BIHEAPS, A NEW TYPE OF DOUBLE ENDED HEAP WITH APPLICATIONS TO PIVOT SELECTION AND DOUBLE ENDED 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 is a max heap, and a part lies in both simultaneously. We give a simple to implement $O(\mathrm{N})$ 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 $O(\mathrm{N})$ insertions and amortized $O(\mathrm{N})$ pops. We define an $O(\mathrm{N})$ 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 $\mathrm{N} \geq 16$, at least $2^{\lceil \log_3 \mathrm{N} \rceil}$ elements have values ≤ pivot_value and at least $2^{\lceil \log_3 \mathrm{N} \rceil}$ elements have values ≥ pivot_value, where $2^{\lceil \log_3 \mathrm{N} \rceil} \geq \mathrm{N}^{1/\log_2 3}$ with $1/\log_2 3 \approx 0.6309$. Testing on randomly generated arrays shows that on average, approximately $0.48N$ elements have values ≤ pivot_value and approximately $0.48N$ elements have values ≥ pivot_value. Thus, this pivot represents a substantial improvement above random selection of a pivot for use in the QuickSort and QuickSelect algorithms.

Part 1. Introduction

It is assumed that the reader is familiar with graphs, trees, heaps, the standard $O(\mathrm{N})$ heapify algorithm and the proof that it is $O(\mathrm{N})$, the usual algorithms for sifting elements up and down heaps, and the QuickSort and QuickSelect algorithms. 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. We will henceforth assume that N is a positive integer and that we are given an ordered list of N nodes denoted by $\mathrm{V} = \mathrm{V}+0, \ldots, \mathrm{V}+(\mathrm{N}-1)$. If these nodes are assigned values then we will denote the value of node w by $^*w$. Throughout this paper, we will use C++ like pseudocode so that in particular, the result of performing non-negative integer division $a = b/c$; $(c \neq 0)$ assigns the value $\left\lfloor \frac{b}{c} \right\rfloor$ to a, where $\lfloor\cdot\rfloor$ (resp. $\lceil\cdot\rceil$) is the floor (resp. ceiling) function and the modulo of b by c is denoted by $b\%c$.

Define an involutory bijection Flip : $\{0, \ldots, \mathrm{N}-1\} \to \{0, \ldots, \mathrm{N}-1\}$ by

$$\mathrm{Flip}(i) = \mathrm{N}-1-i.$$

By *the i^{th} node* we mean the node $\mathrm{V}+i$ 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 $w = \mathrm{V}+i$ while by its *mirror coordinate* or its *max heap coordinate* we mean the integer $\mathrm{Flip}(i)$. We will say that a node *has coordinates* (i, m) or $i(m)$, if (1) i and m are integers in $\{0, \ldots, \mathrm{N}-1\}$, (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 $i = \mathrm{Flip}(m)$ and $m = \mathrm{Flip}(i)$. 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 $\mathrm{V} + (i + k)$. In the pseudocode, any variable that stores the heap coordinate (resp. mirror coordinate) of a node will 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. If N is even then we will call $\mathrm{V} + (\mathrm{N}/2 - 1)$ *the left middle* node and call $\mathrm{V} + (\mathrm{N}/2)$ *the right middle* node while if N is odd then we will call $\mathrm{V} + ((\mathrm{N}-1)/2)$ *the middle* node. If N is odd then we may also refer to the middle node as *the left middle* node or as *the right middle* node.

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

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

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

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 $0 < n \leq \mathrm{N}$, the sets $\mathbb{N}_{<n} := \{0, \ldots, n-1\}$ and $\mathbb{N}^{<n} := \operatorname{Flip}(\mathbb{N}_{<n}) = \{(\mathrm{N}-1) - (n-1), \ldots, \mathrm{N}-1\}$ denote the first (resp. last) n nodes of $\mathrm{V}, \ldots, \mathrm{V} + (\mathrm{N}-1)$. ■

Definition 2.2 (Min Heaps and Max Heaps)**.** Let $h \in \{1, \ldots, \mathrm{N}\}$. 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 $j = \operatorname{LeftChild}(i)$ or $j = \operatorname{RightChild}(i)$, 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* $\mathrm{N}-1$ *on* h *nodes* we mean the graph $\left(\mathbb{N}^{<h}, E\right)$ 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 $\operatorname{Flip}(j) = \operatorname{LeftChild}(\operatorname{Flip}(i))$ or $\operatorname{Flip}(j) = \operatorname{RightChild}(\operatorname{Flip}(i))$, where in the former (resp. latter) case we'll say that j is i's *(max heap) left* (resp. *right*) *child.* Note that if $m = \operatorname{Flip}(i)$ and $n = \operatorname{Flip}(j)$ are the max heap coordinates of these nodes then (i, j) is in E if and only if either $n = \operatorname{LeftChild}(m)$ or $n = \operatorname{RightChild}(m)$; 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* $\mathrm{N}-1$ *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* $\mathrm{N}-1$) *on* h *nodes* we mean the undirected graph induced by the directed complete binary tree rooted at 0 (resp. $\mathrm{N}-1$) on h nodes. ■

Definition 2.3 (BiHeaps)**.** For any positive integer $n \leq \mathrm{N}$, 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. $\mathrm{N}-1$) (def. 2.2). Let U_n denote the graph on $\{0, \ldots, \mathrm{N}-1\}$ 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. $\mathbb{N}^{<n}$). Due to the symmetry of this construction, note that R_n and R^n are necessarily isomorphic graphs (via $\operatorname{Flip}()$) and that R_n is a binary tree rooted at 0 if and only if R^n is a binary tree rooted at $\mathrm{N}-1$, in which case $R_n = C_n$, $R^n = C^n$, and they are isomorphic (via $\operatorname{Flip}()$) as rooted binary trees.

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

Letting $h = \operatorname{HeapSize}(\mathrm{N})$, by *the heap* or *the min heap* of B_{N}, denoted by MinH, we mean the graph $R_h = C_h$ and by *the mirror heap* or *the max heap* of B_{N}, denoted by MaxH, we mean the graph $R^h = C^h$. Letting $\rho = \lceil \mathrm{N}/2 \rceil$, we will call $R_\rho = C_\rho$ (resp. $R^\rho = C^\rho$) *the pure heap*, *the pure min heap*, or the PMinH (resp. *the pure mirror heap*, *the pure max heap*, or the PMaxH) of B_{N}.

If the nodes $0, \ldots, \mathrm{N}-1$ are assigned values and if $\prec$ is a partial order on these values then we will say that B_{N} *is a BiHeap with respect to* $\prec$ or a *directed BiHeap* if both of the following conditions hold:

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

where $h = \text{HeapSize}(\text{N})$. If the partial order $\prec$ 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 $\prec$ be a total order, we will henceforth assume this as well. ■

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 2.4 (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.

The analogous definitions for the pure min heap and the pure max heap should be clear. ■

Definition 2.5. An edge 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*) if it belongs to the pure min (resp. pure max) heap. An edge is *extended min* (resp. *extended max*) if it is an edge that belongs to the min (resp. max) heap but not to the pure min (resp. max) heap. An edge is *extended* (resp. *pure*) if it is an extended (resp. pure) min edge or an extended (resp. pure) max edge.

- Note that in the case where the BiHeap has an odd number, N, of nodes, the "last edge" of the pure min (resp. pure max) heap, which ends at the middle node (i.e. node $\frac{\text{N}-1}{2}$), is not an extended edge.

■

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 2.6. Let $\vec{\text{B}}_\text{N}$ denote B_N Define *the directed BiHeap graph on* N *nodes* 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 $v_1 \to v_2$ and $v_2 \to v_1$). 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.

2.1. Example Construction of a Directed BiHeap Graph. We now explain how one may construct the directed BiHeap graph on N = 14 nodes.

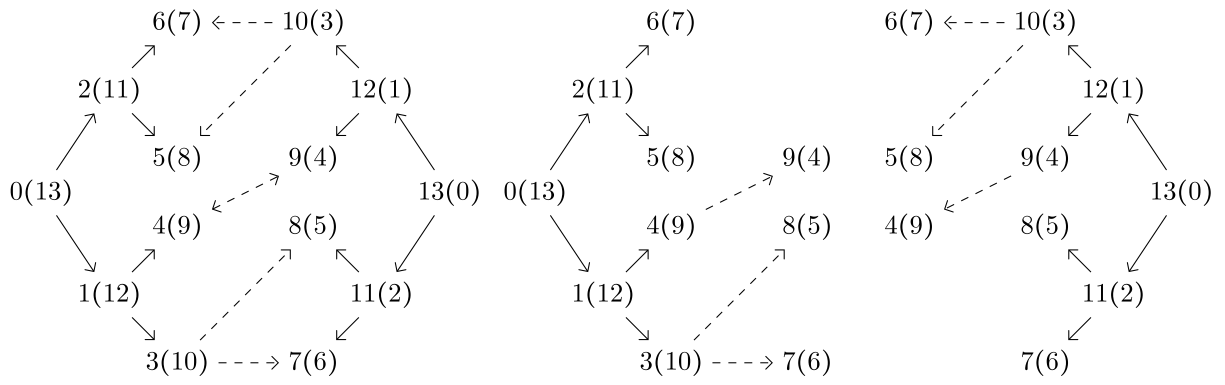

FIGURE 1. On the left is the directed (def. 2.6) BiHeap graph on 14 nodes, which has HeapSize (14) = 10. For the BiHeap graph, the double arrow should be interpreted as being a single edge in the graph. In the center is this BiHeap's directed (def. 2.2) min heap graph, whose restriction to the solid 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 this BiHeap graph 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 become comfortable with BiHeaps, the author recommends that the reader construct a few BiHeap graphs (for any 6 consecutive values of N) by hand by going through the construction in definition 2.3 and (symmetrically) adding two edges at a time. For instance first add edges $0(13) \to 1(12)$ and $13(0) \to 12(1)$ (i.e. from a min/max root to its left child) and then check the defining condition (which holds). Follow this by adding edges $0(13) \to 2(11)$ and $13(0) \to 11(2)$ (i.e. from a min/max root to its right child) and then checking the defining condition (which holds). Now add edges $1(12) \to 3(10)$ and $12(1) \to 10(3)$ 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 tells you that you should not have added the last two edges (i.e. you should only have done 6 more iterations rather than 7).

2.2. Illustrations. 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. 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 draw 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 includes 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 BiHeap graph then, ignoring all nodes' labels, one obtains the max heap (resp. pure max heap).

Henceforth, we will identify the node $\mathrm{V} + c$ with the node in the BiHeap graph on N nodes labeled (c, m) or $c(m)$ where $m = \mathrm{Flip}(c)$. In this way, the node labeled (c, m) or $c(m)$ will be said to have coordinate c and mirror coordinate $m = \mathrm{Flip}(c)$.

