\lfloor \frac{N+5}{12} \right\rfloor - 1$ leaves that have exactly two children in the min heap.

To prove (d), note that the min heap last node (i.e. the node with the largest min heap coordinate) that could possibly have two children has min heap coordinate $\text{Parent}\left(\frac{2(N-1)}{3}\right) = \left\lfloor \frac{2N-5}{6} \right\rfloor$ where since either $N \bmod 6 = 1$ or 4, this becomes $\frac{2N-8}{6} = \frac{N-1}{3} - 1$. On the other hand, the first leaf of the pure min heap has min heap coordinate $\text{Parent}(\rho - 1) + 1 = \left\lfloor \frac{\rho}{2} \right\rfloor = \left\lfloor \frac{N}{4} \right\rfloor$, where $\rho = \left\lfloor \frac{N}{2} \right\rfloor$. From this it follows that the pure min heap has $\frac{N-1}{3} - \left\lfloor \frac{N}{4} \right\rfloor = \left\lfloor \frac{N-4}{12} \right\rfloor$ leaves that have exactly two children in the min heap. Since $N \bmod 3 = 1$, we must have $N \bmod 12 = 1, 4, 7$, or 10 and going through each of these cases allows us to conclude that $x = \left\lfloor \frac{N+5}{12} \right\rfloor = \left\lfloor \frac{N+6}{12} \right\rfloor = \left\lfloor \frac{N+7}{12} \right\rfloor$.

Assume now that v and (i, m) are as described above and let $\epsilon = \lceil \frac{N}{2} \rceil$. Let us determine how many nodes are in the min heap. Let

$$\chi = \begin{cases} 1 & \text{if } \epsilon \text{ is even} \\ 0 & \text{if } \epsilon \text{ is odd} \end{cases} \quad \text{and} \quad \alpha = \begin{cases} 0 & \text{if } N \text{ is even} \\ 1 & \text{if } N \text{ is odd} \end{cases}$$

Let τ be thrice the number of pure* min heap leaves having exactly two children in the min heap. Note that

- (1) Every pure* min heap leaf is incident to either 2 or 0 impure min heap edges.
- (2) If ϵ is even then by proposition 7.7, node with min heap coordinate $\text{Parent}(\epsilon - 1)$ has exactly one impure min heap edge going to the last node of the pure* max heap (where if N is odd, this node is also not the middle node), and by the symmetry of the BiHeap construction, there is a similar edge going from the last interior pure* max heap node into the last (non-middle) leaf of the pure* min heap.
- (3) If N is odd, then the node with min heap coordinate $\epsilon - 1$ is incident to both a pure* min edge and a pure* max edge.

From this we conclude that the min heap has exactly $\mu := \epsilon + \frac{2}{3}\tau + \chi$ nodes and that $\tau = \lceil \frac{\epsilon}{2} \rceil - \alpha - \chi$, as we now explain. Recall that $\lceil \frac{\epsilon}{2} \rceil$ is the number of leaves in the pure* min heap, we subtract 1 if the pure* min heap and max heap share a node, and we also subtract 1 if there is a edge going from the last interior pure* max heap node into the last pure* min heap node. Note that among the τ nodes that were counted, for every node incident to 2 impure min heap edges, there are two such nodes not incident to any min heap edge, which is why the $\frac{2}{3}\tau$ was added while χ was added to count, in the case where ϵ is even, the last pure* max heap node (which will necessarily belong to the min heap). From this and the formula $\lceil \frac{\epsilon}{2} \rceil = \lceil \frac{N}{4} \rceil$ we get (c).

We will now go through the four cases of ϵ and N being even or odd to show that $\mu = \frac{2N+1}{3}$ in each of them. If ϵ is even then $\mu = \epsilon + \frac{2}{3}(\epsilon/2 - 1 - \alpha) + 1 = \frac{4\epsilon+1-2\alpha}{3}$ so if N is even then $\epsilon = \frac{N}{2}$ so $\mu = \frac{2N+1}{3}$ while if N is odd then $\epsilon = \frac{N+1}{2}$ so $\mu = \frac{2(N+1)+1-2}{3} = \frac{2N+1}{3}$. If ϵ is odd then $\tau = \frac{\epsilon+1}{2} - \alpha$ and $\mu = \epsilon + \frac{2}{3}(\frac{\epsilon+1}{2} - \alpha) = \frac{4\epsilon+1-2\alpha}{3}$ so if N is even then $\mu = \frac{2N+1}{3}$ while if N is odd then $\mu = \frac{2N+1}{3}$. This implies the formula for i , which in turn gives the formula for $m = \text{Flip}(i)$ and statements (a) and (b). ■

9. Corollaries of the Classification of BiHeap Graphs.

9.1. Computing HeapSize(N). From propositions 8.1, 8.2, and 8.4 we conclude that for all $N > 0$,

$$\begin{aligned} \text{HeapSize}(N) &= N - \frac{N - (N \bmod 3)}{3} \\ &= N - \lfloor N/3 \rfloor \\ &= \left\lceil \frac{2N}{3} \right\rceil \\ &= \frac{2N + (N \bmod 3)}{3} \end{aligned}$$

9.2. Other Corollaries. We now list some other important corollaries to propositions 8.1, 8.2, and 8.4.

- (1) $\text{HeapSize}(N)$ is odd if and only if $N \bmod 3$ is odd.
- (2) i belongs to the max heap if and only if $\text{Flip}(i) < \text{HeapSize}(N)$, which holds if and only if $\frac{N - (N \bmod 3)}{3} \leq i$.
- (3) The first leaf of the min heap has min heap coordinate

$$\text{FirstLeaf}(N) := \left\lceil \frac{\text{HeapSize}(N)}{2} \right\rceil = \left\lceil \frac{2N + (N \bmod 3)}{6} \right\rceil = \frac{N - (N \bmod 3)}{3} + \left\lceil \frac{N \bmod 3}{2} \right\rceil$$

- (4) Items (2) and (3) imply that every leaf of the min heap belongs to the max heap. The symmetry of the BiHeap construction implies that every leaf of the max heap belongs to the min heap.

- (5) For $N > 1$, the parent of the last min heap node has min heap coordinate $\lfloor \frac{N-2}{3} \rfloor$ and max heap coordinate $\lfloor \frac{2N-1}{3} \rfloor$.
- (6) The first leaf of the min heap has min heap coordinate $\lfloor \frac{N+1}{3} \rfloor$.
- (7) For $N \neq 3$, the number of pure* min heap leaves with exactly two children in the min heap is

$$\begin{aligned} \left\lfloor \frac{N-6+4((2N) \bmod 3)}{12} \right\rfloor &= \left\lfloor \frac{N-5+4((2N) \bmod 3)}{12} \right\rfloor = \left\lfloor \frac{N-4+4((2N) \bmod 3)}{12} \right\rfloor \\ &= \left\lfloor \frac{N-3+4((2N) \bmod 3)}{12} \right\rfloor - 1 \end{aligned}$$

and for $N > 3$, the number of pure* min heap leaves with exactly two children in the min heap is

$$\frac{1}{3} \left(\left\lfloor \frac{N+1}{4} \right\rfloor - (N \bmod 2) - 1 + (\epsilon \bmod 2) - ((N+2) \bmod 3) \right)$$

while it is 0 for $N \leq 3$.

- (8) The number of pure min heap leaves with exactly two children in the min heap is

$$\begin{aligned} \left\lfloor \frac{N-3+4((2N) \bmod 3)}{12} \right\rfloor &= \left\lfloor \frac{N-2+4((2N) \bmod 3)}{12} \right\rfloor = \left\lfloor \frac{N-1+4((2N) \bmod 3)}{12} \right\rfloor \\ &= \left\lfloor \frac{N+4((2N) \bmod 3)}{12} \right\rfloor - 1 \end{aligned}$$

- (9) For $N \neq 3$, the number of pure* min heap nodes incident to exactly one impure min heap edge (def. 3.7(2)) is

$$((\epsilon + 1) \bmod 2) + (((N \bmod 3) + 1) \bmod 2)$$

where we let $\epsilon = \lfloor \frac{N}{2} \rfloor$.

- (10) The number of pure min heap leaves incident (in the min heap) to exactly one impure min heap edge is

$$\left\lfloor \frac{(N-1) \bmod 3}{2} \right\rfloor$$

If $N \neq 3$ then this is also the number of pure min heap leaves incident (in the min heap) to exactly one impure* min heap edge as well as the number of pure* min heap leaves incident (in the min heap) to exactly one impure* min heap edge. If $N = 3$ then these latter two quantities are both 0.

9.3. In Nodes. For the following lemma, recall the definitions 3.2 and 3.6.

Lemma 9.1. Let $\Lambda(\text{MinH})$ (resp. $\Lambda(\text{MaxH})$) denote the leaves of the min heap (resp. max heap). If $N > 1$ then let w (resp. x) denote the last min (resp. max) heap interior node (def. 3.2). Then,

- (1) A node with min heap coordinate i is an In node (def. 3.6) if and only if $\frac{N-(N \bmod 3)}{3} \leq i < \frac{2N+(N \bmod 3)}{3}$.
- (2) $\text{In}(B_N) = \Lambda(\text{MinH}) \cup \Lambda(\text{MaxH})$.
- (3) If $N \bmod 3 \neq 2$ then $\text{In}(B_N) = \Lambda(\text{MinH}) \cap \Lambda(\text{MaxH})$.
- (4) If $N \bmod 3 = 2$ then

$$\begin{aligned} \text{In}(B_N) &= \Lambda(\text{MinH}) \cap \Lambda(\text{MaxH}) \cup \{w, x\} \\ &= \Lambda(\text{MinH}) \cup \{w\} \\ &= \Lambda(\text{MaxH}) \cup \{x\} \end{aligned}$$

- (5) The last In node (def. 3.6) has min heap coordinate $\text{HeapSize}(N) - 1$ and the first In node (def. 3.6) has min heap coordinate

$$\begin{aligned} \text{FirstIn}(N) &= N - \text{HeapSize}(N) \\ &= \left\lfloor \frac{N}{3} \right\rfloor \\ &= \frac{N - (N \bmod 3)}{3} \end{aligned}$$

Note in particular that $\text{In}(N) - \text{FirstIn}(N) = N \bmod 3$.

- (6) The number of In nodes in B_N is

$$\begin{aligned} \text{In}(N) &= N - 2 \left\lfloor \frac{N}{3} \right\rfloor \\ &= \left\lfloor \frac{N+2}{3} \right\rfloor + \left\lfloor \frac{N \bmod 3}{2} \right\rfloor \\ &= \left\lfloor \frac{N+2}{3} \right\rfloor - \left\lfloor \frac{(N+2) \bmod 3}{2} \right\rfloor \\ &= 2 \left\lfloor \frac{2N}{3} \right\rfloor - N \\ &= \frac{N+2(N \bmod 3)}{3} \end{aligned}$$

so that in particular,

- $\text{In}(N) = \left\lceil \frac{N+2}{3} \right\rceil - 1$ or $\left\lceil \frac{N+2}{3} \right\rceil$, and $\text{In}(N) = \left\lfloor \frac{N+2}{3} \right\rfloor$ or $\left\lfloor \frac{N+2}{3} \right\rfloor + 1$,
- $3 \text{In}(N) - 4 \leq 3 \text{In}(N) - 2(N \bmod 3) = N \leq 3 \text{In}(N)$, and
- the number of Out nodes is $\text{Out}(N) = \frac{2}{3} [N - (N \bmod 3)] = 2 \left\lfloor \frac{N}{3} \right\rfloor$.

Proof. Let $h = \text{HeapSize}(N)$ and $\chi = N \bmod 3$. Note that for any node with min heap coordinate i , i is an In node if and only if $i < h$ and $\text{Flip}(i) < h$, from which (1) follows. As mentioned before, every leaf of the min heap belongs to the max heap and every leaf of the max heap belongs to min heap, thus giving us $\Lambda(\text{MinH}) \cup \Lambda(\text{MaxH}) \subseteq \text{In}(B_N)$.

Let i be an In and such that i is not a leaf of the min heap. Since i is an interior node in the min heap, we must have $i < \text{FirstLeaf}(N) = \frac{N - (N \bmod 3)}{3} + \left\lfloor \frac{(N \bmod 3)}{2} \right\rfloor$ but since i belongs to the max heap we also have $\frac{N - (N \bmod 3)}{3} \leq i$, which is only possible if $\left\lfloor \frac{(N \bmod 3)}{2} \right\rfloor = 1$ and $i = \frac{N - (N \bmod 3)}{3}$. This implies that $N \bmod 3 = 2$ and $i = \frac{N-2}{3}$, where we now conclude from proposition 8.1 that i is a leaf of the max heap. Thus $\text{In}(B_N) \subseteq \Lambda(\text{MinH}) \cup \Lambda(\text{MaxH})$, which proves (2).

We have also shown above that an In node i is an interior node of the min heap if and only if $N \bmod 3 = 2$ and $i = \frac{N-2}{3}$. By symmetry of the BiHeap construction, an In node i is an interior node of the max heap if and only if $N \bmod 3 = 2$ and $\text{Flip}(i) = \frac{N-2}{3}$, where the latter equality holds if and only if $i = \frac{2N-1}{3}$. Parts (3) and (4) now follows immediately from (2) and these observations. That the last In node has min heap coordinate $\text{HeapSize}(N) - 1$ is immediate and the formulas for $\text{FirstIn}(N)$ come from the observation that $\text{FirstIn}(N) = \text{Flip}(\text{HeapSize}(N) - 1)$ and formulas 9.1 for $\text{HeapSize}(N)$, which gives us part (5). It is now clear that $\text{In}(N) = N - 2 \text{FirstIn}(N) = N - 2(N - \text{HeapSize}(N))$ where most of part (6) now follows from this, part (5), and formulas 9.1 for $\text{HeapSize}(N)$. By considering the three cases of $N \bmod 3 = 0, 1, 2$, one can prove that $\text{In}(N) = \left\lfloor \frac{N+2}{3} \right\rfloor + \left\lfloor \frac{N \bmod 3}{2} \right\rfloor = \left\lceil \frac{N+2}{3} \right\rceil - \left\lfloor \frac{(N+2) \bmod 3}{2} \right\rfloor$, which is the rest of part (6). ■

We now make some important observations about In nodes:

- $\text{In}(B_2) = B_2$ and $\text{In}(B_3)$ consists solely of the B_3 's middle node.
- If N is odd then $\text{In}(B_N)$ contains the BiHeap's middle node.

- Out (N) is even.
- For any positive integers k and N , $k = |\text{In}(\text{B}_N)|$ if and only if $N = 3k, 3k - 2$, or $3k - 4$.
- Given $N_0 > 1$, the recurrence relation $N_{i+1} = 3N_i + 4$ has the solution $N_i = (N_0 - 2)3^i + 2$, where $(N_i)_{i=0}^\infty$ represents a sequence of BiHeap sizes such that $\text{In}(N_i) = N_{i-1}$ and $N_i \bmod 3 = 2$ for all $i > 0$. Such sequences are important to consider due to the additional complexity that the existence of a double headed arrow introduces into BiHeaps that have them.

Part 4. BiHeapification

Having obtained closed form formulas for $\text{HeapSize}(N)$, we can now define a function to take any collection of N elements and form a directed BiHeap (def. 2.5) with respect to any given total order on these elements.

10. Overview of BiHeapification. Recall that if we want to heapify N elements using an $O(N)$ algorithm, then we proceed inductively on $k = 1, \dots, N$ by first heapifying elements $N - k - 1, \dots, N - 1$ (which has the obvious meaning so that in a sense, these elements constitute “the current heap”) and then sifting down element $N - k$ by at most $\lceil \log_2(N + 1) \rceil - \lceil \log_2(k + 1) \rceil$ levels. Our $\text{BiHeapify}()$ algorithm will essentially start by doing the same thing, except that after it has sifted down to an In node (def. 3.6), which recall is an element that belongs to both the min heap and the max heap, it will begin to sift up the other heap.

For notational convenience, we describe the operation in the case where N is odd. The description in the case where N is even is almost identical, except for the notation of “the current BiHeap”, which will be defined shortly. If $\rho = \lfloor \frac{N}{2} \rfloor$, then the $\text{BiHeapify}()$ algorithm will proceed inductively as follows:

- (1) Suppose that for some $1 \leq k \leq \rho$ we have BiHeapified elements $\rho - (k - 1), \dots, \rho - 1, \rho, \rho + 1, \dots, \rho + (k - 1)$, which we will call the *current BiHeap*, where this of course means that those elements in the min heap satisfy the min heap condition while those in the max heap satisfy the max heap condition. The elements that belong to both the min (resp. max) heap and the current BiHeap will be referred to as the *current min (resp. max) heap*.
- (2) Next we take element $\rho - k$, which necessarily belongs to the pure* min heap, and sift it down the current min heap until either it has entered the current max heap or otherwise can go no further down the current min heap. If it enters into the max heap then we begin to sift up in the current max heap until we can go no further up. It is important to note here that all of this sifting takes place within the current BiHeap since this will end up guaranteeing that the $\text{BiHeapify}()$ operation is $O(N)$.
- (3) At this point, we will have BiHeapified elements $\rho - k, \rho - (k - 1), \dots, \rho, \dots, \rho + (k - 1)$.
- (4) We now repeat the process in (2) with element $\rho + k$ except that, since we are now starting in the pure* max heap, we reverse the roles of the current min heap and the current max heap. That is, we first sift down the current max heap and then sift up the current min heap.
- (5) At this point, we have BiHeapified elements $\rho - k, \dots, \rho, \dots, \rho + k$ so if $k = \rho$ then we’re done. Otherwise, we start (1) with k in place of $k - 1$ and continue.

Note that all of the above operations consist of nothing more than the usual operations to sift up/down a min/max heap so its implementation requires just slightly more work than what it takes to fully implement both a min heap and a max heap.

11. Bounded Sifting Up Heaps. Throughout, if a variable’s name is suffixed with “hc” (resp. “mc”), which stands for “(min) heap coordinate” (resp. “mirror coordinate”), then that variable stores the (min) heap coordinate (resp. mirror/max heap coordinate) of some node. Any variable suffixed with “nd” represents a node. For the reader who is unfamiliar with the notion of iterators, it is sufficient for our needs to consider “iterator” and “node” to be synonyms. The function $\text{SwapValues}(v, w)$ can then be considered to simply be a function that swaps the values $*v$ and $*w$. The variables beginning with “pos” store “the current node”’s position.

The following functions are nothing more than the usual sift up functions for min and max heaps, except that the operations are limited to some given range of nodes. That is, the sifting operation stops once a

node's min heap or max heap coordinate goes below a certain lower bound. This is what will guarantee that the BiHeapify () operation is $O(N)$.

```

1 //Assumes that pos_mc is a node in the max heap.
void SiftUpMaxHeap(Node V, int N, int pos_mc, int last_node_in_biheap_mc)
3   if (pos_mc == 0 or Parent(pos_mc) < last_node_in_biheap_mc)
       return
5   int parent_mc = Parent(pos_mc)
   Node pos_node = V + Flip(pos_mc)
7   do:
       parent_mc = Parent(pos_mc)
       Node parent_node = V + Flip(parent_mc)
       if (*pos_node <= *parent_node)
11        return
       SwapValues(pos_node, parent_node)
       pos_mc = parent_mc
       pos_node = parent_node
15  while (pos_mc > 0 and Parent(pos_mc) >= last_node_in_biheap_mc)
       return
17
18 //Assumes that pos_hc is a node in the min heap.
void SiftUpMinHeap(Node V, int N, int pos_hc, int first_node_in_biheap_hc)
19  if (pos_hc == 0 or Parent(pos_hc) < first_node_in_biheap_hc)
       return
21  int parent_hc = Parent(pos_hc)
   Node pos_node = V + pos_hc
23  do:
       parent_hc = Parent(pos_hc)
       Node parent_node = V + parent_hc
25  if (*pos_node >= *parent_node)
       return
       SwapValues(pos_node, parent_node)
       pos_hc = parent_hc
       pos_node = parent_node
29  while (pos_hc > 0 and Parent(pos_hc) >= first_node_in_biheap_hc)
       return
31
32
33

```

12. The BiHeapify Algorithm. To implement BiHeapify (), we must first implement two other functions, the first (resp. second) of which takes an element in the min (resp. max) heap and sifts it down until it reaches the max (resp. min) heap, at which point it proceeds to sift the element up for as long as it remains within bounds (as was discussed in the overview (2) above).

```

1 //Assumes that the node pos_hc belongs to the min heap and that
  // pos_hc <= last_node_in_biheap_hc.
void SiftFromMinToMax(Node V, int N, int pos_hc, int last_node_in_biheap_hc)
3   int heap_size = HeapSize(N)
   int first_node_in_mirror_heap = N - heap_size
   //Sift down the min heap while not yet in the max heap.
7   while (pos_hc < first_node_in_mirror_heap)
       int left_child_hc = LeftChild(pos_hc)
       int right_child_hc = RightChild(pos_hc)
       Node left_child_node = V + left_child_hc
       Node right_child_node = V + right_child_hc
       Node pos_node = V + pos_hc
11  bool is_right_child_valid = right_child_hc <= last_node_in_biheap_hc and
       right_child_hc < heap_size
       Node smaller_node
15  if (is_right_child_valid and *right_child_node < *left_child_node)
       smaller_node = right_child_node
17