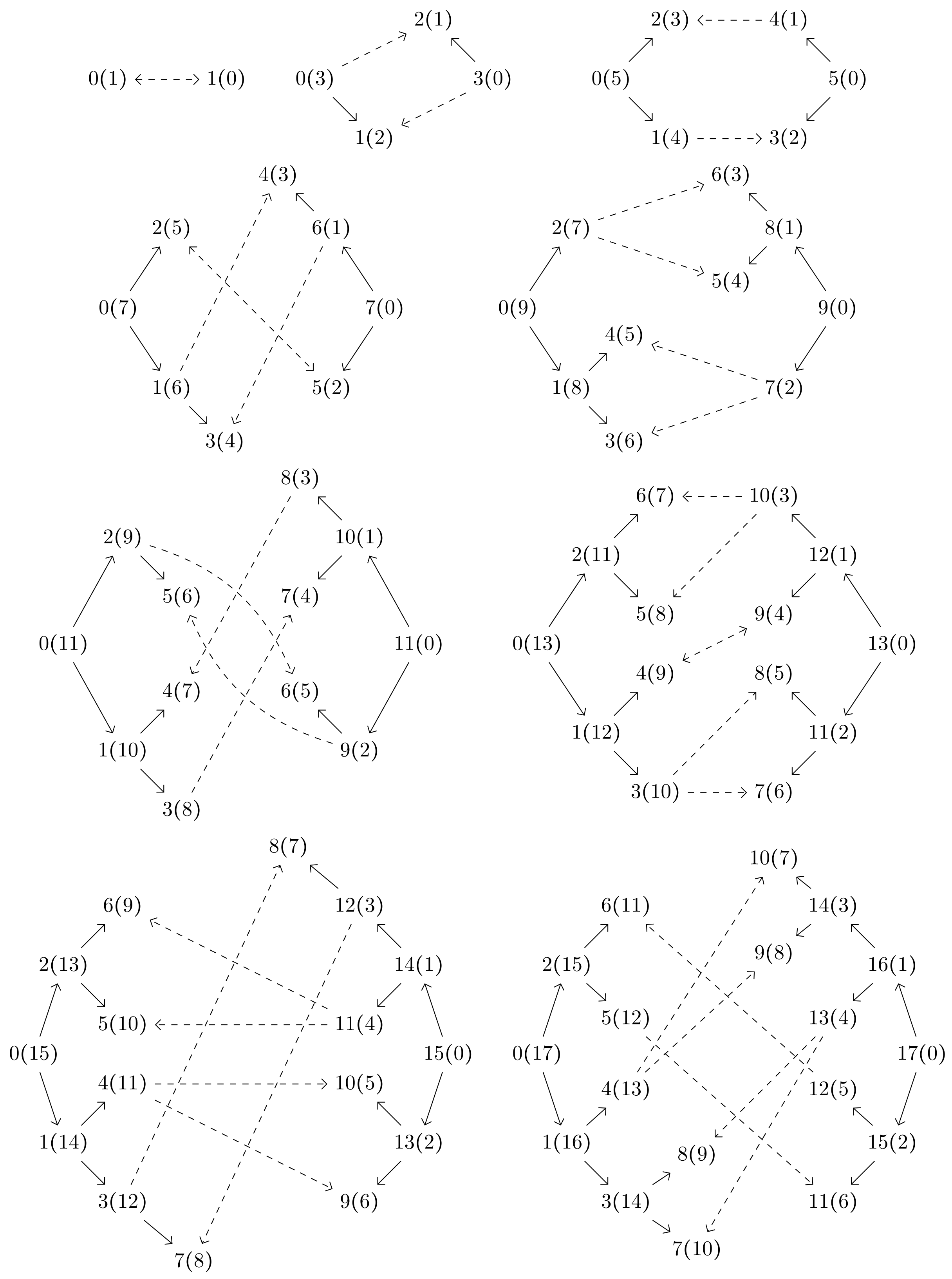

FIGURE 2. From left to right and from the top down, these are the BiHeap graphs on $2, 4, 6, 8, 10, 12, 14, 16$, and 18 nodes, respectively. The two connected components of the subgraph consisting of only the solid arrows and their nodes form the two "pure" subheaps of these BiHeap graphs.

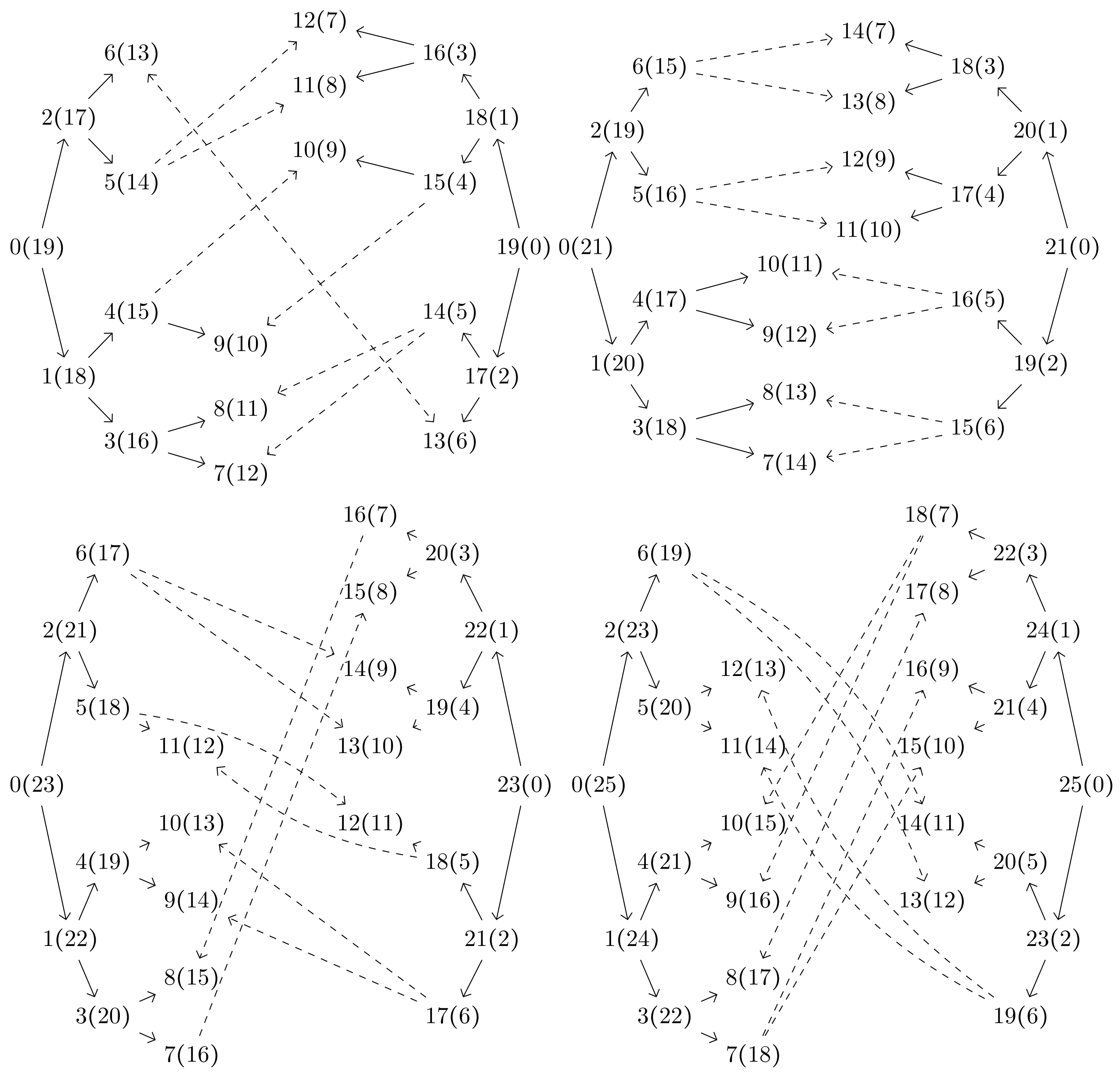

Figure 3. From left to right and from the top down, these are the BiHeap graphs on $20, 22, 24$, and 26 nodes, respectively.

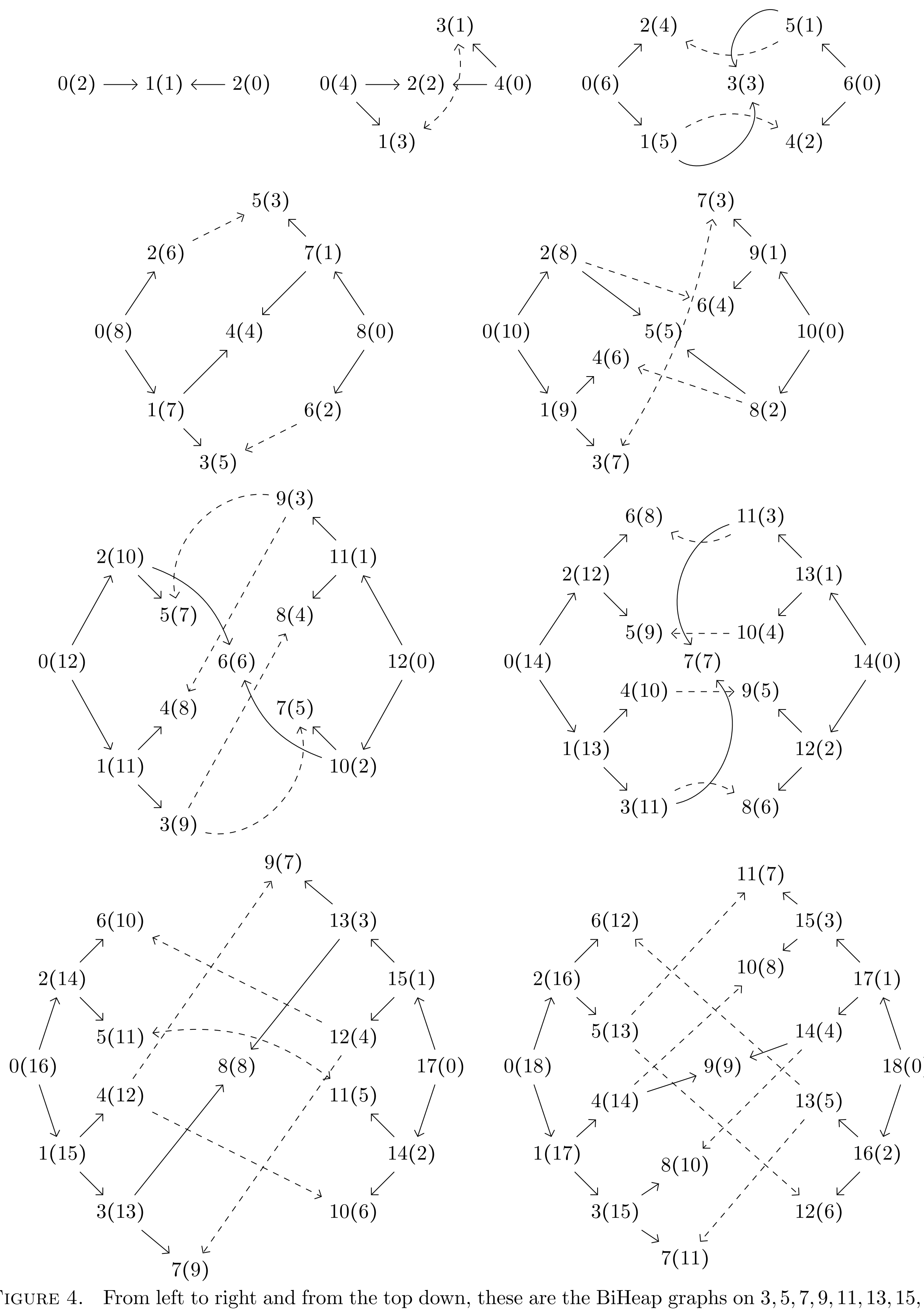

FIGURE 4. From left to right and from the top down, these are the BiHeap graphs on $3, 5, 7, 9, 11, 13, 15, 17,$ and 19 nodes, respectively. The subgraph consisting of the middle node and all nodes on the left (resp. right) half of the graph together with the solid arrows between these nodes form the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