```

```

    pos_hc      = right_child_hc
19  else
    smaller_node = left_child_node
21  pos_hc      = left_child_hc
    if (*pos_node <= *smaller_node)
23  return
    SwapValues(pos_node, smaller_node)
25  SiftUpMaxHeap(V, N, Flip(pos_hc), Flip(last_node_in_biheap_hc))
    return

//Assumes that the node pos_mc belongs to the max heap and that
2 // Flip(pos_mc) >= first_node_in_biheap_hc.
void SiftFromMaxToMin(Node V, int N, int pos_mc, int first_node_in_biheap_hc)
4  int heap_size      = HeapSize(N)
    int first_node_in_mirror_heap = N - heap_size
6  int pos_hc        = Flip(pos_mc)
    //Sift down the max heap while not yet in the min heap.
8  while (pos_mc < first_node_in_mirror_heap)
    int left_child_mc  = LeftChild(pos_mc)
10   int right_child_mc = RightChild(pos_mc)
    int left_child_hc  = Flip(left_child_mc)
12   int right_child_hc = Flip(right_child_mc)
    Node pos_node      = V + pos_hc
14   Node left_child_node = V + left_child_hc
    Node right_child_node = V + right_child_hc
16   //Note that right_child_hc >= first_node_in_biheap_hc necessarily holds.
    bool is_right_child_valid = right_child_mc < heap_size
18   Node larger_node
    if (is_right_child_valid and *right_child_node > *left_child_node)
20   larger_node = right_child_node
    pos_hc      = right_child_hc
22   pos_mc      = right_child_mc
    else
24   larger_node = left_child_node
    pos_hc      = left_child_hc
26   pos_mc      = left_child_mc
    if (*pos_node >= *larger_node)
28   return
    SwapValues(pos_node, larger_node)
30  SiftUpMinHeap(V, N, pos_hc, first_node_in_biheap_hc)
    return

1 int MinHeapCoordinateOfLastMinHeapNode(int size_of_heap)
    return size_of_heap - 1

void BiHeapify(Node V, int N)
2  if (N < 2)
    return
4  int heap_size = HeapSize(N)
    //Ignore all In nodes,...
6  int last_node_in_biheap_hc = MinHeapCoordinateOfLastMinHeapNode(heap_size)
    if (N % 3 == 2) //... unless N mod 3 == 2, in which case
8  last_node_in_biheap_hc -- //don't ignore the double headed arrow's nodes.
    int first_node_in_biheap_hc = Flip(last_node_in_biheap_hc)
10 //At any point during execution, the "current BiHeap" consists of the nodes with min
    // heap coordinates in the range [first_node_in_biheap_hc, last_node_in_biheap_hc].
12 //Invariant: All such nodes, except possibly one of the endpoints, will be guaranteed
    // to satisfy the biheap condition.
14 while (first_node_in_biheap_hc > 0)

```

```

    first_node_in_biheap_hc -- //Sift the next pure min heap node.
16  SiftFromMinToMax(V, N, first_node_in_biheap_hc, last_node_in_biheap_hc)
    last_node_in_biheap_hc++ //Sift the next pure max heap node.
18  SiftFromMaxToMin(V, N, Flip(last_node_in_biheap_hc), first_node_in_biheap_hc)
    return

```

Now, performing (2) requires moving element $\rho - k$ no more than $2d + 1$ levels, where $d = \lceil \log_2(\rho + 1) \rceil - \lceil \log_2(k + 2) \rceil$, and the same is true of performing (4). From here, recalling that $\sum_{i=0}^{\lceil \log_2 N \rceil} i \left\lceil \frac{N}{2^{i+1}} \right\rceil \leq 2N$, it should be now obvious how the usual proof that the heapify operation is $O(N)$ generalizes immediately to proving that the BiHeapify() operation is $O(N)$.

For suppose that the pure* min heap is a full binary tree and that SiftFromMinToMax() is being applied to the pure* min heap node v , whose min heap coordinate is i . Let v 's height in the min heap be l . The worst case cost SiftFromMinToMax sifting v down until it reaches a min heap leaf is at most l after which it is sifted up the max heap, which costs at most l swaps due to the symmetry of the construction. Thus if $h = \text{HeapSize}(N) = \frac{2N + (N \bmod 3)}{3}$ is the size of the min heap, then since there are $\left\lceil \frac{h}{2^{l+1}} \right\rceil$ nodes of height l in the min heap (and since the same is true of the max heap), the total cost of all calls to SiftFromMinToMax() is bounded above by

$$h + 2 \sum_{l=0}^{\lceil \log_2 h \rceil} l \left\lceil \frac{h}{2^{l+1}} \right\rceil \leq 5h = 5 \left\lceil \frac{2N + (N \bmod 3)}{3} \right\rceil \leq \frac{10}{3}(N+1)$$

Similarly, the total cost of all calls to SiftFromMinToMax is bounded above by $5h$ so that the total cost of calling BiHeapify is bounded above by $10h \leq 10 \left\lceil \frac{2N}{3} \right\rceil \leq \frac{20N}{3} + 20$. However, notice that we skip applying these functions to all In nodes that are not incident to a double headed arrow, of which there are at least $\frac{N - (N \bmod 3)}{3}$ so that for $N \geq 60$, the total cost is in fact at most $5N$.

Empirical testing indicates that it is likely that the total number of swaps required to BiHeapify any collection of N nodes is at most $\frac{7}{3}N$ for all $N > 0$, where the ratio of the cost of calling BiHeapify to N appears to approach $\frac{7}{3}$ as N increases.¹

Observe that the BiHeapify algorithm is idempotent. That is, calling BiHeapify() on a list of nodes that have just been passed as input to BiHeapify() does not change the value of any of the nodes. Note that if one were to add between lines 4 and 5 of the BiHeapify algorithm the code

```

1  int first_in_node = N - heap_size
   int num_in_nodes = heap_size - first_in_node
3  if (N > 2)
    BiHeapify(V + first_in_node, num_in_nodes)

```

which recurses on the In nodes that were initially ignored, then BiHeapify would become an $O(N)$ recursive algorithm that would no longer be idempotent.

Part 5. Sifting an Element

If we replace the value of some element in a BiHeap with some other value, then it may cease to be a BiHeap. We can correct this with the following $O(\log N)$ algorithm. We describe this algorithm although it will not be used elsewhere.

```

void BiHeapSift(Node V, int N, int pos_hc)
2  if (N < 2)
    return
4  int heap_size = HeapSize(N)
   int pos_mc = Flip(pos_hc)
6  auto value = *(V + pos_hc)

```

¹ Indeed, consider the BiHeap graph on N nodes and assign the node with min heap coordinate i the value $N - i$ so that the values are strictly descending as a function of i . It is easy to see that the cost of calling BiHeapify on this list of values gives a close approximation (if not the exact value) of the maximum possible cost of BiHeapifying N nodes.

```

    bool is_node_in_min_heap = pos_hc < heap_size
8   bool is_node_in_max_heap = pos_mc < heap_size
    if (is_node_in_min_heap and (pos_hc == 0 or *(V + Parent(pos_hc)) <= value))
10    SiftFromMinToMax(V, N, pos_hc, N - 1)
    else if (is_node_in_max_heap and (pos_mc == 0 or *(V + Flip(Parent(pos_mc))) >= value))
12    SiftFromMaxToMin(V, N, pos_mc, 0)
    else if (is_node_in_min_heap and *(V + Parent(pos_hc)) > value)
14    SiftUpMinHeap(V, N, pos_hc, 0)
    else
16    SiftUpMaxHeap(V, N, pos_mc, 0)
    return

```

At the time of writing, it is not known whether or not there exists an $O(\log N)$ algorithm that allows one to insert a new element into an existing BiHeap so that the result is again a BiHeap (of size one larger than the original).

Part 6. Recursively BiHeapifying In Nodes with BiHeapifyInwards()

13. Minimum Number of Elements Below and Above an In Value.

Definition 13.1. Let T be a rooted tree. By an *ancestor* (resp. a *strict ancestor*) of a node v in T we mean any node, including v (resp. excluding v), that lies along the unique shortest path from v to the root. If v is a node in T then $\text{Ancestors}_T(v)$ denotes the set of all ancestors of v in T . If S is a subset of nodes in T then $\text{Ancestors}_T(S)$ denotes $\bigcup_{v \in S} \text{Ancestors}(v)$. If T is understood then we may omit its mention. ■

Lemma 13.2. Let B be a rooted binary tree on $N > 1$ nodes. Let Λ be a non-empty subset of B 's leaves and let $T = \text{Ancestors}(\Lambda)$. Let S denote the set of nodes in T that have exactly one child in T (i.e. those non-root nodes in T that have degree 2 in T and including the root if it has degree 1 in T). Then T is a binary sub-tree of B rooted at B 's root and if there is a node in B that is not in T then S is not empty.

Proof. It is clear that T is a binary tree rooted at B 's root. Suppose for the sake of contradiction that S is empty and observe that this implies that every node in T that is not in Λ has exactly two children in T . In particular, the root must have two children in T . Let v_0 be any node in B that is not in T and note that since the root is in T , there must be some node $v \in B \setminus T$ along the unique path from v_0 to the root such that v 's parent, call it p , belongs to T . Since p is not a leaf in B , p does not belong to Λ but since p belongs to $T = \text{Ancestors}(\Lambda)$, it must therefore have a child that does belong to T . Thus p is a node with exactly one child in T , which contradicts the assumption that S is empty. ■

Notation 13.3. If T is a tree then let $\Lambda(T)$ denote the leaves of T . ■

Proposition 13.4. Suppose that the $N > 0$ and that the values $*V, \dots, *(V + (N - 1))$ form a BiHeap, B . Let $I = \text{In}(B)$ and let pivot_value be a value such that there exist nodes $V + i$ and $V + j$ in I such that $*(V + i) \leq \text{pivot_value} \leq *(V + j)$. Let Σ (resp. Γ) denote the set of nodes in B whose values are less than or equal to (resp. greater than or equal to) pivot_value . Let C denote $\Sigma \cap I$ (resp. $\Gamma \cap I$).

If $C = I$ then $|\Sigma| \geq \text{HeapSize}(N) = \lceil \frac{2N}{3} \rceil = 2|\Sigma \cap I| - (N \bmod 3)$ (resp. $|\Gamma| \geq \lceil \frac{2N}{3} \rceil = 2|\Gamma \cap I| - (N \bmod 3)$).

If any of the following conditions hold

- (1) $N \bmod 3 = 0$,
- (2) $N \bmod 3 = 1$ and $C \neq I$,
- (3) $N \bmod 3 = 2$, $|C| < |I| - 1$, and C does not contain the last node of the min (resp. max) heap.

then $|\Sigma| \geq 2|\Sigma \cap I|$ and $\frac{|\Sigma|}{N} \geq \frac{2}{3} \frac{|\Sigma \cap I|}{|I|}$ (resp. $|\Gamma| \geq 2|\Gamma \cap I|$ and $\frac{|\Gamma|}{N} \geq \frac{2}{3} \frac{|\Gamma \cap I|}{|I|}$).

Proof. Note that if $N = 5$ and Σ does not contain the last node of the min heap then the conclusion can be immediately verified. So assume that $N \neq 5$. If $N = 2, 4$, or 7 and $\Sigma \cap I \neq I$ then the conclusion can be

directly verified. The conclusion can also be directly verified for $N = 3$ and $N = 6$ so assume that N is none of these. We may thus assume that $N > 7$.

Assume for now that $C = I$. Then all of the min heap's leaves (and thus also the min heap itself) belongs to Σ so that $|\Sigma| \geq \text{HeapSize}(N) = \lceil \frac{2N}{3} \rceil = \frac{N+2(N \bmod 3)}{3}$. Now, $2|\Sigma \cap I| - (N \bmod 3) = 2|I| - (N \bmod 3) = 2\left(\frac{N+2(N \bmod 3)}{3}\right) - (N \bmod 3) = \text{HeapSize}(N)$. Note in particular that if $N \bmod 3 = 0$ then $|\Sigma| \geq \text{HeapSize}(N) = 2|\Sigma \cap I|$. With this observation, it now suffices to prove the conclusion under the assumption that $C \neq I$.