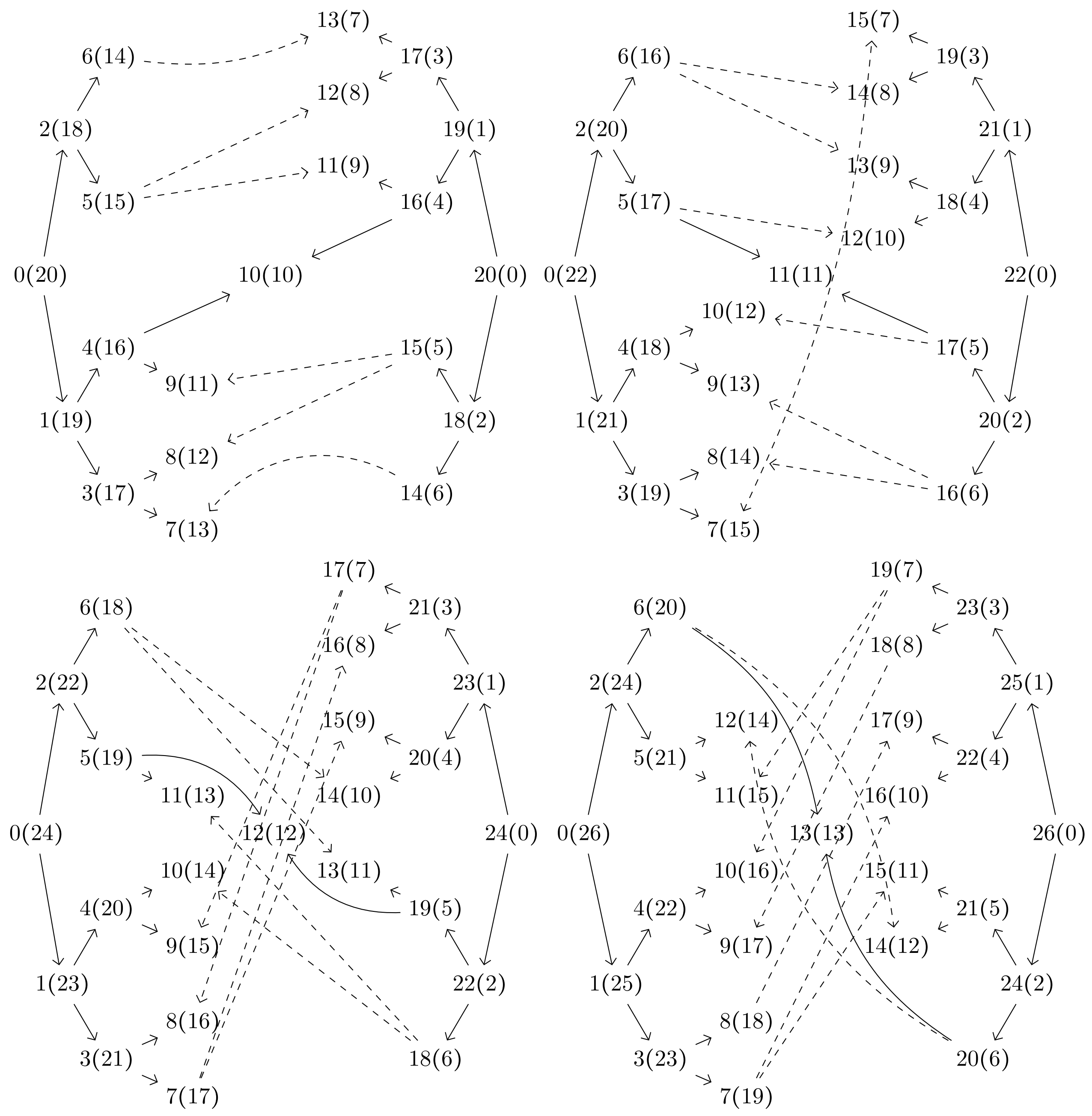

FIGURE 5. From left to right and from the top down, these are the BiHeap graphs on $21, 23, 25$, and 27 nodes, respectively.

2.3. 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. We have already defined the function HeapSize() but we will give the formula for computing it after we've classified all BiHeap graphs.

```
//Returns true if and only if the nodes V, ..., V + (N − 1) form a BiHeap.
bool IsBiHeap(Node V, int N)
  int heap_size = HeapSize(N)
```

```
//Check that the nodes V, ..., V + (heap_size - 1) form
// 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.
while(RightChild(parent_hc) < heap_size) //While the right child is in the min heap.
  auto parent_value     = *(V + parent_hc)
  auto left_child_value = *(V + LeftChild(parent_hc))
  if (parent_value > left_child_value)
    return false
  auto right_child_value = *(V + RightChild(parent_hc))
  if (parent_value > right_child_value)
    return false
  parent_hc++
if (LeftChild(parent_hc) < heap_size) //If there is a parent with a single child.
  auto parent_value     = *(V + parent_hc);
  auto left_child_value = *(V + LeftChild(parent_hc))
  if (parent_value > left_child_value)
    return false
//Check that the nodes V + Flip(0), ..., V + Flip(heap_size - 1)
// form a max heap with the maximum at V + (N - 1).
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.
  auto parent_value     = *(V + Flip(parent_hc))
  auto left_child_value = *(V + Flip(LeftChild(parent_mc)))
  if (parent_value < left_child_value)
    return false
  auto right_child_value = *(V + Flip(RightChild(parent_mc)))
  if (parent_value < right_child_value)
    return false
  parent_mc++
if (LeftChild(parent_mc) < heap_size)
  auto parent_value     = *(V + Flip(parent_mc))
  auto left_child_value = *(V + Flip(LeftChild(parent_mc)))
  if (parent_value < left_child_value)
    return false
return true
```

2.4. 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. Recall that with definition 2.4, we have disambiguated the notions of child and parent.

Definition 2.7.

(1) Say that a node is a *min* (resp. *max*) *leaf* if it is a leaf in the min (resp. max) heap.
(2) Say that a node is a *pure min* (resp. *pure max*) *leaf* if it is a leaf in the pure min (resp. pure max) heap.
(3) Say that a node is *extended min* (resp. *extended 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.
(4) Say that a node v is *leaf extended min* (resp. *leaf extended max*) if (a) it is an extended min (resp. extended max) node and (b) in the min (resp. max) heap, there exists an edge to v from a leaf of the pure min (resp. pure max) heap.
(5) Say that a node v is *interior extended min* (resp. *interior extended max*) if (a) it is an extended min (resp. extended max) node and (b) in the min (resp. max) heap, there exists an edge to v from an node that is in the interior of the pure min (resp. pure max) heap.

■

For edges, in addition to definition 2.5, we introduce the following terminology, where in reference to an edge, "extended" (resp. "pure") refers to the dashed (resp. solid) arrows in figures 2, 3, 4, and 5.

Definition 2.8.

(1) Say that an edge is *leaf extended min* (resp. *leaf extended max*) if it is an extended min (resp. extended) edge and it is incident to a leaf extended min (resp. leaf extended max) node. That is, an edge is leaf extended min (resp. leaf extended 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 extended* if it is a leaf extended min edge or a leaf extended max edge.
(2) Say that an edge is *interior extended min* (resp. *interior extended max*) if it is an extended min (resp. extended max) edge and it is incident to an interior extended min (resp. interior extended max) node. It is *interior extended* if it is an interior extended min edge or an interior extended max edge.
(3) Say that an edge between two nodes v and w is a *double arrow* if both v and w belong to both the min heap and the max heap and if they are incident to each other in both the min heap and max heap.

■

3. Properties of BiHeaps.

3.1. Basic Properties. 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 3.1. Let v and w be two nodes in $\mathrm{B_N}$ with coordinates (i, m) and (j, n), respectively (so that by definition, $m = \mathrm{Flip}(i)$ and $n = \mathrm{Flip}(j)$) 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) if and only if the node with coordinate (m, i) belongs to the max heap (resp. 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.

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

(4) There is at most one pure min heap interior node that has an extended min heap edge leaving it along with an extended max heap edge entering it (since otherwise the max heap would have a cycle). Hence, if the edge from v to w is such an edge then $i \geq \lfloor (\rho - 1)/2 \rfloor$, which is the coordinate of the last min heap interior node, where $\rho = \lceil \mathrm{N}/2 \rceil$.
(5) If v belongs to the min heap, $i \leq j$, and there exists an extended 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 $\rho = \lceil N/2 \rceil$ 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 $(f, \operatorname{Flip}(f))$.
   (ii) If $k > \operatorname{Flip}(f)$ then node $(k, \operatorname{Flip}(k))$ 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 $j \leq \operatorname{Flip}(f)$.
   (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 extended min heap edge incident to any give pure max heap node.

Lemma 3.2. Let v be the first leaf in the pure min heap (i.e. the pure min heap leaf with the smallest min heap coordinate) and let w be the first leaf in the pure max heap. Let (i, m) and (j, n) be the coordinates of v and w, respectively, and let $\rho = \lceil \mathrm{N}/2 \rceil$ be the size of the pure min heap. Then

$$i = n = \lfloor \rho/2 \rfloor = \frac{1}{4}\left(\mathrm{N} - 2 + ((3\mathrm{N}+2) \mod 4)\right)$$

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

and

$$\rho = \lceil \mathrm{N}/2 \rceil = \frac{\mathrm{N} + (\mathrm{N} \mod 2)}{2}.$$

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 $j = \operatorname{Flip}(\lfloor \rho/2 \rfloor)$. One can verify that the following equalities hold by going through each of the four cases:

$$\lfloor \lceil \mathrm{N}/2 \rceil / 2 \rfloor = \begin{cases} \frac{1}{4}(\mathrm{N}-2+3) & \text{if N is odd and } 4 \nmid \mathrm{N}-1 \iff (3\mathrm{N}+2) \mod 4 = 3 \iff 2 \nmid \mathrm{N} \text{ and } 2 \mid \rho \\ \frac{1}{4}(\mathrm{N}-2+2) & \text{if } 4 \mid \mathrm{N} \iff (3\mathrm{N}+2) \mod 4 = 2 \iff 2 \mid \mathrm{N} \text{ and } 2 \mid \rho \\ \frac{1}{4}(\mathrm{N}-2+1) & \text{if N is odd and } 4 \mid \mathrm{N}-1 \iff (3\mathrm{N}+2) \mod 4 = 1 \iff 2 \nmid \mathrm{N} \text{ and } 2 \nmid \rho \\ \frac{1}{4}(\mathrm{N}-2+0) & \text{if N is even and } 4 \nmid \mathrm{N} \iff (3\mathrm{N}+2) \mod 4 = 0 \iff 2 \mid \mathrm{N} \text{ and } 2 \nmid \rho \end{cases}$$

It then remains to go through each of those cases again and note that in each, the value is necessarily equal to $\frac{1}{4}(\mathrm{N}-2+((3\mathrm{N}+2) \mod 4))$. Since $j = \operatorname{Flip}(i)$, it follows that $j = \frac{1}{4}(3\mathrm{N}-2-((3\mathrm{N}+2) \mod 4))$. By the symmetry in the construction of the BiHeap graph, we have that $i = n$ and $j = m$. ∎

Corollary 3.3. 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 $\rho = \lceil \mathrm{N}/2 \rceil$ be the size of the pure heap. Then

$$i = n = \lfloor \rho/2 \rfloor - 1 = \frac{1}{4}\left(\mathrm{N} - 6 + ((3\mathrm{N}+2) \mod 4)\right)$$

and

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

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

4. 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 4.2. Another way to classify BiHeaps is to group them depending on whether or not they have interior extended edges (def. 2.7(5)), which is the content of proposition 4.6. 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.

4.1. Even or Odd Number Of Nodes.

Proposition 4.1. 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 4.2. 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 $\text{Parent}(\text{Flip}(\text{LeftChild}(i))) = \left\lfloor \frac{N-2i-3}{2} \right\rfloor$ while $\text{Parent}(\text{Flip}(\text{RightChild}(i))) = \left\lfloor \frac{N-2i-4}{2} \right\rfloor$ 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 4.3. 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 4.4. 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 4.2.

Corollary 4.5. N is even if and only if the BiHeap graph on N nodes is a single edge, a hexagon, or contains a rectangle.

4.2. BiHeaps with Interior Extended Edges.

Proposition 4.6. Let $\rho = \lceil N/2 \rceil$ be the size of the pure heap. Then the following are equivalent:

(1) ρ is even.
(2) 4 divides N or 4 divides N +1.
(3) There exists an interior extended edge (def. 2.8(2)).
(4) There exists exactly one interior extended min edge and/or there exists exactly one interior extended max edge.

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

$$\begin{aligned} i &= \frac{1}{4}\left(\mathrm{N}-6+((3\,\mathrm{N}+2) \mod 4)\right) \\ m &= \frac{1}{4}\left(3\,\mathrm{N}+2-((3\,\mathrm{N}+2) \mod 4)\right) \\ j &= \frac{1}{2}\left(\mathrm{N}-4+((3\,\mathrm{N}+2) \mod 4)\right) = \mathrm{RightChild}\left(\rho/2-1\right) \\ n &= \frac{1}{2}\left(\mathrm{N}+2-((3\,\mathrm{N}+2) \mod 4)\right) \end{aligned}$$

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 3.2. ■

4.3. BiHeaps whose Heaps Share an Edge.

Proposition 4.7. The following are equivalent:

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

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

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

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

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

$$\frac{1}{3}\left(\left\lceil \frac{\rho}{2} \right\rceil - 2 + (\rho \mod 2) - (\mathrm{N} \mod 2)\right)$$

where $\rho = \left\lceil \frac{\mathrm{N}}{2} \right\rceil$ is the size of the pure min heap.
(f) If $\mathrm{N} > 2$ then N is even (resp. odd) if and only if $w = \mathrm{RightChild}_{\mathrm{Min}}\left(\mathrm{Parent}_{\mathrm{Min}}(w)\right)$ (resp. $w = \mathrm{LeftChild}_{\mathrm{Min}}\left(\mathrm{Parent}_{\mathrm{Min}}(w)\right)$).

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

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 $2i + 2 = \text{RightChild}(i) = m = \text{Flip}(i) = \text{N} - 1 - i$ so that $\text{N} = 3(i + 1)$. But we also have that $j = m = \text{RightChild}(n) = \text{RightChild}(\text{Flip}(j))$, which gives us $j = 2\text{N} - 2j$. It follows that $3j = 2\text{N}$, $j = 2(i + 1)$ so that j is even and $j = \frac{2\text{N}}{3}$. 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 $\text{N} - 1 - i = \text{Flip}(i) = j = m = i = \text{LeftChild}(n) = \text{LeftChild}(i) = 2i + 1$. This shows that $\text{N} = 3i + 2$ and $i = \frac{\text{N}-2}{3}$. Using $m = \text{LeftChild}(n) = \text{LeftChild}\left(\frac{\text{N}-2}{3}\right)$ we obtain $m = \frac{2\text{N}-1}{3}$.

We've thus proven both (a) and $(1) \implies (2)$. To prove $(2) \implies (1)$, let $i = \frac{\text{N}-2}{2}$ 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 $\text{LeftChild}(n) = \frac{2\text{N}-1}{3}$ while expanding $m = \text{Flip}(i) = \text{Flip}\left(\frac{\text{N}-2}{3}\right)$ also gives us $\frac{2\text{N}-1}{3}$, as desired.

Suppose that x had 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 $p = \text{N} - 1 - (j+1) = \text{Flip}(j) - 1 = n - 1$. Note that (b) is clearly true for $\text{N} = 2$ and $\text{N} = 5$ so assume that $\text{N} > 5$. Since $\text{N} > 5$ 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 leaving node (k, p) and going into a min heap node. Thus the restriction of the BiHeap graph, B_N, to node $\{0, \ldots, j + 1\}$ contains the cycle:

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

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). The remaining statements follow immediately from what has already been shown.

To prove (e), let χ equal 1 if $\rho := \left\lceil \frac{\text{N}}{2} \right\rceil$ 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 $6x + 2 + 2\chi + (\text{N} \mod 2)$ and $2\left\lceil \frac{\rho}{2} \right\rceil - (\text{N} \mod 2)$.

Note that $\text{Parent}(i) = \text{Parent}\left(\frac{\text{N}-2}{3}\right) = \left\lfloor \left(\frac{\text{N}-2}{3} - 1\right)/2 \right\rfloor$ so that if N is even (resp. odd) then $\text{Parent}(i) = \frac{\text{N}-8}{6}$ (resp. $\text{Parent}(i) = \left(\frac{\text{N}-2}{3} - 1\right)/2$) 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 (f). ■

4.4. BiHeaps with Two Distinct Leaf Extended Edges Having Out-Degree 1.

Proposition 4.8. 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.

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

(3) 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 (3) 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

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

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 $\text{N} > 3$ 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 $k \geq \frac{\mathrm{N}}{3}$.
(f) A node (k,p) is in the min heap if and only if $k < \frac{2\mathrm{N}}{3}$.
(g) If $\mathrm{N} \neq 3$, then the number of pure min heap leaves with exactly two children in the min heap is

$$\frac{1}{3}\left(\left\lceil \frac{\rho}{2} \right\rceil - 3 + (\rho \mod 2) - (\mathrm{N} \mod 2)\right)$$

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

Remark 4.9. Note that statement 3 does *not* say that v has only one min heap child.

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

Suppose for the sake of contradiction that v had two min heap children, call the second one z. Were v not a child of z in the max heap then the uniqueness property of w would be violated. Thus v is a max heap child of z. But now condition (1) of proposition 4.7 is satisfied so by (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, 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. That the node w' is the last node in the max heap implies that $n' = j$, $j' = n$, and $m' = \mathrm{Parent}(n') = \left\lfloor \frac{j-1}{2} \right\rfloor = \left\lfloor \frac{(2i+1)-1}{2} \right\rfloor = i$ so that $i' = \mathrm{Flip}(m') = \mathrm{Flip}(i) = m$. Note that if $j' \leq i$ then w' being in the max heap would necessitate that v, and hence also w, belong to the max heap as well, which would cause the max heap to have a cycle. Thus $j' > i$. Now suppose that $j' > i+1$. Let $k = i+1$, $p = \mathrm{Flip}(k)$, $p' = m+1$, and $k' = \mathrm{Flip}(p')$. Let z be the node whose coordinate is (k,p) and let z' be the node whose coordinate is (k',p'). Since $n = j' > i+1 = m+1$, z does not belong to the max heap and 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 of the BiHeap graph on N nodes. Thus $j' \leq i+1$ so that $j' = i+1$. From $i+1 = j' = n = \mathrm{Flip}(j) = \mathrm{Flip}(\mathrm{LeftChild}(i))$, we conclude that $i = \frac{\mathrm{N}}{3} - 1$ so that in particular, 3 divides N. From $j = \mathrm{LeftChild}(i)$, $m = \mathrm{Flip}(i)$, and $n = \mathrm{Flip}(j)$ 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 $j = \mathrm{LeftChild}(i)$ and that $\mathrm{Parent}(m) = \left\lfloor \frac{\mathrm{N}}{3} - \frac{1}{2} \right\rfloor = \frac{\mathrm{N}}{3} - 1 \neq \frac{\mathrm{N}}{3} = n$, 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 $\mathrm{Parent}(m) = \left\lfloor \frac{\mathrm{N}}{3} - \frac{1}{2} \right\rfloor = \frac{\mathrm{N}}{3} - 1 = n - 1 = p$ 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 (3).

To prove (b), note that since 3 divides N, $\text{Parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor = \left\lfloor \frac{N}{6} \right\rfloor - 1$ so if N is even then $\text{Parent}(i) = \frac{N}{6} - 1$, which implies that $\text{LeftChild}(\text{Parent}(i)) = 2\left(\frac{N}{6} - 1\right) + 1 = \frac{N}{3} - 1 = i$. Conversely, $i = \text{Parent}(\text{LeftChild}(\text{Parent}(i)))$ implies $\frac{N}{3} - 1 = 2\left\lfloor \frac{N}{6} \right\rfloor - 1$ so that $\frac{N}{6} = \left\lfloor \frac{N}{6} \right\rfloor$, 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 $\frac{N}{3}$, $\left\lfloor \frac{N}{6} \right\rfloor = \left\lfloor \frac{N/3}{2} \right\rfloor = \frac{N/3}{2} - \frac{1}{2}$ so that $\text{RightChild}(\text{Parent}(i)) = \frac{N}{3} - 1 = i$. Conversely, if $i = \text{RightChild}(\text{Parent}(i))$ then $\frac{N}{3} - 1 = 2\left(\left\lfloor \frac{N}{6} \right\rfloor - 1\right) - 2$ gives us $\frac{N}{6} = \left\lfloor \frac{N}{6} \right\rfloor - \frac{1}{2}$ 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 $\rho := \left\lceil \frac{N}{2} \right\rceil$ 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 $6x + 4 + 2\chi + (N \mod 2)$ and $2\left\lceil \frac{\rho}{2} \right\rceil - (N \mod 2)$. ∎

4.5. BiHeaps Whose Heaps Lack Parents With a Single Child.

Proposition 4.10. The following are equivalent:

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

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 pure min heap, and let (i, m) be its coordinates. Then

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

so that, in particular, i, j, m, n, v, and w 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

$$\frac{1}{3}\left(\left\lceil \frac{\rho}{2} \right\rceil - 1 + (\rho \mod 2) - (N \mod 2)\right)$$

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

Proof. If (1) hold then propositions 4.7 and 4.8 imply that $N \mod 3 \neq 2$ and $N \mod 3 \neq 0$, 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 $\rho := \left\lceil \frac{N}{2} \right\rceil$ is even and 0 otherwise, let x be the desired quantity, and note that the number of BiHeap leaves is equal to both $6x + 2\chi + (N \mod 2)$ and $2\left\lceil \frac{\rho}{2} \right\rceil - (N \mod 2)$.

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

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

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 extended min heap edges.
(2) If ρ is even then by proposition 4.6, node with min heap coordinate $\text{Parent}(\rho - 1)$ has exactly one extended 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 $\rho - 1$ is incident to both a pure min edge and a pure max edge.

From this we conclude that the min heap has exactly $\mu := \rho + \frac{2}{3}\tau + \chi$ nodes and that $\tau = \left\lceil \frac{\rho}{2} \right\rceil - \alpha - \chi$, as we now explain. Recall that $\left\lceil \frac{\rho}{2} \right\rceil$ 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 extended 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 ρ is even, the last pure max heap node (which will necessarily belong to the min heap). From this we get (c).

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

5. Corollaries of the Classification of BiHeap Graphs.

5.1. Computing HeapSize (N).

From propositions 4.7, 4.8, and 4.10 we conclude that for all $\mathrm{N} > 0$,

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

Similarly, these propositions imply that for $\mathrm{N} > 3$, the number of pure min heap nodes incident to exactly one extended min heap edge is

$$((\rho + 1) \mod 2) + (((\mathrm{N} \mod 3) + 1) \mod 2).$$

where we let $\rho = \left\lceil \frac{\mathrm{N}}{2} \right\rceil$. They also imply that for $\mathrm{N} > 3$, the number of pure min heap leaves with exactly two children in the min heap is

$$\frac{1}{3}\left(\left\lfloor \frac{\rho}{2} \right\rfloor - (\mathrm{N} \mod 2) - 1 + (\rho \mod 2) - ((\mathrm{N}+2) \mod 3)\right)$$

while it is 0 for $\mathrm{N} \leq 3$.

5.2. In **Nodes.**

Notation 5.1. Let $\mathrm{In}(\mathrm{B_N})$ denote the set of nodes in the *in*tersection of the set of nodes in the min heap with the set of nodes in the max heap (i.e. the set of nodes common to both the min heap and the max heap). Let $\mathrm{In}(\mathrm{N})$ denote the number of nodes in $\mathrm{In}(\mathrm{B_N})$. Let $\mathrm{In_{PMin}}(\mathrm{B_N})$ ($\mathrm{In_{PMax}}(\mathrm{B_N})$) denote the nodes that belong to both the pure min (resp. pure max) heap and $\mathrm{In}(\mathrm{B_N})$ and let $\mathrm{In_{PMin}}(\mathrm{N}) = |\mathrm{In_{PMin}}(\mathrm{B_N})|$ (resp. $\mathrm{In_{PMax}}(\mathrm{N}) = |\mathrm{In_{PMax}}(\mathrm{B_N})|$). We will call elements of $\mathrm{In}(\mathrm{B_N})$ In *nodes*. By the *first* (resp. *last*) In *node* we mean the In node with the smallest (resp. largest) min heap coordinate.

Any node in $\mathrm{B_N}$ that is not an In node will be called an Out *node*. The set of Out nodes will be denoted by $\mathrm{Out}(\mathrm{B_N})$ and the number of Out nodes will be denoted by $\mathrm{Out}(\mathrm{N})$. Let $\mathrm{Out_{PMin}}(\mathrm{B_N})$

$(\mathrm{Out}_{\mathrm{PMax}}(\mathrm{B_N}))$ denote the nodes that belong to both the pure min (resp. pure max) heap and $\mathrm{Out}(\mathrm{B_N})$ and let $\mathrm{Out}_{\mathrm{PMin}}(\mathrm{N}) = |\mathrm{Out}_{\mathrm{PMin}}(\mathrm{B_N})|$ (resp. $\mathrm{Out}_{\mathrm{PMax}}(\mathrm{N}) = |\mathrm{Out}_{\mathrm{PMax}}(\mathrm{B_N})|$) ■