Note that $\Sigma \cap I$ is contained in the leaves of the min heap so let T denote the set $\Sigma \cap I$ together all all of $\Sigma \cap I$'s min heap ancestors. It follows from the definition of a min heap that T forms a binary tree rooted at V . Note that if $N \bmod 3 \neq 2$ then all of the remaining assumption imply that there exists some leaf in the min heap that does not belong to T . Lemma 13.2 now allows us to conclude that T necessarily contains some node with exactly one child. If, however, $N \bmod 3 = 2$ then the assumption that $|C| < |I| - 1$ and that C does not contain the last node of the min heap allow us, by simply ignoring the last node of the min heap, to apply this previous reasoning again to conclude that T necessarily contains some node with exactly one child.

Recall that in any binary tree, if l is the number of leaves in the tree and if d_1 (resp. d_2) is the number of nodes in the tree with exactly one child (resp. exactly two children), then $d_2 = l - 1$ and the number of nodes in the tree is $2l + d_1 - 1$. Thus $|T| \geq 2|\Lambda(T)| = 2|\Sigma \cap I|$. Since $T \subseteq \Sigma$, we have the desired conclusion.

Recalling that $3|\ln(B_N)| - 4 \leq 3|\ln(B_N)| - 2(N \bmod 3) = N \leq 3|\ln(B_N)|$, the inequality $\frac{|\Sigma|}{N} \geq \frac{2}{3} \frac{|\Sigma \cap I|}{|I|}$ now follows from $\frac{1}{N} \geq \frac{1}{|I|}$.

A similar argument proves the inequalities involving Γ . ■

14. The BiHeapifyInwards Algorithm. We want a function that finds a pivot value with a guaranteed minimum number of values less/greater than or equal to this pivot value. In light of proposition 13.4, the following $O(N)$ function is the most natural candidate. It finds a pivot value by applying BiHeapify(), restricting the list of nodes to the BiHeap's in nodes, and then repeating.

```

1 void BiHeapifyInwards(Node V, int N)
  if (N < 10)
3   sort(V, N) //Sort nodes V, V + 1, ..., V + (N - 1).
  return
5   BiHeapify(V, N) //This is an O(N) operation.
  int heap_size = HeapSize(N)
7   int first_in_node = N - heap_size // = Flip(heap_size - 1)
  int num_in_nodes = heap_size - first_in_node
9   //The inequality below necessarily holds, which implies that this is an O(N) algorithm.
  assert(num_in_nodes <= (N + 4) / 3)
11  BiHeapifyInwards(V + first_in_node, num_in_nodes)
  return

```

Since this is a tail recursive function, it can also be implemented as a loop in which case it will then use $O(\text{constant})$ additional memory. Since $|\ln(B_N)| = \frac{N+2(N \bmod 3)}{3}$ and $\sum_{i=0}^{\infty} \frac{1}{3^i} = \frac{3}{2}$, if $C > 0$ is such that for all sufficiently large N , BiHeapify performs no more than $C(N - \frac{4}{3})$ operations then for all sufficiently large N , BiHeapifyInwards performs no more than $\frac{3C}{2}(N - \frac{4}{3})$ operations. Furthermore, proceeding as in the proof of lemma 23.1, for $N > 2$, one may conclude from the inequalities $\frac{N}{3} \leq |\ln(B_N)| \leq \frac{N+4}{3}$ that if a call to BiHeapifyInwards recurses L times, then $\lceil \log_3(\frac{N}{10}) \rceil \leq L \leq \lfloor \log_3(\frac{N-2}{10-2}) \rfloor$.

Recalling that empirical testing shows that C is approximately $\frac{7}{3}$, it follows that empirical testing should indicate that the maximum number of swaps called by BiHeapifyInwards on N nodes is bounded above by approximately $\frac{7}{2}N$. Indeed this is what empirical testing shows with no test performing more than $\frac{7}{2}N$ swaps and with the ratio of the cost of calling BiHeapifyInwards to N appearing to approach $\frac{7}{2}$ as N increases.

15. The BiHeapifyInwardsNicerMath() Algorithm. Although the BiHeapifyInwards() function is, in relation to the consideration of In nodes, the simplest and most natural candidate for a function that finds a desirable pivot value, it has the unfortunate property that the conditions of proposition 13.4 that give desired inequalities are not necessarily satisfied. We now show how a slight modification rectifies this issue. First, we will need any $O(N)$ function that moves a maximal element to the end of the given list and moves a minimal element to the start of the list. The following function satisfies these requirements.

```
void EmplaceMinAndMax(Node V, int N)
2  BiHeapify(V, N)
   return
```

We also define in the appendix a simple $O(N)$ function, EmplaceAllMinsAndMaxs(Node V, int N), that places all minimum values at the start of the list and all maximum values at the end of the list. This function returns a pair of integers (a, b) where a is the largest integer such that $*(V+a)$ has the minimal value and b is the smallest non-negative integer such that $*(V+b)$ has the maximal value.

The function used to obtain the desired pivots is now defined as follows.

```
1  void BiHeapifyInwardsNicerMath(Node V, int N)
   if (N < 30)
3   sort(V, N) //Sort nodes V, V + 1, ..., V + (N - 1)
   if (N < 10)
5   return
   (a, b) = EmplaceAllMinsAndMaxs(V, N)
7   if (a >= N / 2 or b <= (N - 1) / 2) //If we've emplaced the median(s).
   return
9   if (N % 2 == 0 and (a == (N - 1) / 2 or b == N / 2)) //If we've found at least one median,
   EmplaceMinAndMax(V + (a + 1), b - (a + 1)) //then emplace the other median.
11  return
   int new_N = N
13  if (new_N % 3 == 2)
   V = V + 1
15  new_N = new_N - 2
   //At this point, new_N % 3 == 0 or else new_N % 3 == 1 and the final pivot value (whatever it
17  // may end up being) will either be equal to the median or else lie strictly in between two
   // other In nodes, thereby satisfying (2) of the above proposition.
19  BiHeapify(V, new_N)
   int heap_size = HeapSize(new_N)
21  int first_in_node = new_N - heap_size
   int num_in_nodes = heap_size - first_in_node
23  BiHeapifyInwardsNicerMath(V + first_in_node, num_in_nodes)
   return
```

15.1. The Number of Recursive Calls Made by BiHeapifyInwardsNicerMath().

Lemma 15.1. Let $\chi_N = \begin{cases} 0 & \text{if } N \bmod 3 = 0 \\ 2 & \text{if } N \bmod 3 = 1 \text{ and let } I(N) = \frac{N + \chi_N}{3}, \text{ so that } I(N) \text{ is the number of nodes} \\ -2 & \text{if } N \bmod 3 = 2 \end{cases}$

passed to a recursive call of BiHeapifyInwardsNicerMath(). For any non-negative integer i , let I^i denote the composition of I with itself i -times, where I^0 is the identity map.

There exists at most one non-negative integer i such that $I^i(N)$ is of the form $3^k + 2$ for some non-negative integer k , in which case $I^{i+l}(N) = 3^{k-l}$ for all $1 \leq l < k$. For $N > 8$, if N is not of the form $3^k + 2$ then $\lceil \log_3(N) \rceil = \lceil \log_3(I(N)) \rceil + 1$ and otherwise $\lceil \log_3(N) \rceil = \lceil \log_3(I(N)) \rceil + 2$.

Let ξ_N be 1 if there exists some non-negative integer i such that $I^i(N) = 3^k + 2$ for some integer $k > 1$ and let it be 0 otherwise. Let $\iota(N)$ be the number of times that BiHeapifyInwardsNicerMath() would call itself if it always recursed down until it was operating on less than 10 nodes. Then for $N =$ and $N > 5$,

$\iota(N) = \lceil \log_3(N) \rceil - 2 - \xi_N$. In particular, if $N > 3$ is even then $\text{BiHeapifyInwardsNicerMath}()$ would call itself at most $\lceil \log_3(N) \rceil - 2$ times.

Proof. If $N \bmod 3 = 0$ then $\lceil \log_3(N) \rceil = \lceil \log_3(I(N)) \rceil + 1$ is immediate so assume that $N \bmod 3 \neq 0$. Let k be a positive integer such that $3^k \leq N < 3^{k+1}$ and note that $\lceil \log_3(N) \rceil = k + 1$ (since $N \bmod 3 \neq 0$). If $N \bmod 3 = 1$ then $3^k < N \leq 3^{k+1}$ so that $\lceil \log_3(I(N)) \rceil = \lceil \log_3(N+2) \rceil - 1 = k$ from which we deduce $\lceil \log_3(N) \rceil = \lceil \log_3(I(N)) \rceil + 1$. Suppose now that $N \bmod 3 = 2$ so that $3^k \leq N-2 < 3^{k+1}$. If $3^k < N-2$, which happens if N is even (since $N \bmod 3 = 2$), then $\lceil \log_3(I(N)) \rceil = \lceil \log_3(N-2) \rceil - 1 = k$ so we again obtain $\lceil \log_3(N) \rceil = \lceil \log_3(I(N)) \rceil + 1$. If $N = 3^k + 2$ then $\lceil \log_3(I(N)) \rceil = k - 1$ so we obtain $\lceil \log_3(N) \rceil = \lceil \log_3(I(N)) \rceil + 2$.

Note that if $N = 3^k + 2$ for some $k > 1$ then $I(N) = 3^{k-1}$ so that $I^l(N) = 3^{k-l} \neq 3^{k-l} + 2$ for all $1 \leq l < k$. This means that for all N , there exists at most one non-negative integer i such that $I^i(N)$ is of the form $3^k + 2$ for some non-negative integer k .

Clearly, if $0 < N < 10$ then $\iota(N) = 0$, which is equal to $\lceil \log_3(N) \rceil - 2 - \xi_N$ if $N = 4$ or $N > 5$ so assume that $N \geq 10$. The claim about $\iota(N)$ should be clear from what has been proved so far after observing that $\iota(N)$ is just the smallest non-negative integer such that $I^{\iota(N)}(N) < 10$. ■

15.2. Pivots.

Definition 15.2. Suppose that we're given a list of N values $*V, \dots, *(V+(N-1))$ and that after calling some function $F()$ on this list, this list of values has been changed so as to become the list v_0, \dots, v_{N-1} .

- (1) If N is even then by the *F-left pivot value* (resp. *F-right pivot value*) of this list, denoted by pivot_value_L^F (resp. pivot_value_R^F), we mean the value $v_{\frac{N}{2}-1}$ (resp. $v_{\frac{N}{2}}$).
- (2) If N is odd then
 - (a) by the *F-pivot value* of this list, denoted by pivot_value^F , we mean the value $v_{\frac{N-1}{2}}$.
 - (b) by the *F-left pivot value* (resp. *the F-right pivot value*) of this list, denoted by pivot_value_L^F (resp. pivot_value_R^F), we mean the F-pivot value.