Observe that

- $\mathrm{In}_{\mathrm{PMin}}(\mathrm{B_N})$ $(\mathrm{In}_{\mathrm{PMax}}(\mathrm{B_N}))$ denotes those leaves of $\mathrm{PMin}(\mathrm{N})$ (resp. $\mathrm{PMax}(\mathrm{N})$) that belong to the max (resp. min) heap.
- $\mathrm{In}(\mathrm{B_2}) = \mathrm{B_2}$ and $\mathrm{In}(\mathrm{B_3})$ consists solely of the $\mathrm{B_3}$'s middle node.
- If N is odd then $\mathrm{In}(\mathrm{B_N})$ contains the BiHeap's middle node.
- $\mathrm{Out}(\mathrm{N})$ is even.
- The number of In nodes in $\mathrm{B_N}$ is
$$\begin{aligned}\mathrm{In}(\mathrm{N}) &= \mathrm{N} - 2(\mathrm{N} - \mathrm{HeapSize}(\mathrm{N}))\\ &= \mathrm{N} - 2\left\lfloor \frac{\mathrm{N}}{3} \right\rfloor\\ &= 2\left\lceil \frac{2\,\mathrm{N}}{3} \right\rceil - \mathrm{N}\\ &= \frac{\mathrm{N} + 2(\mathrm{N} \mod 3)}{3}\end{aligned}$$
so that in particular, $3\,\mathrm{In}(\mathrm{N}) - 4 \le 3\,\mathrm{In}(\mathrm{N}) - 2(\mathrm{N} \mod 3) = \mathrm{N} \le 3\,\mathrm{In}(\mathrm{N})$ and the number of Out nodes is $\frac{2}{3}[\mathrm{N} - (\mathrm{N} \mod 3)]$.
- The last In node has min heap coordinate $\mathrm{HeapSize}(\mathrm{N}) - 1$ and the first In node has min heap coordinate
$$\begin{aligned}\mathrm{Flip}(\mathrm{HeapSize}(\mathrm{N}) - 1) &= \mathrm{N} - \mathrm{HeapSize}(\mathrm{N})\\ &= \left\lfloor \frac{\mathrm{N}}{3} \right\rfloor\\ &= \frac{\mathrm{N} - (\mathrm{N} \mod 3)}{3}\end{aligned}$$
- For any positive integers k and N, $k = |\mathrm{In}(\mathrm{B_N})|$ if and only if $\mathrm{N} = 3k, 3k-2$, or $3k-4$.
- The recurrence $\mathrm{N}_{i+1} = 3\,\mathrm{N}_i + 4$, with $\mathrm{N}_0 > 1$ given, has the solution $\mathrm{N}_i = (\mathrm{N}_0 - 2)\,3^i + 2$, where this represents a sequence of BiHeap sizes such that $\mathrm{In}(\mathrm{N}) = \mathrm{N}_{i-1}$ and $\mathrm{N}_i \mod 3 = 2$ for all $i > 0$. Such sequences are important to consider due to the additional complexity that the existence of a double arrow introduces into BiHeaps that have them.

Part 3. BiHeapification

Having obtained closed form formulas for HeapSize (), we can now define a function to take any collection of N elements and form a directed BiHeap with respect to any given total ordering on these elements.

6. Overview of BiHeapification. Recall that if we want to heapify N elements using an $O(\mathrm{N})$ algorithm, then we proceed inductively by first heapifying elements $\mathrm{N} - (k-1), \ldots, \mathrm{N} - 1$ (which in a sense constitute "the current heap") and then we sift down element $\mathrm{N} - k$ by at most $\lceil \log_2(\mathrm{N}+1) \rceil - \lceil \log_2(k+2) \rceil$ levels. Our BiHeapify () algorithm will essentially start by doing the same thing, except that after it has sifted down to an element that belongs to both the min heap and the max heap, it will begin to sift up.

For notational reasons, we describe the operation for the case where N is odd. The description in the case where N is even is almost identical, but for the notation of "the current BiHeap" that will be defined shortly. If $\rho = \left\lfloor \frac{\mathrm{N}}{2} \right\rfloor$, then the BiHeapify () algorithm will proceed inductively as follows:

(1) Suppose that we have BiHeapified elements $\rho - (k-1), \ldots, \rho - 1, \rho, \rho + 1, \ldots, \rho + (k-1)$, 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) We now 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 BiHeapify () operation is $O(\mathrm{N})$.

(3) At this point, we will have BiHeapified elements $\rho - k, \rho - (k-1), \ldots, \rho, \ldots, \rho + (k-1)$.

(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 $\rho - k, \ldots, \rho, \ldots, \rho + k$ 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.

7. 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 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 $O(\mathrm{N})$.

```
//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)
  if (pos_mc == 0 or Parent(pos_mc) < last_node_in_biheap_mc)
    return
  int  parent_mc = Parent(pos_mc)
  Node pos_node  = V + Flip(pos_mc)
  do:
    parent_mc          = Parent(pos_mc)
    Node parent_node = V + Flip(parent_mc)
    if (*pos_node <= *parent_node)
      return
    SwapValues(pos_node, parent_node)
    pos_mc    = parent_mc
    pos_node = parent_node
  while (pos_mc > 0 and Parent(pos_mc) >= last_node_in_biheap_mc)
  return

//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)
  if (pos_hc == 0 or Parent(pos_hc) < first_node_in_biheap_hc)
    return
  int  parent_hc = Parent(pos_hc)
  Node pos_node  = V + pos_hc
  do:
    parent_hc        = Parent(pos_hc)
    Node parent_node = V + parent_hc
    if (*pos_node >= *parent_node)
      return
    SwapValues(pos_node, parent_node)
    pos_hc   = parent_hc
    pos_node = parent_node
  while (pos_hc > 0 and Parent(pos_hc) >= first_node_in_biheap_hc)
  return
```

8. 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).

```
//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)
  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.
  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
    bool is_right_child_valid = right_child_hc <= last_node_in_biheap_hc and
                                right_child_hc < heap_size
    Node smaller_node
    if (is_right_child_valid and *right_child_node < *left_child_node)
      smaller_node = right_child_node
      pos_hc       = right_child_hc
    else
      smaller_node = left_child_node
      pos_hc       = left_child_hc
    if (*pos_node <= *smaller_node)
      return
    SwapValues(pos_node, smaller_node)
  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
// Flip(pos_mc) >= first_node_in_biheap_hc.
void SiftFromMaxToMin(Node V, int N, int pos_mc, int first_node_in_biheap_hc)
  int heap_size                  = HeapSize(N)
  int first_node_in_mirror_heap = N - heap_size
  int pos_hc                     = Flip(pos_mc)
  //Sift down the max heap while not yet in the min heap.
  while (pos_mc < first_node_in_mirror_heap)
    int  left_child_mc     = LeftChild(pos_mc)
    int  right_child_mc    = RightChild(pos_mc)
    int  left_child_hc     = Flip(left_child_mc)
    int  right_child_hc    = Flip(right_child_mc)
    Node pos_node         = V + pos_hc
    Node left_child_node  = V + left_child_hc
    Node right_child_node = V + right_child_hc
    //Note that right_child_hc >= first_node_in_biheap_hc necessarily holds.
    bool is_right_child_valid = right_child_mc < heap_size
    Node larger_node
    if (is_right_child_valid and *right_child_node > *left_child_node)
      larger_node = right_child_node
      pos_hc      = right_child_hc
      pos_mc      = right_child_mc
    else
      larger_node = left_child_node
      pos_hc      = left_child_hc
      pos_mc      = left_child_mc
    if (*pos_node >= *larger_node)
      return
    SwapValues(pos_node, larger_node)
  SiftUpMinHeap(V, N, pos_hc, first_node_in_biheap_hc)
  return
```

```
int MinHeapCoordinateOfLastMinHeapNode(int size_of_heap)
  return size_of_heap - 1
```

```
void BiHeapify(Node V, int N)
  if (N < 2)
    return
  int heap_size = HeapSize(N)
  //Ignore all In nodes,...
  int last_node_in_biheap_hc = MinHeapCoordinateOfLastMinHeapNode(heap_size)
  if (N % 3 == 2)              //...unless N mod 3 == 2, in which case
    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)
  while (first_node_in_biheap_hc > 0)
    first_node_in_biheap_hc-- //Sift the next pure min heap node.
    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.
    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 $\sum_{h=0}^{\infty} h \lceil \frac{\mathrm{N}}{2^{h+1}} \rceil \leq 2\,\mathrm{N}$, it should be now obvious how the usual proof that the heapify operation is $O(\mathrm{N})$ generalizes immediately to proving that the BiHeapify() operation is $O(\mathrm{N})$.

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 $h = \mathrm{HeapSize}(\mathrm{N})$ is the size of the min heap, then the total cost of all calls to SiftFromMinToMax is bounded above by

$$h + 2\sum_{l=0}^{\infty} l \left\lceil \frac{h}{2^{l+1}} \right\rceil \leq 5h$$

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 $10h \leq 10\lceil \frac{2\,\mathrm{N}}{3} \rceil \leq \frac{20\,\mathrm{N}}{3} + 20$. However, notice that we skip applying these functions to all In nodes that are not incident to a double arrow, of which there are at least $\frac{\mathrm{N} - (\mathrm{N} \mod 3)}{3}$ so that for $\mathrm{N} \geq 60$, the total cost is in fact at most $5\,\mathrm{N}$.

Empirical testing indicates that it is likely that the total number of swaps required to BiHeapify any collection of N nodes is at most $\frac{7}{3}\mathrm{N}$ for all $\mathrm{N} > 0$, where the ratio of the cost of calling BiHeapify to N appears to approach $\frac{7}{3}$ as N increases. Indeed, it is easy to see that the cost of calling BiHeapify on N integer valued nodes when node the node with min heap coordinate i initially has value $\mathrm{N} - i$ (i.e. the values are initially strictly descending) gives a close approximation (if not the exact value) of the maximum possible cost of BiHeapifying N nodes.

Observe that the BiHeapify algorithm is idempotent. That is, calling BiHeapify () on a list of nodes that have 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

```
int first_in_node = N - heap_size
int num_in_nodes  = heap_size - first_in_node
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 $O(\mathrm{N})$ recursive algorithm that would no longer be idempotent.

Part 4. 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 $O(\log \mathrm{N})$ algorithm. We describe this algorithm although it will not be used elsewhere.

```
void BiHeapSift(Node V, int N, int pos_hc)
  if (N < 2)
    return
  int heap_size = HeapSize(N)
  int pos_mc    = Flip(pos_hc)
  auto value    = *(V + pos_hc)
  bool is_node_in_min_heap = pos_hc < heap_size
  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))
    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))
    SiftFromMaxToMin(V, N, pos_mc, 0)
  else if (is_node_in_min_heap and *(V + Parent(pos_hc)) > value)
    SiftUpMinHeap(V, N, pos_hc, 0)
  else
    SiftUpMaxHeap(V, N, pos_mc, 0)
  return
```

At the time of writing, it is not known whether or not there exists an $O(\log \mathrm{N})$ 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 5. Recursively BiHeapifying In Nodes with BiHeapifyInwards ()

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

Definition 9.1. Let T be a rooted tree. By *an ancestor* of a node v in T we mean any node, including v, that lies along the unique shortest path from v to the root. If v is a node in T then $\mathrm{Ancestors}(v)$ denotes the set of all ancestors of v in T. If S is a subset of T then $\mathrm{Ancestors}(S)$ denotes $\bigcup_{v \in S} \mathrm{Ancestors}(v)$. ■

Lemma 9.2. Let B be rooted binary tree on $\mathrm{N} > 1$ nodes. Let Λ be a non-empty subset of B's leaves and let $T = \mathrm{Ancestors}(\Lambda)$. Let S denote the set of nodes v 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 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 9.3. If T is a tree then let $\Lambda(T)$ denote the leaves of the tree. ■

Proposition 9.4. Suppose that the $\mathrm{N} > 0$ and that the values $^*\mathrm{V}, \ldots, ^*(\mathrm{V}+(\mathrm{N}-1))$ form a BiHeap, B. Let $I = \text{In}(B)$ and let pivot_value be a value such that there exist nodes $\mathrm{V}+i$ and $\mathrm{V}+j$ in I such that $*(\mathrm{V}+i) \leq \text{pivot\_value} \leq *(\mathrm{V}+j)$. Let Σ (resp. Γ) denote the set of values in B that are less than or equal to (resp. greater than or equal to) pivot_value. Let C denote $\Sigma \cap I$ (resp. $\Gamma \cap I$). Let χ be 1 if $\mathrm{N} \mod 3 = 2$ and C contains the last node of the min (resp. max) heap and let it be 0 otherwise.

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

If any of the following conditions hold

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

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

Proof. Note that if $\mathrm{N} = 5$ and Σ does not contain the last node of the min heap then the conclusion can be immediately verified. So assume that $\mathrm{N} \neq 5$. If $\mathrm{N} = 2, 4,$ or 7 and $\Sigma \cap I \neq I$ then the conclusion can be directly verified. The conclusion can also be directly verified for $\mathrm{N} = 3$ and $\mathrm{N} = 6$ so assume that N is neither of these. We may thus assume that $\mathrm{N} > 7$.

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 $|\Sigma| \geq \text{HeapSize}(\mathrm{N}) = \left\lceil \frac{2\mathrm{N}}{3} \right\rceil = \frac{\mathrm{N}+2(\mathrm{N} \mod 3)}{3}$. Now, $2|\Sigma \cap I| - (\mathrm{N} \mod 3) = 2|I| - (\mathrm{N} \mod 3) = 2\left(\frac{\mathrm{N}+2(\mathrm{N} \mod 3)}{3}\right) - (\mathrm{N} \mod 3) = \text{HeapSize}(\mathrm{N})$. Note in particular that if $\mathrm{N} \mod 3 = 0$ 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 $\mathrm{N} \mod 3 \neq 2$ then all of the remaining assumption imply that there exists some leaf in the min heap that does not belong to T. Lemma 9.2 now allows us to conclude that T necessarily contains some node with exactly one child. If, however, $\mathrm{N} \mod 3 = 2$ 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 $3|\text{In}(\mathrm{B_N})| - 4 \leq 3|\text{In}(\mathrm{B_N})| - 2(\mathrm{N} \mod 3) = \mathrm{N} \leq 3|\text{In}(\mathrm{B_N})|$, the inequality $\frac{|\Sigma|}{\mathrm{N}} \geq \frac{2}{3} \frac{|\Sigma \cap I|}{|I|}$ now follows from $\frac{1}{\mathrm{N}} \geq \frac{1}{|I|}$.

A similar argument proves the inequalities involving Γ. ■

10. 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 9.4, the following $O(\mathrm{N})$ 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.

```
void BiHeapifyInwards(Node V, int N)
  if (N < 10)
    sort(V, N) //Sort nodes V, V + 1, ..., V + (N - 1)
    return
  BiHeapify(V, N) //This is an O(N) operation.
  int heap_size      = HeapSize(N)
  int first_in_node = N - heap_size //= Flip(heap_size - 1)
  int num_in_nodes  = heap_size - first_in_node
  //The inequality below necessarily holds, which implies that this is an O(N) algorithm.
  assert(num_in_nodes <= (N + 4) / 3)
  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 $|\mathrm{In}(\mathrm{B_N})| = \frac{\mathrm{N}+2(\mathrm{N} \mod 3)}{3}$ 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 $C\left(\mathrm{N} - \frac{4}{3}\right)$ operations then for all sufficiently large N, BiHeapifyInwards performs no more than $\frac{3C}{2}\left(\mathrm{N} - \frac{4}{3}\right)$ operations.

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 $\frac{7}{2}\mathrm{N}$. Indeed this is what empirical testing shows with no test performing more than $\frac{7}{2}\mathrm{N}$ swaps and with the ratio of the cost of calling BiHeapifyInwards to N appearing to approach $\frac{7}{2}$ as N increases.

11. 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 9.4 that give desired inequalities are not necessarily satisfied. We now show how a slight modification rectifies this issue. First, we will need any $O(\mathrm{N})$ 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)
  BiHeapify(V, N)
  return
```

We also define an $O(\mathrm{N})$ function 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) such that a is the largest integer such that $^*(\mathrm{V}+a)$ has minimal value and b is the smallest non-negative integer such that $^*(\mathrm{V}+b)$ has maximal value.

```
pair<int, int> EmplaceAllMinsAndMaxs(Node V, int N)
  EmplaceMinAndMax(V, N)
  int min_index = 0, max_index = N - 1
  auto min_value = *(V + min_index)
  auto max_value = *(V + max_index)
  if (min_value == max_value)
    return (max_index, 0)
  if (N < 3)
    return (0, max_index)
  int i = 1
  while (i < max_index)
    auto current_value = *(V + i)
    if (*(V + i) == min_value)
      min_index++
      if (min_index < i)
        SwapValues(V + min_index, V + i)
      else
        i++
    else if (current_value == max_value)
      max_index--
      SwapValues(V + max_index, V + i)
    else
      i++
  return (min_index, max_index)
}
```

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

```
void BiHeapifyInwardsNicerMath(Node V, int N)
  if (N < 30)
    sort(V, N) //Sort nodes V, V + 1, ..., V + (N - 1)
  if (N < 10)
    return
  (a, b) = EmplaceAllMinsAndMaxs(V, N)
  if (a >= N / 2 or b <= (N - 1) / 2) //If we've emplaced the median(s).
    return
  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.
    return
  int new_N = N
  if (new_N % 3 == 2)
   V      = V + 1
    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
  // 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.
  BiHeapify(V, new_N)
  int heap_size     = HeapSize(new_N)
  int first_in_node = new_N - heap_size
  int num_in_nodes  = heap_size - first_in_node
  BiHeapifyInwardsNicerMath(V + first_in_node, num_in_nodes)
  return
```

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

Lemma 11.1. Let $\chi_{\mathrm{N}} = \begin{cases} 0 & \text{if N} \mod 3 = 0 \\ 2 & \text{if N} \mod 3 = 1 \\ -2 & \text{if N} \mod 3 = 2 \end{cases}$ and let $I(\mathrm{N}) = \frac{\mathrm{N} + \chi_{\mathrm{N}}}{3}$, so that $I(\mathrm{N})$ is the number of nodes 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 $I^i(\mathrm{N})$ is of the form $3^k + 2$ for some non-negative integer k, in which case $I^{i+l}(\mathrm{N}) = 3^{k-l}$ for all $1 \leq l < k$. For $\mathrm{N} > 8$, if N is not of the form $3^k + 2$ then $\lceil \log_3(\mathrm{N}) \rceil = \lceil \log_3(I(\mathrm{N})) \rceil + 1$ and otherwise $\lceil \log_3(\mathrm{N}) \rceil = \lceil \log_3(I(\mathrm{N})) \rceil + 2$.

Let ξ_{N} be 1 if there exists some non-negative integer i such that $I^i(\mathrm{N}) = 3^k + 2$ for some integer $k > 1$ and let it be 0 otherwise. Let $\iota(\mathrm{N})$ 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 $\mathrm{N} > 5$, $\iota(\mathrm{N}) = \lceil \log_3(\mathrm{N}) \rceil - 2 - \xi_{\mathrm{N}}$. In particular, if $\mathrm{N} > 3$ is even then BiHeapifyInwardsNicerMath() would call itself at most $\lceil \log_3(\mathrm{N}) \rceil - 2$ times.

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

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

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

11.2. Pivots.

Definition 11.2. Suppose that we're given a list of N values $^*\mathrm{V}, \ldots, ^*(\mathrm{V}+(\mathrm{N}-1))$ and that after calling some function F() on this list, this list of values has been changed so as to become the list $v_0, \ldots, v_{\mathrm{N}-1}$.

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

11.3. Minimum Number of Elements Below and Above BiHeapifyInwardsNicerMath()-Pivots.

It now follows from lemma 11.1 and proposition 9.4 that as N increases exponentially so too does the number of values that are less (resp. greater) than or equal to the BiHeapifyInwardsNicerMath()-left (resp. right) pivot value, for *any* set of N inputs. We state this formally.

Theorem 11.3. Let χ_{N}, I, and ξ_{N} be defined as in lemma 11.1. Suppose that the $\mathrm{N} > 0$ and that the values $^*\mathrm{V}, \ldots, ^*(\mathrm{V}+(\mathrm{N}-1))$, denoted by B, are passed to BiHeapifyInwardsNicerMath(). Let $\mathrm{pivot\_value}_{\mathrm{L}}$ (resp. $\mathrm{pivot\_value}_{\mathrm{R}}$) be the BiHeapifyInwardsNicerMath()-left (resp. right) pivot value (def. 11.2) and let Σ (resp. Γ) denote the set of values in B that are less than or equal to $\mathrm{pivot\_value}_{\mathrm{L}}$ (resp. greater than or equal to $\mathrm{pivot\_value}_{\mathrm{R}}$).

Then for all $\mathrm{N} \geq 16$, $|\Sigma| \geq 2^{\lceil \log_3 \mathrm{N} \rceil + 1 - \xi_{\mathrm{N}}}$ and $|\Gamma| \geq 2^{\lceil \log_3 \mathrm{N} \rceil + 1 - \xi_{\mathrm{N}}}$. In particular, for all $\mathrm{N} \geq 16$, $\mathrm{N} \geq 16$, $|\Sigma| \geq 2^{\lceil \log_3 N \rceil}$ and $|\Gamma| \geq 2^{\lceil \log_3 \mathrm{N} \rceil}$ where observe that $2^{\lceil \log_3 \mathrm{N} \rceil} \geq \mathrm{N}^{\frac{1}{\log_2 3}}$ 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 $16 \leq \mathrm{N} < 30$ so assume that $\mathrm{N} > 29$. Suppose first that $\mathrm{N} \mod 3 = 0$ or 1. Note that $\xi_{\mathrm{N}} = \xi_{I(\mathrm{N})}$. By our inductive hypothesis and proposition 9.4 we have that

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

So we will now henceforth assume that $\mathrm{N} \mod 3 = 2$.

Note that we have

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

Note that if N is not of the form $3^k + 2$ for some integer k then $\xi_{\mathrm{N}} = \xi_{I(\mathrm{N})}$ and we can proceed as before to conclude that $|\Sigma| \geq 2^{\lceil \log_3 (\mathrm{N}) \rceil + 1 - \xi_{\mathrm{N}}}$. So assume that $\mathrm{N} = 3^k + 2$ for some integer k, which gives us that $\xi_{\mathrm{N}} = 1$. From lemma 11.1, we can conclude that $\xi_{I(\mathrm{N})} = 0$. Note that

$$\lceil \log_3 (\mathrm{N} - 2) \rceil = \left\lceil \log_3 \left(3^k\right) \right\rceil = k = \lceil \log_3 (\mathrm{N}) \rceil - 1$$

so that

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

which gives us our desired conclusion. ∎

Now although $2^{\lceil \log_3 \mathrm{N} \rceil}$ 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 $0.48\,\mathrm{N}$. This means that in applications, this $O(\mathrm{N})$ 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 $0.48\,\mathrm{N}$), and, by virtue of the inequalities in proposition 9.4, is guaranteed to have a minimum of *approximately* $2^{\lceil \log_3 \mathrm{N} \rceil}$ values above and below it. However, the exact expression for this lower bound is of course much more complicated than that given in theorem 11.3, which is why although BiHeapifyInwards() is recommended, we instead investigated its variant, BiHeapifyInwardsNicerMath().

Part 6. Fused BiHeaps

Observe that if v is an In node in $\mathrm{B_N}$ and $\mathrm{N} > 2$ 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 implemented a doubled ended heap that allows for amortized $O(\mathrm{N})$ insertions and pops.

Definition 11.4. 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 reserve 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 11.5. Suppose that F is some set (possibly empty) of In nodes from the BiHeap $\mathrm{B_N}$, 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* $\mathrm{B_N}$*)*, denoted by $\mathrm{B}_{\mathrm{N}}^{\mathrm{Fuse}(F)}$, we mean the graph defined by $\mathrm{B}_{\mathrm{N}}^{\mathrm{Fuse}(\varnothing)} = \mathrm{B_N}$ if $F = \varnothing$, $\mathrm{N} = 1$, or $\mathrm{N} = 2$, or otherwise if $F \neq \varnothing$ and $\mathrm{N} > 2$ then it is the graph obtained as follows: denote the distinct elements of F by $v_1, \ldots, v_f$ and for all $1 \leq l \leq f$, inductively define $\mathrm{B}_{\mathrm{N}}^{\mathrm{Fuse}(\{v_1,\ldots,v_l\})}$ to be the graph $\mathrm{B}_{\mathrm{N}}^{\mathrm{Fuse}(\{v_1,\ldots,v_l\} \setminus \{v_l\})}$ fused at v_l.

If a graph is the F-fused BiHeap graph from $\mathrm{B_N}$ then we will call $\mathrm{B_N}$ a *parent (BiHeap)* of the F-fused BiHeap graph. ∎

Remark 11.6. Observe that this definition is in fact independent of the ordering of F's elements. That is, if α is a permutation of $\{1, \ldots, f\}$ then repeating the above construction with the vertices ordered by $v_{\alpha(1)}, \ldots, v_{\alpha(f)}$ will result in the same graph. Also observe that definition 11.4 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 $\mathrm{Parent}_{\mathrm{Min}}(v)$ and $\mathrm{Parent}_{\mathrm{Max}}(v)$.

Definition 11.7. If v is an In node in B_{N} and $\mathrm{N} > 2$ then we will call the nodes $\mathrm{Parent}_{\mathrm{Min}}(v), v,$ and $\mathrm{Parent}_{\mathrm{Max}}(v)$ the *triple at* v and we will say that this triple is *ordered* or that it *satisfies the BiHeap triple condition* if $^*\mathrm{Parent}_{\mathrm{Min}}(v) \leq {}^*v \leq {}^*\mathrm{Parent}_{\mathrm{Max}}(v)$.