If the function F is understood from context then we may drop the superscript F from the notation for the pivot value. ■

15.3. Minimum Number of Elements Below and Above $\text{BiHeapifyInwardsNicerMath}()$ -Pivots. It now follows from lemma 15.1 and proposition 13.4 that as N increases exponentially so too does the number of values that are less (resp. greater) than or equal to the $\text{BiHeapifyInwardsNicerMath}()$ -left (resp. right) pivot value, for *any* set of N inputs. We state this formally.

Theorem 15.3. Let χ_N, I , and ξ_N be defined as in lemma 15.1. Suppose that the values $*V, \dots, *(V+(N-1))$, denoted by B , are passed to $\text{BiHeapifyInwardsNicerMath}()$. Let pivot_value_L (resp. pivot_value_R) be the $\text{BiHeapifyInwardsNicerMath}()$ -left (resp. right) pivot value (def. 15.2) and let Σ (resp. Γ) denote the set of values in B that are less than or equal to pivot_value_L (resp. greater than or equal to pivot_value_R).

Then for all $N \geq 16$, $|\Sigma| \geq 2^{\lceil \log_3 N \rceil + 1 - \xi_N}$ and $|\Gamma| \geq 2^{\lceil \log_3 N \rceil + 1 - \xi_N}$. In particular, for all $N \geq 16$, $N \geq 16$, $|\Sigma| \geq 2^{\lceil \log_3 N \rceil}$ and $|\Gamma| \geq 2^{\lceil \log_3 N \rceil}$ where observe that $2^{\lceil \log_3 N \rceil} \geq N^{\frac{1}{\log_2 3}}$ and $\frac{1}{\log_2 3} \approx 0.6309$.

Proof. We prove only the inequality involving $|\Sigma|$ since the proof of the inequality involving $|\Gamma|$ is completely analogous.

Clearly the conclusion holds for all $16 \leq N < 30$ so assume that $N > 29$. Suppose first that $N \bmod 3 = 0$ or 1. Note that $\xi_N = \xi_{I(N)}$. By our inductive hypothesis and proposition 13.4 we have that

$$\begin{aligned} |\Sigma| &\geq 2|\Sigma \cap \text{In}(B_N)| \geq 2 \left(2^{\lceil \log_3 I(N) \rceil + 1 - \xi_{I(N)}} \right) \\ &= 2 \left(2^{\lceil \log_3 (N + \chi_N) \rceil - 1 + 1 - \xi_{I(N)}} \right) = 2^{\lceil \log_3 (N + \chi_N) \rceil + 1 - \xi_N} \text{ since } \xi_N = \xi_{I(N)} \\ &\geq 2^{\lceil \log_3 (N) \rceil + 1 - \xi_N} \text{ since } \chi_N = 0 \text{ or } 2 \end{aligned}$$

So we will now henceforth assume that $N \bmod 3 = 2$.

Note that we have

$$|\Sigma| \geq 1 + 2|\Sigma \cap \text{In}(B_N)| \geq 1 + 2 \left(2^{\lceil \log_3 I(N) \rceil + 1 - \xi_{I(N)}} \right) = 1 + 2^{\lceil \log_3 (N-2) \rceil + 1 - \xi_{I(N)}}$$

Note that if N is not of the form $3^k + 2$ for some integer k then $\xi_N = \xi_{I(N)}$ and we can proceed as before to conclude that $|\Sigma| \geq 2^{\lceil \log_3 (N) \rceil + 1 - \xi_N}$. So assume that $N = 3^k + 2$ for some integer k , which gives us that $\xi_N = 1$. From lemma 15.1, we can conclude that $\xi_{I(N)} = 0$. Note that

$$\lceil \log_3 (N-2) \rceil = \lceil \log_3 (3^k) \rceil = k = \lceil \log_3 (N) \rceil - 1$$

so that

$$\lceil \log_3 (N-2) \rceil + 1 - \xi_{I(N)} = \lceil \log_3 (N) \rceil + 1 - 1 = \lceil \log_3 (N) \rceil + 1 - \xi_N$$

which gives us our desired conclusion. ■

Now although $2^{\lceil \log_3 N \rceil}$ is a lower bound for $|\Sigma|$ and $|\Gamma|$, testing on randomly generated data shows that on average, one should expect the minimum of $|\Sigma|$ and $|\Gamma|$ to be approximately $0.48N$. This means that in applications, this $O(N)$ function provides a pivot that is a very good approximation of the median.

For actual applications, the author recommends using `BiHeapifyInwards()` rather than `BiHeapifyInwardsNicerMath()` since it is less computationally intensive, produces a pivot value that is generally (based on empirical testing on data sets of sizes up to 2^{21} elements) just as good as that produced by `BiHeapifyInwardsNicerMath()` (meaning that on average, the minimum of $|\Sigma|$ and $|\Gamma|$ is also approximately $0.48N$), and, by virtue of the inequalities in proposition 13.4, is guaranteed to have a minimum of *approximately* $2^{\lceil \log_3 N \rceil}$ values above and below it. However, the exact expression for this lower bound is of course much more complicated than that given in theorem 15.3, which is why although `BiHeapifyInwards()` is recommended, we instead investigated its variant, `BiHeapifyInwardsNicerMath()`.

Part 7. Fused BiHeaps

Observe that if v is an In node in B_N and $N > 2$ then v has both a min heap parent and a max heap parent and furthermore, these two parents are the only nodes that v is adjacent to. This property makes In nodes the subject of special attention. The following definition will be used to implement a doubled ended heap that allows for amortized $O(N)$ insertions and pops.

Definition 15.4. Let w be a node in a graph G that is incident to exactly two edges and let $v \neq w$ and $z \neq w$ be the distinct other end points of these two edges. By *the fusion at w* we mean the edge between v and z and if we say that we *fuse the edges at w* then we mean that we remove from the graph the node w and its two incident edges and insert the edge between v and z ; we will call the resulting graph *the graph G fused at w* . If there was a parent-child relationship between v and w defined in G , with w being the parent of v , then with respect to the graph G fused at w we will say that z is the child of w and that w is the parent of z .

If an edge between two node v and z is the result of a fusion at an In node w , then to *un-fuse this edge (at w)* we mean to reverse the fusion of the edges at w ; that is, we insert the node w back into the graph and replace the edge between v and z with the edge between v and w and the edge between w and z . ■

Definition 15.5. Suppose that F is some set (possibly empty) of In nodes from the BiHeap B_N , which we'll call *the fused nodes*. Any In node that is not in F we'll say is *permitted*. By *the BiHeap graph (from N nodes) with F fused* or *the F -fused BiHeap graph (from B_N)*, denoted by $B_N^{\text{Fuse}(F)}$, we mean the graph defined by $B_N^{\text{Fuse}(\emptyset)} = B_N$ if $F = \emptyset$, $N = 1$, or $N = 2$, or otherwise if $F \neq \emptyset$ and $N > 2$ then it is the graph obtained as follows: denote the distinct elements of F by v_1, \dots, v_f and for all $1 \leq l \leq f$, inductively define $B_N^{\text{Fuse}(\{v_1, \dots, v_l\})}$ to be the graph $B_N^{\text{Fuse}(\{v_1, \dots, v_l\} \setminus \{v_l\})}$ fused at v_l .

If a graph is the F -fused BiHeap graph from B_N then we will call B_N a *parent (BiHeap)* of the F -fused BiHeap graph. ■

Remark 15.6. Observe that this definition is in fact independent of the ordering of F 's elements. That is, if α is a permutation of $\{1, \dots, f\}$ then repeating the above construction with the vertices ordered by $v_{\alpha(1)}, \dots, v_{\alpha(f)}$ will result in the same graph. Also observe that definition 15.4 transfers, at each step of the construction, any parent-child relationship that a node $v \in F$ may have to a new parent-child relationship between the nodes $\text{Parent}_{\text{Min}}(v)$ and $\text{Parent}_{\text{Max}}(v)$.

Definition 15.7. If v is an In node in B_N and $N > 2$ then we will call the nodes $\text{Parent}_{\text{Min}}(v)$, v , and $\text{Parent}_{\text{Max}}(v)$ the *triple at v* and we will say that this triple is *ordered* or that it *satisfies the BiHeap triple condition* if $*\text{Parent}_{\text{Min}}(v) \leq *v \leq *\text{Parent}_{\text{Max}}(v)$.

Suppose that $N \bmod 3 = 2$ and let v (resp. w) be the last node in the max heap (resp. min heap), whose min (resp. max) heap coordinate is $\frac{N-2}{3}$ (i.e. these are the nodes incident to the double headed arrow, so $w = \text{Parent}_{\text{Max}}(v)$ and $v = \text{Parent}_{\text{Min}}(w)$). We will call the nodes $\text{Parent}_{\text{Min}}(v)$, v , w , and $\text{Parent}_{\text{Max}}(w)$ the the BiHeap's *quadruple* and we will say that this quadruple is *ordered* or that it *satisfies the BiHeap quadruple condition* if $*\text{Parent}_{\text{Min}}(v) \leq *v \leq *w \leq *\text{Parent}_{\text{Max}}(w)$.

The BiHeap condition requires that the triple at every In node v be ordered so that if this condition is not satisfied then the BiHeapify algorithm will necessarily have to swap v 's value with the value of one of its two parents. However, during the course of the BiHeapify() algorithm it possible that the value of v is changed multiple times so that if one has some important information about v 's value then after the BiHeapify() algorithm is complete, v 's value may potentially be neither its original value nor the original value of one of its parents. Furthermore, as the BiHeapify() algorithm swaps values, it is difficult to know exactly which node will hold v 's original value after BiHeapify() completes, which is potentially very useful information for forming inequalities, for instance.

To mitigate these issues, we begin by weakening the BiHeap condition. The idea behind the following definition is that it ignores any BiHeap condition associated with any In node by replacing the BiHeap triple condition (i.e. $*\text{Parent}_{\text{Min}}(v) \leq *v \leq *\text{Parent}_{\text{Max}}(v)$ in def. 15.7) with the condition that the value of the min heap parent of the In node is less than or equal to the value of the In's max heap parent (i.e. $*\text{Parent}_{\text{Min}}(v) \leq *\text{Parent}_{\text{Max}}(v)$).

Definition 15.8. Let $V, \dots, V + (N - 1)$ be a collection of $N > 0$ nodes. Say that these nodes satisfy the *Fully Fused BiHeap condition* if:

Any two distinct nodes v and x , neither of which is an In node, satisfy the following conditions:

- (1) (*The min (resp. max) heap condition*) If v and x are both min (resp. max) heap nodes with x being the min (resp. max) heap parent of v then the min (resp. max) heap condition holds (i.e. $*x \leq *v$ (resp. $*x \geq *v$)).
- (2) (*The weakened BiHeap triple condition*) If v is a min heap node, x is a max heap node, and if there exists an In node w such that $v = \text{Parent}_{\text{Min}}(w)$ and $x = \text{Parent}_{\text{Max}}(w)$ then $*v \leq *w$.
- (3) (*The weakened quadruple condition*) If $N \bmod 3 = 2$ then we also require that $*\text{Parent}_{\text{Min}}(\text{Parent}_{\text{Min}}(w)) \leq *\text{Parent}_{\text{Max}}(w)$ where w is the last node in the min heap (whose max heap coordinate is $\frac{N-2}{3}$). This is equivalent to requiring that $*\text{Parent}_{\text{Max}}(\text{Parent}_{\text{Max}}(z)) \geq *\text{Parent}_{\text{Min}}(z)$ where z is the last node in the max heap (whose min heap coordinate is $\frac{N-2}{3}$).

Now suppose that in addition we have some set $S = \{V + i_1, \dots, V + i_k\}$ of In nodes from B_N , which we'll call *permitted nodes*, and let F denote those nodes in In that are not permitted. Then $B_N^{\text{Fuse}(F)}$ is an *F-Fused BiHeap* (from B_N) and it is a *Fused BiHeap* (from B_N) *permitting* S if either $N = 1$ or otherwise if it satisfies the Fully Fused BiHeap condition and:

- (1) if $N = 2$ then either there exists a non-permitted node or else $B_N^{\text{Fuse}(F)} = B_2$ forms a BiHeap.
- (2) if $N > 2$ the following conditions hold for all permitted nodes $v = V + i_l$:
 - (a) If v is not the end of a double headed arrow or if it is the end of a double headed arrow and both of the double headed arrow's ends are permitted, then we require that v satisfy the BiHeap triple condition.
 - (b) If v is the end of a double headed arrow and exactly one end of the double headed arrow is permitted then we require that
 - (i) $*v \leq * \text{Parent}_{\text{Max}}(\text{Parent}_{\text{Max}}(v))$ if v belongs to the pure* min heap, or else
 - (ii) $*v \geq * \text{Parent}_{\text{Min}}(\text{Parent}_{\text{Min}}(v))$ if v belongs to the pure* max heap.

If $F = \text{In}(B_N)$ consists of all In nodes from a BiHeap graph B_N then we will call $B_N^{\text{Fuse}(F)}$ a *Fully Fused BiHeap* (from B_N) if it is an *F-Fused BiHeap* from B_N . ■

If $V, \dots, V + (N-1)$ form an Fused BiHeap then observe that a call to BiHeapify has the effect of sifting all In nodes up the min heap or up the max heap (whichever is needed, if any) where if $N \bmod 3 = 2$ then in addition it guarantees that the quadruple (def. 15.7) is ordered. This means that calling BiHeapify on an Fused BiHeap either *only* moves any given In node up the min heap or else *only* moves it up the max heap.

16. FusedBiHeapify with a Range of Permitted Nodes. We now give an algorithm that, given a range $F = \{F_0, \dots, F_{f-1}\}$ of f In nodes in B_N , makes $B_N^{\text{Fuse}(F)}$ into an Fused BiHeap permitting $\text{In}(B_N) \setminus F$. No node in F will have its value even considered, let alone changed. Note that the FusedBiHeapify algorithm is idempotent. That is, calling FusedBiHeapify() on a list of nodes that have been passed as input to FusedBiHeapify() does not change the value of any of the nodes.

This algorithm generalizes immediately to the case where F is an arbitrary set of In node. If the algorithm for checking whether or not a node belongs to F has $O(\text{constant})$ complexity then the resulting algorithm will clearly have $O(N)$ complexity. However, since $\text{In}(N) = \frac{N+2(N \bmod 3)}{3}$ is $O(N)$, if this criterion for inclusion in F is checked by a function that does not have $O(\text{constant})$ complexity then the resulting algorithm may fail to have $O(N)$ complexity.

The case where $N \bmod 3 = 2$ adds so much additional complexity, that we separate the algorithm into the two cases of $N \bmod 3 = 2$ and $N \bmod 3 \neq 2$. This algorithm is just a variant of the BiHeapify algorithm given above, modified so as to “skip over” any node in F that would have been processed by the BiHeapify algorithm. It is so similar to the BiHeapify algorithm that familiarity with the BiHeapify algorithm along with the comments that are included should be enough to make clear that this algorithm halts, is correct, and has $O(N)$ complexity.

If $F = \emptyset$ then this is indicated by calling this FusedBiHeapify() with any values of first_F_hc and last_F_hc such that $\text{last_F_hc} < \text{first_F_hc}$. To have first_F_hc (resp. last_F_hc) be set to represent the In node with the smallest (resp. largest) min heap coordinate, then call FusedBiHeapify() by passing 0 for first_F_hc (resp. N for last_F_hc).

The FusedBiHeapify() function will be the following dispatch function.

```

2 void FusedBiHeapify(Note V, int N, int fuse_first_hc, int fuse_last_hc)
   if (N % 3 != 2)
       FusedBiHeapifyNoDHA(V, N, fuse_first_hc, fuse_last_hc)
4   else
       FusedBiHeapifyWithDHA(V, N, fuse_first_hc, fuse_last_hc)
6   return

```

We separate it into two cases based on that value of $N \bmod 3$ for two reasons. The first is that if $N \bmod 3 = 2$ then having to deal with the double headed arrow introduces considerable additional complexity. Secondly, to implement a double ended priority queue using fused BiHeaps, it suffices to implement the FusedBiHeapify()

function only for the case where $N \bmod 3 \neq 2$ so we may avoid the additional complexity that exists when $N \bmod 3 = 2$. There is not much insight to be gained from the additional complexity of the $N \bmod 3 = 2$ so we relegate the definition of `FusedBiHeapifyWithDHA()` to the appendix.

We specialize the above function to the important case where we fuse all `In` nodes.

```
void FusedBiHeapify(Node V, int N)
2   FusedBiHeapify(V, N, 0, N)
   return
```

16.1. FusedBiHeapify With $N \bmod 3 \neq 2$. In this case, the BiHeap B_N has no Double Headed Arrow (DHA) so no `In` node has a parent that could possibly also be in F . Note that although this algorithm was intended to form an Fused BiHeap in the case where $N \bmod 3 \neq 2$, it also works in the case where $N \bmod 3 = 2$ and neither endpoint of the double headed arrow is included in F .

```
1 void FusedBiHeapifySiftFromMinToMaxNoDHA(Node V, int N, int pos_hc,
                                     int last_node_in_biheap_hc, int first_F_hc, int last_F_hc)
3   int heap_size      = HeapSize(N)
   int first_in_node_hc = N - heap_size
5   while (pos_hc < first_in_node_hc)
       int left_child_hc = LeftChild(pos_hc)
7       int right_child_hc = RightChild(pos_hc)
       bool is_right_valid = false
9       if (right_child_hc <= last_node_in_biheap_hc and right_child_hc < heap_size)
           is_right_valid = true
11      bool is_left_valid = false
       if (left_child_hc <= last_node_in_biheap_hc and left_child_hc < heap_size)
           is_left_valid = true
13      //If the left child is in F then "skip over it."
15      if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)
           left_child_hc = Flip(Parent(Flip(left_child_hc)))
17      //If the right child is in F then "skip over it."
       if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
           right_child_hc = Flip(Parent(Flip(right_child_hc)))
19      //If a child did have to be "skipped over", then is it still valid?
21      if (is_right_valid and right_child_hc <= last_node_in_biheap_hc) is_right_valid = true
       else is_right_valid = false
23      if (is_left_valid and left_child_hc <= last_node_in_biheap_hc) is_left_valid = true
       else is_left_valid = false
25      if (!is_left_valid and !is_right_valid)
           return
27      Node pos_node      = V + pos_hc
       Node left_child_node = V + left_child_hc
29      Node right_child_node = V + right_child_hc
       Node smaller_node
31      if (!is_left_valid or (is_right_valid and *right_child_node < *left_child_node))
           smaller_node = right_child_node
33      pos_hc      = right_child_hc
       else //Here, the left child is valid.
35      smaller_node = left_child_node
       pos_hc      = left_child_hc
37      if (*pos_node <= *smaller_node)
           return
39      SwapValues(pos_node, smaller_node)
       SiftUpMaxHeap(V, N, Flip(pos_hc), Flip(last_node_in_biheap_hc))
41  return

1 void FusedBiHeapifySiftFromMaxToMinNoDHA(Node V, int N, int pos_mc,
                                     int first_node_in_biheap_hc, int first_F_hc, int last_F_hc)
3   int heap_size      = HeapSize(N)
```

```

    int first_in_node_hc = N - heap_size
5   int pos_hc          = Flip(pos_mc)
    while (pos_mc < first_in_node)
7       int left_child_mc = LeftChild(pos_mc)
        int right_child_mc = RightChild(pos_mc)
9       int left_child_hc = Flip(left_child_mc)
        int right_child_hc = Flip(right_child_mc)
11      bool is_right_valid = false
        if (right_child_mc < heap_size and right_child_hc >= first_node_in_biheap_hc)
13          is_right_valid = true
            if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)
15                left_child_hc = Parent(left_child_hc)
                left_child_mc = Flip(left_child_hc)
17            if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
                right_child_hc = Parent(right_child_hc)
19                right_child_mc = Flip(right_child_hc)
            if (is_right_valid and right_child_hc >= first_node_in_biheap_hc) is_right_valid = true
21        else
                                                    is_right_valid = false
            bool is_left_valid = false
23            if (left_child_hc >= first_node_in_biheap_hc)
                is_left_valid = true
25            if (!is_left_valid and !is_right_valid)
                return
27        Node pos_node          = V + pos_hc
        Node left_child_node    = V + left_child_hc
29        Node right_child_node = V + right_child_hc
        Node larger_node
31        if (!is_left_valid or (is_right_valid and *right_child_node > *left_child_node))
            larger_node = right_child_node
33            pos_hc          = right_child_hc
            pos_mc          = right_child_mc
35        else //Here, the left child is valid.
            larger_node = left_child_node
37            pos_hc          = left_child_hc
            pos_mc          = left_child_mc
39        if (*pos_node >= *larger_node)
            return
41        SwapValues(pos_node, larger_node)
        SiftUpMinHeap(V, N, pos_hc, first_node_in_biheap_hc)
43    return

1 void FusedBiHeapifyNoDHA(Node V, int N, int first_F_hc, int last_F_hc)
    if (N < 2)
3        return
    int heap_size          = HeapSize(N)
    int first_in_node_hc = N - heap_size
    if (first_F_hc < first_in_node_hc) //If first_F_hc is not an In node.
7        first_F_hc = first_in_node_hc
    if (last_F_hc >= heap_size) //If last_F_hc is not an In node.
9        last_F_hc = heap_size - 1
    //The rest of this algorithm is the same as in the BiHeapify() algorithm.
11   int last_node_in_biheap_hc = heap_size - 1
    if (N % 3 == 2)
13        last_node_in_biheap_hc --
    int first_node_in_biheap_hc = Flip(last_node_in_biheap_hc)
15   while (first_node_in_biheap_hc > 0)
        first_node_in_biheap_hc --
17   FusedBiHeapifySiftFromMinToMaxNoDHA(V, N, first_node_in_biheap_hc,
                                         last_node_in_biheap_hc, first_F_hc, last_F_hc)

```

```

19     last_node_in_biheap_hc++
    FusedBiHeapifySiftFromMaxToMinNoDHA(V, N, Flip(last_node_in_biheap_hc),
21     first_node_in_biheap_hc, first_F_hc, last_F_hc)
    return

```

Remark 16.1. Observe that the call `FusedBiHeapifyNoDHA(V, N, N, 0)` reduces the algorithm down to be nothing more than the `BiHeapify(V, N)` algorithm since the inequalities that check whether or not `pos_hc` belongs to the (degenerate) interval $[first_F_hc, last_F_hc]$ will always evaluate to false. The same is true of the call `FusedBiHeapifyWithDHA(V, N, N, 0)` found in the appendix. Thus, the call `FusedBiHeapify(V, N, N, 0)` forms a BiHeap out of any collection of values. However, it does this less efficiently than the the call to `BiHeapify(V, N)` due to the extra comparisons involving `pos_hc`.