Suppose that $\mathrm{N} \mod 3 = 2$ 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 arrow, so $w = \mathrm{Parent}_{\mathrm{Max}}(v)$ and $v = \mathrm{Parent}_{\mathrm{Min}}(w)$). We will call the nodes $\mathrm{Parent}_{\mathrm{Min}}(v), v, w,$ and $\mathrm{Parent}_{\mathrm{Max}}(w)$ the the BiHeap's *quadruple* and we will say that this quadruple is *ordered* or that it *satisfies the BiHeap quadruple condition* if $^*\mathrm{Parent}_{\mathrm{Min}}(v) \leq {}^*v \leq {}^*w \leq {}^*\mathrm{Parent}_{\mathrm{Max}}(w)$.

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. $^*\mathrm{Parent}_{\mathrm{Min}}(v) \leq {}^*v \leq {}^*\mathrm{Parent}_{\mathrm{Max}}(v)$ in def. 11.7) 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. $^*\mathrm{Parent}_{\mathrm{Min}}(v) \leq {}^*\mathrm{Parent}_{\mathrm{Max}}(v)$).

Definition 11.8. Let $\mathrm{V}, \ldots, \mathrm{V} + (\mathrm{N} - 1)$ be a collection of $\mathrm{N} > 0$ 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 $v = \mathrm{Parent}_{\mathrm{Min}}(w)$ and $x = \mathrm{Parent}_{\mathrm{Max}}(w)$ then $^*v \leq {}^*w$.
(3) (*The weakened quadruple condition*) If $\mathrm{N} \mod 3 = 2$ then we also require that $^*\mathrm{Parent}_{\mathrm{Min}}(\mathrm{Parent}_{\mathrm{Min}}(w)) \leq {}^*\mathrm{Parent}_{\mathrm{Max}}(w)$ 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 $^*\mathrm{Parent}_{\mathrm{Max}}(\mathrm{Parent}_{\mathrm{Max}}(z)) \geq {}^*\mathrm{Parent}_{\mathrm{Min}}(z)$ 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 $S = \{V + i_1, \ldots, V + i_k\}$ 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 $\mathrm{B}_{\mathrm{N}}^{\mathrm{Fuse}(F)}$ is an *F-Fused BiHeap (from* B_{N}*)* and it is a *Fused BiHeap (from* B_{N}*) permitting* S if either $\mathrm{N} = 1$ or otherwise if it satisfies the Fully Fused BiHeap condition and:

(1) if $\mathrm{N} = 2$ then either there exists a non-permitted node or else $\mathrm{B}_{\mathrm{N}}^{\mathrm{Fuse}(F)} = \mathrm{B}_2$ forms a BiHeap.
(2) if $\mathrm{N} > 2$ the following conditions hold for all permitted nodes $v = V + i_l$:
    (a) If v is not the end of a double arrow or if it is the end of a double arrow and both of the double arrow's ends are permitted, then we require that v satisfy the BiHeap triple condition.
    (b) If v is the end of a double arrow and exactly one end of the double arrow is permitted then we require that

(i) $*v \le *\,\mathrm{Parent}_{\mathrm{Max}}(\mathrm{Parent}_{\mathrm{Max}}(v))$ if v belongs to the pure min heap, or else
(ii) $*v \ge *\,\mathrm{Parent}_{\mathrm{Min}}(\mathrm{Parent}_{\mathrm{Min}}(v))$ if v belongs to the pure max heap.

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

If $\mathrm{V}, \ldots, \mathrm{V} + (\mathrm{N}-1)$ 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 $\mathrm{N} \mod 3 = 2$ then in addition it guarantees that the quadruple (def. 11.7) 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.

12. FusedBiHeapify **with a Range of Permitted Nodes.**

We now give an algorithm that, given a range $F = \{F_0, \ldots, F_{f-1}\}$ of f In nodes in $\mathrm{B_N}$, makes $\mathrm{B_N^{Fuse(\mathit{F})}}$ into an Fused BiHeap permitting $\mathrm{In}(\mathrm{B_N}) \smallsetminus F$. 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 $O(\mathrm{N})$ complexity. However, since $\mathrm{In}(\mathrm{N}) = \frac{\mathrm{N} + 2(\mathrm{N} \mod 3)}{3}$ is $O(\mathrm{N})$, 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 $O(\mathrm{N})$ complexity.

The case where $\mathrm{N} \mod 3 = 2$ adds so much additional complexity, that we separate the algorithm into the two cases of $\mathrm{N} \mod 3 = 2$ and $\mathrm{N} \mod 3 \ne 2$. 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 $O(\mathrm{N})$ complexity.

If $F = \varnothing$ then this is indicated by calling this FusedBiHeapify () with any values of first_F_hc and last_F_hc such that last_F_hc < 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.

```
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)
  else
    FusedBiHeapifyWithDHA(V, N, fuse_first_hc, fuse_last_hc)
  return
```

We separate it into two cases based on that value of $\mathrm{N} \mod 3$ for two reasons. The first is that if $\mathrm{N} \mod 3 = 2$ then having to deal with the double 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 $\mathrm{N} \mod 3 \ne 2$ so we may avoid the additional complexity that exists when $\mathrm{N} \mod 3 = 2$. There is not much insight to be gained from the additional complexity of the $\mathrm{N} \mod 3 = 2$ 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(Note V, int N)
  FusedBiHeapify(V, N, 0, N)
  return
```

12.1. FusedBiHeapify **With** $\mathrm{N} \mod 3 \ne 2$.

In this case, the BiHeap $\mathrm{B_N}$ 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 mod 3 ≠ 2, it also works in the case where N mod 3 = 2 and neither endpoint of the double headed arrow is included in F.

```
void FusedBiHeapifySiftFromMinToMaxNoDHA(Node V, int N, int pos_hc,
                        int last_node_in_biheap_hc, int first_F_hc, int last_F_hc)
  int heap_size         = HeapSize(N)
  int first_in_node_hc = N - heap_size
  while (pos_hc < first_in_node)
    int  left_child_hc  = LeftChild(pos_hc)
    int  right_child_hc = RighChild(pos_hc)
    bool is_right_valid = false
    if (right_child_hc <= last_node_in_biheap_hc and right_child_hc < heap_size)
      is_right_valid = true
    bool is_left_valid = false
    if (left_child_hc <= last_node_in_biheap_hc and left_child_hc < heap_size)
      is_left_valid = true
    //If the left child is in F then "skip over it."
    if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)
      left_child_hc = Flip(Parent(Flip(left_child_hc)))
    //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)))
    //If a child did have to be "skipped over", then is it still valid?
    if (is_right_valid and right_child_hc <= last_node_in_biheap_hc) is_right_valid = true
    else                                                             is_right_valid = false
    if (is_left_valid and left_child_hc <= last_node_in_biheap_hc) is_left_valid  = true
    else                                                           is_left_valid  = false
    if (!is_left_valid and !is_right_valid)
      return
    Node pos_node         = V + pos_hc
    Node left_child_node  = V + left_child_hc
    Node right_child_node = V + right_child_hc
    Node smaller_node
    if (!is_left_valid or (is_right_valid and *right_child_node < *left_child_node))
      smaller_node = right_child_node
      pos_hc       = right_child_hc
    else //Here, the left child is valid.
      smaller_node = left_child_node
      pos_hc       = left_child_hc
    if (*pos_node <= *smaller_node)
      return
    SwapValues(pos_node, smaller_node)
  SiftUpMaxHeap(V, N, Flip(pos_hc), Flip(last_node_in_biheap_hc))
  return
```

```
void FusedBiHeapifySiftFromMaxToMinNoDHA(Node V, int N, int pos_mc,
                        int first_node_in_biheap_hc, int first_F_hc, int last_F_hc)
  int heap_size         = HeapSize(N)
  int first_in_node_hc = N - heap_size
  int pos_hc            = Flip(pos_mc)
  while (pos_mc < first_in_node)
    int left_child_mc  = LeftChild(pos_mc)
    int right_child_mc = RightChild(pos_mc)
    int left_child_hc  = Flip(left_child_mc)
    int right_child_hc = Flip(right_child_mc)
    bool is_right_valid = false
    if (right_child_mc < heap_size and right_child_hc >= first_node_in_biheap_hc)
      is_right_valid = true
    if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)
```

```
      left_child_hc = Parent(left_child_hc)
      left_child_mc = Flip(left_child_hc)
    if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
      right_child_hc = Parent(right_child_hc)
      right_child_mc = Flip(right_child_hc)
    if (is_right_valid and right_child_hc >= first_node_in_biheap_hc) is_right_valid = true
    else                                                              is_right_valid = false
    bool is_left_valid = false
    if (left_child_hc >= first_node_in_biheap_hc)
      is_left_valid = true
    if (!is_left_valid and !is_right_valid)
      return
    Node pos_node         = V + pos_hc
    Node left_child_node  = V + left_child_hc
    Node right_child_node = V + right_child_hc
    Node larger_node
    if (!is_left_valid or (is_right_valid and *right_child_node > *left_child_node))
      larger_node = right_child_node
      pos_hc      = right_child_hc
      pos_mc      = right_child_mc
    else //Here, the left child is valid.
      larger_node = left_child_node
      pos_hc      = left_child_hc
      pos_mc      = left_child_mc
    if (*pos_node >= *larger_node)
      return
    SwapValues(pos_node, larger_node)
  SiftUpMinHeap(V, N, pos_hc, first_node_in_biheap_hc)
  return
```

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

Remark 12.1. Observe that the call FusedBiHeapifyNoDHA $(V, \mathrm{N}, \mathrm{N}, 0)$ 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 $(V, \mathrm{N}, \mathrm{N}, 0)$ found in the appendix. Thus, the call FusedBiHeapify $(V, \mathrm{N}, \mathrm{N}, 0)$

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.

13. 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 $\mathrm{N} \mod 3 \neq 2$. For the algorithm that sifts an element in the case where $\mathrm{N} \mod 3 = 2$, the reader is refered to the implementation present in the code in author's GitHub account. We now givethe 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 ≤ pos_hc ≤ last_F_hc.

```
void FusedBiHeapSiftNoDHA(Node V, int N, int pos_hc, int first_F_hc, int last_F_hc)
  if (N < 2)
    return
  int  heap_size = HeapSize(N)
  int  pos_mc    = Flip(pos_hc)
  Node pos_it    = V + pos_hc
  auto pos_value = *pos_it
  bool is_node_in_min_heap = pos_hc < heap_size
  bool is_node_in_max_heap = pos_mc < heap_size
  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)
    int  maxh_parent_of_pos_mc = Parent(pos_mc)
    int  maxh_parent_of_pos_hc = Flip(maxh_parent_of_pos_mc)
    Node minh_parent_of_pos_it = V + minh_parent_of_pos_hc
    Node maxh_parent_of_pos_it = V + maxh_parent_of_pos_hc
    if (pos_value > *maxh_parent_of_pos_it)
      SwapValues(pos_it, maxh_parent_of_pos_it)
      SiftUpMaxHeap(V, N, N - 1 - maxh_parent_of_pos_hc, 0)
    else if (pos_value < *minh_parent_of_pos_it)
      SwapValues(pos_it, minh_parent_of_pos_it)
      SiftUpMinHeap(V, N, minh_parent_of_pos_hc, 0)
    return
  //At this point, pos is not an In node.
  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)))
      FusedBiHeapifySiftFromMinToMaxNoDHA(V, N, pos_hc, N - 1, F_first_hc, F_last_hc)
    else
      SiftUpMinHeap(V, N, pos_hc, 0)
  else //Then it's in the pure max heap and not in the min heap.
    if (pos_mc == 0 or *(V + Flip(Parent(pos_mc))) >= pos_value)
      FusedBiHeapifySiftFromMaxToMinNoDHA(V, N, pos_mc, 0, F_first_hc, F_last_hc)
    else
      SiftUpMaxHeap(V, N, pos_mc, 0)
  return
```

Part 7. Lambdas

14. 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 rewritting 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.

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

For instance,

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

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