17. Sifting an Value in a Fused BiHeap. To implement a BiQueue using fused BiHeaps, we need to be able to sift an element up or down such a structure. As mentioned before, we need only implement such a sifting algorithm in the case where $N \bmod 3 \neq 2$. For the algorithm that sifts an element in the case where $N \bmod 3 = 2$, the reader is referred to the implementation present in the code in author's GitHub account. We now give this algorithm, which assumes that the min heap coordinate `pos_hc` of that node that is to be sifted does not satisfy $first_F_hc \leq pos_hc \leq last_F_hc$.

```

void FusedBiHeapSiftNoDHA(Node V, int N, int pos_hc, int first_F_hc, int last_F_hc)
2   if (N < 2)
    return
4   int heap_size = HeapSize(N)
   int pos_mc = Flip(pos_hc)
6   Node pos_it = V + pos_hc
   auto pos_value = *pos_it
8   bool is_node_in_min_heap = pos_hc < heap_size
   bool is_node_in_max_heap = pos_mc < heap_size
10  if (is_node_in_min_heap && is_node_in_max_heap) //If pos is an In node
    int minh_parent_of_pos_hc = Parent(pos_hc)
12    int maxh_parent_of_pos_mc = Parent(pos_mc)
    int maxh_parent_of_pos_hc = Flip(maxh_parent_of_pos_mc)
14    Node minh_parent_of_pos_it = V + minh_parent_of_pos_hc
    Node maxh_parent_of_pos_it = V + maxh_parent_of_pos_hc
16    if (pos_value > *maxh_parent_of_pos_it)
        SwapValues(pos_it, maxh_parent_of_pos_it)
18        SiftUpMaxHeap(V, N, N - 1 - maxh_parent_of_pos_hc, 0)
    else if (pos_value < *minh_parent_of_pos_it)
20        SwapValues(pos_it, minh_parent_of_pos_it)
        SiftUpMinHeap(V, N, minh_parent_of_pos_hc, 0)
22    return
    //At this point, pos is not an In node.
24    if (!is_node_in_max_heap) //Then it's in the pure min heap and not in the max heap.
        if (pos_hc == 0 or pos_value >= *(V + Parent(pos_hc)))
26            FusedBiHeapifySiftFromMinToMaxNoDHA(V, N, pos_hc, N - 1, F_first_hc, F_last_hc)
        else
28            SiftUpMinHeap(V, N, pos_hc, 0)
    else //Then it's in the pure max heap and not in the min heap.
30        if (pos_mc == 0 or *(V + Flip(Parent(pos_mc))) >= pos_value)
            FusedBiHeapifySiftFromMaxToMinNoDHA(V, N, pos_mc, 0, F_first_hc, F_last_hc)
32        else
            SiftUpMaxHeap(V, N, pos_mc, 0)
34    return

```

Part 8. Lambdas

18. A Short Review of Lambdas. For the reader unfamiliar with the concept of a *lambda*, for our purposes, it may be thought of as a function that is also an object that another function may return, call, or have passed as an argument into it. We will use lambdas in our implementation of a double ended Queue to relabel the nodes of BiHeap graphs and fused BiHeap graphs. Although the use of lambdas is not strictly necessary, their use will save us having to exert the substantial amount of effort that would have been spent repeatedly rewriting slightly altered versions of the algorithms given in this paper. Lambdas are basic elements of many languages including both Perl and C++11 or later. Familiarity with lambdas from either of these languages is sufficient for our needs.

We will define a lambda by filling in the following template.

```

1  let lambda_name = lambda(ParamterList...) -> ReturnType {
2      \\Definition...
3  }
```

For instance,

```

1  let trivial_lambda = lambda(int i) -> int {
2      return i
3  }
```

We also allow lambdas to capture the values of local variables. For instance, if we define

```

1  LambdaType F(int N)
2      let addition_lambda = lambda(int i) -> int {
3          return N + i
4      }
5      return addition_lambda
6
7  lambda_one = F(1)
   lambda_two = F(2)
```

Then the call `lambda_one(42)` returns 43 while `lambda_two(42)` returns 44.

19. Using Lambdas to Relabel Nodes. We use lambdas to obviate the need to completely rewrite algorithms whenever there is a need to relabel nodes in a BiHeap or a fused BiHeap. All of our lambdas will be of the form:

```

1  lambda(int N, int i) -> int {
2      \\Definition
3  }
```

where we allow our relabeling of the node with min heap coordinate `i` to depend on `N`.

We must rewrite all of our algorithms to allow for this. This can be done as follows:

- (1) For any function prototype $F(\text{ParamterList} \dots)$ that we've defined, we add an additional parameter, `LambdaType lambda`, to the end of the parameter list: $F(\text{ParamterList} \dots, \text{LambdaType lambda})$. For instance, `BiHeapify(Node V, int N)` becomes `BiHeapify(Node V, int N, LambdaType lambda)`.
- (2) For any function call $F(\text{Args} \dots)$, , we add an additional argument `lambda` to the end of the list of arguments. For instance, the call `BiHeapify(V, N)` becomes the call `BiHeapify(V, N, lambda)`.
- (3) We replace all strings of the form `V + string` (including those surrounded by parentheses) with `V + lambda(N, string)`.

All algorithms included in this paper were written in a way so that these alterations can all be done quickly using regular expressions (which is the reason why we sometimes write, for instance, `V + 0` rather than simply `V`).

Observe that if we define `trivial_lambda` by

```

1  let trivial_lambda = lambda(int N, int i) -> int {
    return i
3  }

```

then the call $\text{BiHeapify}(V, N, \text{trivial_lambda})$ reduces down to the call $\text{BiHeapify}(V, N)$ of the original BiHeapify algorithm given above. The same is true of all the other algorithms. For this reason, whenever a lambda argument is missing from an argument list in a call, then it should be assumed that trivial_lambda has been passed as that call's lambda argument.

As an example of the application of lambdas, note that the following call to $\text{BiHeapify}()$:

```

1  let flip_lambda = lambda(int N, int i) -> int {
    return (N - 1) - i
3  }
    BiHeapify(V, N, flip_lambda)

```

produces a “reversed BiHeap ” where the maximum is at node $V = V + 0 = V + \text{Flip}(N - 1)$ and the minimum is at node $V + (N - 1) = V + \text{Flip}(0)$. This is because the lambda causes the BiHeapify algorithm to treat every min heap coordinate as if it were the node's max heap coordinate.

Part 9. A Double Ended Queue Based on Fused BiHeaps and Out Nodes

Recall that any node that does not belong to both the min heap and the max heap is called an Out node and that the number of out nodes in B_N is necessarily even. Define the following increasing endomorphism on the positive even integers.

$$\text{ParentBiHeapSize}(N) = \begin{cases} \frac{3N}{2} + 1 & \text{if } \frac{N}{2} \text{ is odd} \\ \frac{3N}{2} & \text{otherwise} \end{cases}$$

The following lemma will allow us to give an implementation for a double ended queue that avoids having to deal with the case where $N \bmod 3 = 2$ unless $N = 2$, thereby considerably simplifying the algorithms needed to implement a BiQueue from what they would otherwise have been.

Lemma 19.1. Suppose that $N > 0$ is even. Then $p := \text{ParentBiHeapSize}(N)$ is even, $p \bmod 3 \neq 2$, and B_p is a BiHeap having exactly N Out nodes.

Proof. Let $\chi \in \{0, 1, 2\}$ be such that $p = \frac{3N}{2} + \chi$ so that $p \bmod 3 = \chi$ is either 0 or 1. The number of Out nodes that B_p has is exactly $p - |\text{In}(B_p)| = p - \frac{p+2(p \bmod 3)}{3} = \frac{3N}{2} + \chi - \frac{(\frac{3N}{2} + \chi) + 2\chi}{3} = N$. ■

We will begin by defining some helper functions, starting with a lambda function that will be used to transform the min heap coordinate of a node into the array index at which its value will be stored.

```

LambdaType get_index_lambda(int num_ele)
2  let index_lambda = lambda(int N_local, int pos_hc) -> int {
    if (pos_hc < num_ele / 2)
4      return 2 * pos_hc
    else
6      return 2 * (num_ele - 1 - pos_hc) + 1 // = 2 * Flip(pos_hc) + 1
    }
8  return index_lambda

```

Note that it stores the values of elements in the pure* min heap at even indices and the values of elements in the pure* max heap at odd indices. Furthermore, if v is an pure* min heap node with min heap coordinate i , then the value of v is stored at index $2i$ while the value of the node corresponding to v 's flip (i.e. the node with max heap coordinate i) is stored at right next to it at index $2i + 1$. In particular, the minimum value is stored at index 0 and the maximum value is stored at index 1.

We will insert and remove elements in such a way so that the following condition holds:

- (BiQueue Memory Condition) If our array stores the value of a node at index $i_0 \geq 0$, then for all integers $0 \leq i \leq i_0$, the value of a some node is stored at index i in the array.

In contrast, if we were to store the value of a node with min heap coordinate i at index i in the array (in the trivial way), then this condition would be satisfied if and only if our fused BiHeap was a BiHeap. In the case where our fused BiHeap is not a BiHeap, there would thus be a “hole” in the array inbetween where the values of the nodes of our fused BiHeap would be stored, with the values of the pure* min heap on one side of this “hole” and the values of the pure* max heap on the other side. Furthermore, due to the nature of how elements are inserted and popped from the BiQueue, this trivial map from nodes to array indices would require us to occasionally move the pure* max heap towards the start of the array if one were to repeatedly pop the min or the max sufficiently many times. These inefficiencies are avoided by storing the value of a node at the array index determined by the lambda returned by `get_index_lambda`.

We will henceforth assume that we have an array `V`, whose first value is stored at index 0 and whose i^{th} value is denoted by `V+i`, that is automatically resized whenever additional memory may be needed. We will also assume that we have the following four global variables:

```

int num_elements //The number of elements in our fused BiHeap and in our BiQueue.
2 int N           //The number of nodes in the parent BiHeap of our fused BiHeap.
int F_first_hc   //The min heap coordinate of the first fused node (if any).
4 int F_last_hc   //The min heap coordinate of the last fused node (if any).
```

A fused BiHeap will be the data structure that holds the values of our BiQueue. The value of the variable `num_elements` will thus always be equal to the number of elements in our BiQueue. Our fused BiHeap will be a BiHeap if and only if `num_elements` ≤ 2 or `first_F_hc` $>$ `last_F_hc`. If `num_elements` $>$ 2 and `first_F_hc` \leq `last_F_hc` then our fused BiHeap’s fused nodes will be any node whose min heap coordinate, i , satisfies `first_F_hc` $\leq i \leq$ `last_F_hc`.

Per the discussion above, we can now define the functions used to obtain the minimum and the maximum values. These functions assume that the BiQueue contains at least one element.

```

auto min()
2   return V[0]
```

```

auto max()
2   if (num_elements == 1)
      return V[0]
4   else
      return V[1]
```

We will also be using the following helper functions.

```

1 void call_fused_biheapify(int new_N)
   LambdaType index_lambda = get_index_lambda(new_N)
3   FusedBiHeapify(V, new_N, F_first_hc, F_last_hc, index_lambda)
   return
```

```

void call_biheapify(int new_N)
2   LambdaType index_lambda = get_index_lambda(new_N)
   FusedBiHeapify(V, new_N, new_N, 0, index_lambda)
4   return
```

```

void call_fused_biheapify_sift(int pos_hc)
2   LambdaType index_lambda = get_index_lambda(N)
   FusedBiHeapifySift(V, N, pos_hc, F_first_hc, F_last_hc, index_lambda)
4   return
```

19.1. Inserting an Element. Suppose that we wish to insert a sequence a_0, a_1, \dots of elements into our BiQueue. If our fused BiHeap is fully fused (i.e. all of the parent BiHeap’s In nodes are fused) then we insert these values as follows.

- (1) Begin by un-fusing the node `first_F_hc` (which necessarily belongs to the pure* min heap), placing the value a_0 at this node, incrementing the value `first_F_hc`, and then sifting this value in the fused BiHeap, which is an $O(N)$ operation.
- (2) For the next value, we un-fuse the node `last_F_hc` (which necessarily belongs to the pure* max heap), place the value a_1 at this node, decrement the value `last_F_hc`, and then sifting this value in the fused BiHeap, which is again an $O(N)$ operation.
- (3) For the value a_2 , repeat step (1) and for the value a_3 repeat step (2), and so forth where if $i \bmod 2 = 0$ then we repeat step (1) for value a_i and we repeat step (2) for value a_{i+1} .

However, the above steps can only be repeated while there exists some node for us to un-fuse. This means that once our fused BiHeap is a BiHeap, there will be no node where we can place the next value. This happens after we have inserted $\text{In}(N) = \frac{N+2(N \bmod 3)}{3}$ values.

In this case we must enlarge our parent BiHeap so that our BiQueue's values are exactly the Out nodes of this new parent BiHeap. This is done by the following function, which is $O(N)$ since the new parent BiHeap size, given by `ParentBiHeapSize` above, is not more than $\frac{3N}{2} + 1$.

This function will only be called when $N \geq 2$ and `num_elements` is even. If one would need to allocate more memory for the array to insert a new element then this could be done within the following function, although due to how the values of the fused BiHeap are stored in the array, this expansion in the array's size could wait until `num_elements` equals the size of the array (even if N is strictly greater than the size of the array).²

```

void expand_parent_biheap()
2   N      = ParentBiHeapSize(N)
   F_first_hc = (num_elements + 1) / 2
4   F_last_hc  = (N - 1) - (num_elements / 2)
   call_fused_biheapify()
6   return
```

Once the parent BiHeap has been expanded, we then iterate the above process as many times as necessary until all values are inserted. Let us now compute the complexity of performing one such iteration.

Let $C > 0$ (resp. $D > 0$) be such that the number of operations performed by `call_fused_biheapify_sift` (resp. `expand_parent_biheap`) is not more than $C \log N$ (resp. DN). Then the sift operation will be called exactly $\text{In}(N) = \frac{N+2(N \bmod 3)}{3}$ times while the `expand_parent_biheap` function will be called only once, leading to a total cost of no more than

$$C(\log N) \frac{N+2(N \bmod 3)}{3} + D \left(\frac{3N}{2} + 1 \right)$$

where the ratio of this value to N is

$$\frac{C}{3} \log N + D \frac{3}{2} + \frac{D}{N} + \left(\frac{\log N}{N} \right) \frac{2(N \bmod 3)}{3} \leq \frac{C}{3} \log N + D \frac{3}{2} + \frac{D + 2 \log N}{N}.$$

Thus, the amortized cost of inserting an element is $O(\log N)$.

The function to insert a value into the BiQueue begins by handling the special cases where there are 0 or 1 elements in the BiQueue. We then check whether or not we need to expand our fused BiHeap's parent BiHeap in order to make room for the new value, in which case we call `expand_parent_biheap()`. We then determine whether we should un-fuse and place the value at node `first_F_hc` or `last_F_hc` and update the values of `first_F_hc` or `last_F_hc` appropriately, where this choice is determined by the BiQueue Memory Condition described previously. It completes by sifting the newly inserted value into its correct location within the fused BiHeap.

```

void insert(value)
2   if (num_elements < 2)
   F_first_hc = 1
4   F_last_hc  = 1
   if (num_elements == 0)
```

² This is why this function is called `expand_parent_biheap` and not `expand_array_size`.

```

6      V[0] = value
      else if (num_elements == 1)
8          if (value > V[0])
              V[1] = value
10         else
              V[1] = V[0]
12         V[0] = value
      num_elements = num_elements + 1
14      return
      if (num_elements == N) //If we need to make room for the new value.
16          expand_parent_biheap()
      int placement_node_hc
18      //Determine which node to un-fuse and place the value there.
      if (num_elements % 2 == 0) //Then "un-fuse" node F_first_hc to place the value there.
20          placement_node_hc = F_first_hc
          F_first_hc = F_first_hc + 1
22      else //un-fuse" node F_last_hc to place the value there.
          placement_node_hc = F_last_hc
24          F_last_hc = F_last_hc - 1
      V[num_elements] = value //Add it to the end of our array of values. We have now set
26          // the value of node placement_node_hc.
      num_elements = num_elements + 1
28      call_fused_biheapify_sift(placement_node_hc)
      return

```

19.2. Popping the Minimum or the Maximum. The following function accepts as input either 0, to pop the minimum element, or 1, to pop the maximum element. It assumes that the BiQueue contains at least one element. It begins by directly handling the exceptional cases where the BiQueue contains 1 or 2 elements. We then check whether or not there exists an In node that we may fuse (such a node is thus removed from the fused BiHeap). If not, then we call BiHeapify to make the nodes in the BiQueue into a BiHeap. We then determine whether we should fuse the node first_F_hc or the node last_F_hc and then update the values of first_F_hc or last_F_hc appropriately, where this choice is determined by the BiQueue Memory Condition described previously. Whichever of these two nodes is chosen, we swap its value with the value of the node that is to be popped and then sifting this value into its correct location within the fused BiHeap.

The argument showing that this function has amortized $O(N)$ complexity is so similar to the argument showing that insert has amortized $O(N)$ complexity that it is omitted.

```

1 void PopMinOrMax(int pop_index)
      if (num_elements <= 2)
3          N = 2
          F_last_hc = 1
5          if (num_elements == 2)
              if (pop_index == 0)
7                  SwapValues(V[0], V[1]) //Swap the min and max.
              num_elements = num_elements - 1
9              F_first_hc = num_elements
              return
11      int heap_size = HeapSize(N)
      int first_in_node = N - heap_size
13      if (F_first_hc == first_in_node) //If we can not remove any more In nodes.
          N = num_elements
          call_biheapify() //BiHeapify it.
          F_first_hc = (N + 1) / 2
17          F_last_hc = F_first_hc
      else
19          if (num_elements % 2 == 1) //If we should fuse F_first_hc.

```

```

    F_first_hc --
21  else
    F_last_hc++
23  num_elements--
    SizeType pop_node_hc = 0
25  if (pop_index == 1) //If we're to pop the max node
    pop_node_hc = N - 1 //then this node has min heap coordinate N - 1.
27  SwapValues(V[pop_index], V[num_elements])
    call_fused_biheapify_sift(pop_node_hc) //Sift the element into place.
29  return

```

The following functions assume that there is at least one element in the BiQueue.

```

1  void popmax()
    PopMinOrMax(1)
3  return

1  void popmin()
    PopMinOrMax(0)
3  return

```

Part 10. Appendix

20. Checking the BiHeap Condition. The following C++ like pseudocode checks if the nodes, the first of which is labeled V , form a BiHeap. This consists of nothing more than checking the min heap and the max heap conditions.

```

1  //Returns true if and only if the nodes  $V, \dots, V + (N - 1)$  form a BiHeap.
bool IsBiHeap(Node V, int N)
3  int heap_size = HeapSize(N)
    //Check that the nodes  $V, \dots, V + (\text{heap\_size} - 1)$  form
5  // a min heap with the minimum at first. This is half of the BiHeap condition.
    int parent_hc = 0 //This variable stores the parent's min Heap Coordinate.
7  while(RightChild(parent_hc) < heap_size) //While the right child is in the min heap.
    auto parent_value = *(V + parent_hc)
9    auto left_child_value = *(V + LeftChild(parent_hc))
    if (parent_value > left_child_value)
11     return false
    auto right_child_value = *(V + RightChild(parent_hc))
13    if (parent_value > right_child_value)
    return false
15    parent_hc++
    if (LeftChild(parent_hc) < heap_size) //If there is a parent with a single child.
17     auto parent_value = *(V + parent_hc);
    auto left_child_value = *(V + LeftChild(parent_hc))
19     if (parent_value > left_child_value)
    return false
21    //Check that the nodes  $V + \text{Flip}(0), \dots, V + \text{Flip}(\text{heap\_size} - 1)$ 
    // form a max heap with the maximum at  $V + (N - 1)$ .
23    int parent_mc = 0 //This variable stores the parent's Max heap Coordinate.
    while(RightChild(parent_mc) < heap_size) //While the right child is in the max heap.
25     auto parent_value = *(V + Flip(parent_hc))
    auto left_child_value = *(V + Flip(LeftChild(parent_mc)))
27     if (parent_value < left_child_value)
    return false
29     auto right_child_value = *(V + Flip(RightChild(parent_mc)))
    if (parent_value < right_child_value)
31     return false
    parent_mc++

```

```

33  if (LeftChild(parent_mc) < heap_size)
      auto parent_value = *(V + Flip(parent_mc))
35  auto left_child_value = *(V + Flip(LeftChild(parent_mc)))
      if (parent_value < left_child_value)
37      return false
  return true

```

21. Emplacing All Minima and Maxima. The following function places all minimum values at the start of the list and all maximum values at the end of the list. This function returns a pair of integers (a, b) such that a is the largest integer such that $*(V+a)$ has minimal value and b is the smallest non-negative integer such that $*(V+b)$ has maximal value.

```

pair<int, int> EmplaceAllMinsAndMaxs(Node V, int N)
2  EmplaceMinAndMax(V, N)
    int min_index = 0, max_index = N - 1
4  auto min_value = *(V + min_index)
    auto max_value = *(V + max_index)
6  if (min_value == max_value)
    return (max_index, 0)
8  if (N < 3)
    return (0, max_index)
10 int i = 1
    while (i < max_index)
12     auto current_value = *(V + i)
        if (current_value == min_value)
14         min_index++
            if (min_index < i)
16             SwapValues(V + min_index, V + i)
                else
18                 i++
                    else if (current_value == max_value)
20                     max_index--
                        SwapValues(V + max_index, V + i)
22                     else
                        i++
24 return (min_index, max_index)
}

```

REFERENCES

1. S. S. Skiena, *The algorithm design manual*, 2nd ed., Springer-Verlag, 2008.