```
LambdaType F(int N)
  let addition_lambda = lambda(int i) -> int {
    return N + i
  }
  return addition_lambda

lambda_one = F(1)
lambda_two = F(2)
```

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

15. 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:

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

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(ParamterList ...) that we've defined, we add an additional parameter, LambdaType lambda, to the end of the parameter list: F(ParamterList ..., LambdaType lambda). For instance, BiHeapify (Node V, int N) becomes BiHeapify (Node V, int N, LambdaType lambda).
(2) For any function call F(Args ...), , 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

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

then the call BiHeapify (V, N, trivial_lambda) reduces down to the call 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 BiHeapify ():

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

```
BiHeapify(V, N, flip_lambda)
```

produces a "revered BiHeap" where the maximum is at node $V = V + 0 = V + \operatorname{Flip}(N-1)$ and the minimum is at node $V + (N-1) = V + \operatorname{Flip}(0)$. 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 8. 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.

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

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

Lemma 15.1. Suppose that $N > 0$ is even. Then $p \coloneqq \operatorname{ParentBiHeapSize}(N)$ is even, $p \mod 3 \neq 2$, 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 $p \mod 3 = \chi$ is either 0 or 1. The number of Out nodes that B_p has is exactly $p - |\operatorname{In}(B_p)| = p - \frac{p + 2(p \mod 3)}{3} = \frac{3N}{2} + \chi - \frac{(\frac{3N}{2} + \chi) + 2\chi}{3} = N$. ∎

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)
  let index_lambda = lambda(int N_local, int pos_hc) -> int {
    if (pos_hc < num_ele / 2)
      return 2 * pos_hc
    else
      return 2 * (num_ele - 1 - pos_hc) + 1 //= 2 * Flip(pos_hc) + 1
  }
  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 a pure min heap node with min heap cordinate i, then the value of v is stored at index $2i$ while the value of the node correspondong 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 constrast, 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 occationally 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 innefficiencies 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.
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).
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 ≤ last_F_hc then our fused BiHeap's fused nodes will be any node whose min heap coordinate, i, satisfies first_F_hc ≤ i ≤ 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()
  return V[0]
```

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

We will also be using the following helper functions.

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

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

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

15.1. Inserting an Element. Suppose that we wish to insert a sequence $a_0, a_1, \ldots$ 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 $O(\mathrm{N})$ 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 $O(\mathrm{N})$ operation.
(3) For the value a_2, repeat step (1) and for the value a_3 repeat step (2), and so forth where if $i \mod 2 = 0$ 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 $\mathrm{In}(\mathrm{N}) = \frac{\mathrm{N}+2(\mathrm{N} \mod 3)}{3}$ 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 $O(\mathrm{N})$ since the new parent BiHeap size, given by ParentBiHeapSize above, is not more than $\frac{3\mathrm{N}}{2} + 1$.

This function will only be called when $\mathrm{N} \geq 2$ 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).[1]

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

Once the parent BiHeap has been expanded, we then iterate the above process as many times are necessarily 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 perations performed by call_fused_biheapify_sift (resp. expand_parent_biheap) is not more than $C \log \mathrm{N}$ (resp. $D\,\mathrm{N}$). Then the sift operation will be called exactly $\mathrm{In}(\mathrm{N}) = \frac{\mathrm{N}+2(\mathrm{N} \mod 3)}{3}$ times while the expand_parent_biheap function will be called only once, leading to a total cost of no more than

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

where the ratio of this value to N is

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

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

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 appropiately, 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)
  if (num_elements_ < 2)
    F_first_hc = 1
    F_last_hc  = 1
    if (num_elements == 0)
      V[0] = value
    else if (num_elements == 1)
      if (value > V[0])
        V[1] = value
      else
        V[1] = V[0]
        V[0] = value
    num_elements = num_elements + 1
    return
  if (num_elements == N) //If we need to make room for the new value.
    expand_parent_biheap()
  int placement_node_hc
  //Determine which node to un-fuse and place the value there.
```

[1] This is why this function is called expand_parent_biheap and not expand_array_size.

```
  if (num_elements % 2 == 0) //Then "un-fuse" node F_first_hc to place the value there.
    placement_node_hc = F_first_hc
    F_first_hc        = F_first_hc + 1
  else //"un-fuse" node F_last_hc to place the value there.
    placement_node_hc = F_last_hc
    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
                          // the value of node placement_node_hc.
  num_elements = num_elements + 1
  call_fused_biheapify_sift(placement_node_hc)
  return
```

15.2. 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 an 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 appropiately, 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 $O(\mathrm{N})$ complexity is so similar to the argument showing that insert has amortized $O(\mathrm{N})$ complexity that it is omitted.

```
void PopMinOrMax(int pop_index)
  if (num_elements <= 2)
    N         = 2
    F_last_hc = 1
    if (num_elements == 2)
      if (pop_index == 0)
        SwapValues(V[0], V[1]) //Swap the min and max.
    num_elements = num_elements - 1
    F_first_hc   = num_elements
    return
  int heap_size = HeapSize(N)
  int first_in_node = N - heap_size
  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
    F_last_hc    = F_first_hc
  else
    if (num_elements % 2 == 1) //If we should fuse F_first_hc.
      F_first_hc--
    else
      F_last_hc++
  num_elements--
  SizeType pop_node_hc = 0
  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.
  SwapValues(V[pop_index], V[num_elements])
  call_fused_biheapify_sift(pop_node_hc) //Sift the element into place.
  return
```

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

```
void popmax()
```

```
  PopMinOrMax(1)
  return
```

```
void popmin()
  PopMinOrMax(0)
  return
```

Part 9. Appendix

16. $O(\mathrm{N})$ Functions with Multiple Recursive Calls.

Lemma 16.1. Let $m > 0$ be an integer. Let $L(i, \mathrm{N}, \mathrm{Args})$ be an $O(\mathrm{N})$ function taking as input a non-negative integer $i < m$ and a positive integer N along with some (possibly empty) finite list of arguments, collectively denoted by Args, each of which uses $O(\text{constant})$ memory and suppose that its return value, $(\mathrm{N}', \mathrm{Args}')$, along with any non-negative integer $i' < m$ may be passed as inputs to L. In addition, assume that N' is dependent solely on the value of N and that if L accepts $(i_0, \mathrm{N}_0, \mathrm{Args}_0)$ as input then it also accepts $(i, \mathrm{N}_0, \mathrm{Args}_0)$ as input for all integers $0 \leq i < m$.

Let $C > 1$ be such that for all $\mathrm{N} > 0$, any call to $L(i, \mathrm{N}, \mathrm{Args})$ performs no more than $C\,\mathrm{N}$ total operations for all possible inputs to i and Args. Suppose that $r < \frac{1}{m}$ is a positive real number such that for all possible inputs $(i, \mathrm{N}, \mathrm{Args})$ to L, if $(\mathrm{N}', \mathrm{Args}') = L(i, \mathrm{N}, \mathrm{Args})$ then $\mathrm{N}' \leq r\,\mathrm{N}$. Define the following function F, which takes as input any pair $(\mathrm{N}, \mathrm{Args})$ that may be forwarded as the last two inputs to L:

```
void F(N, Args)
  if (N < 2)
    return
  int i = 0
  while (i < m)
    F(L(i, N, Args))
    i = i + 1
  return
```

Then F is an $O(\mathrm{N})$ function that uses at most $O(\log \mathrm{N})$ additional memory.

Remark 16.2. Note that the assumption $r < \frac{1}{m}$ is crucial for the conclusion that F has $O(\mathrm{N})$ complexity. Indeed, if we were to use $m = 2$ but there was no $r < \frac{1}{2}$ satisfying this lemma's hypotheses, then, as with the merge sort algorithm for example, the above function could instead have worst case complexity no better than $O(\mathrm{N} \log \mathrm{N})$.

Proof. Note that if $m = 1$ then F is nothing more than a typical simple recursive function whose body contains only one recursive call, in which case the proof that F is $O(\mathrm{N})$ in this trivial case is well known and immediate. So we may assume that $m > 1$.

To aid in the exposition of the proof, we assume that we have a global variable current_recursive_depth storing a non-negative integer, initially set to 0, and we define a function G by

```
void G(N, Args)
  current_recursive_depth = current_recursive_depth + 1
  if (N < 2)
    current_recursive_depth = current_recursive_depth - 1
    return
  int i = 0
  while (i < m)
    (newN, newArgs) = L(i, N, Args)
    G(newN, newArgs)
    current_level = depth
    i = i + 1
  current_recursive_depth = current_recursive_depth - 1
```

```
    return
}
```

Note that if one were to ignore the assignments to the variable current_recursive_depth and the addition and subtractions by 1 operations then $G(\mathrm{N}, \mathrm{Args})$ would perform the exact same operations as $F(\mathrm{N}, \mathrm{Args})$ so to prove the conclusion about F, it suffices to prove the same conclusion about G while not counting the aforementioned operations in the total operation count.

Note that whenever a call to $L(i, \mathrm{N}, \mathrm{Args})$ is made, the value of current_recursive_depth does not change between when the call to L is first made and when execution leaves L. We may thus unambiguously associate with each call to L the value stored in current_recursive_depth when the call is first made, which we will refer to as the call's *depth*. Similarly, we will refer to the value stored in current_recursive_depth immediately before a call to G is made (i.e. the value of current_recursive_depth immediately *before* the first instruction in this same call to G has a chance to change the value of current_recursive_depth) the call's *depth*. If either a call to L or to G has depth d then we may also refer to it as a call with current_recursive_depth = d.

Assume that a call to $G(\mathrm{N}_0, \mathrm{Args}_0)$ has just been made. We will now count the total number of operation that it performs. We begin by noting that our assumptions imply that there are no more than

(1) m^1 calls to G when current_recursive_depth = 1,
(2) m^2 calls to G when current_recursive_depth = 2,

$$\vdots$$

(3) m^d calls to G when current_recursive_depth = d.

Similarly there are no more than m^d calls to L when current_recursive_depth = d.

Note that the total number of operations performed by all calls to L whose depth is 1, denote it by L_1, is not more than $mC\,\mathrm{N}_0$ and that the value of newN that is returned by any such call to L satisfies $\mathrm{newN} \leq r\,\mathrm{N}_0$. Suppose that we shown that the total number of operations performed by all calls to L whose depth is $d \geq 1$ is not more than $m^d C r^{d-1}\,\mathrm{N}_0$ and that the value of newN that is returned by any such call to L satisfies $\mathrm{newN} \leq r^d\,\mathrm{N}_0$. Any call to L, say $(\mathrm{newN}, \mathrm{newArgs}) = L(i, \mathrm{N}, \mathrm{Args})$, when current_recursive_depth = $d+1$ will return a value of newN satisfying $\mathrm{newN} \leq r\,\mathrm{N}$ and the value of N that was passed to this call of L would (by the inductive hypothesis) necessarily satisfy $\mathrm{N} \leq r^d\,\mathrm{N}_0$. Thus, this single call to L performed no more than $Cr^d\,\mathrm{N}_0$ operations and $\mathrm{newN} \leq r\,\mathrm{N} \leq r^{d+1}\,\mathrm{N}_0$. Since there are no more than m^{d+1} calls to L when current_recursive_depth = $d+1$, it follows that the total number of operations performed by all calls to L whose depth is $d+1$ is not more than $m^{d+1}Cr^d\,\mathrm{N}_0$.

We've thus proved that the total number of operations performed by all calls to L whose depth is $d \geq 1$, denote this by L_d, is not more than $m^d C r^{d-1}\,\mathrm{N}_0$. Of course, no calls to L with depth 0 are ever made so let $L_0 = 0$. Note that since $mr < 1$,

$$\sum_{d=1}^{\infty} L_d \leq C\,\mathrm{N}_0\, m \sum_{d=1}^{\infty} (mr)^{d-1} = C \frac{m}{1-mr}\,\mathrm{N}_0 < \infty$$

Since the value of newN passed to any call to G of depth d satisfies $\mathrm{newN} \leq r^d\,\mathrm{N}_0$ and since execution calls the return statement if the value of N passed to G is less than 2, it follows from the inequality $r^d\,\mathrm{N}_0 \leq 1$ that the largest value ever stored in current_recursive_depth is not more than $D := 1 + \left\lceil \log_r \left(\frac{1}{\mathrm{N}_0}\right) \right\rceil = 1 + \left\lceil \log_{1/r} (\mathrm{N}_0) \right\rceil$. Thus our initial call to G uses at most $O(\log \mathrm{N})$ additional memory. It also follows that the condition in line 3 is checked no more than $m + m^2 + \cdots + m^D$ times, where this equals $-1 + \frac{m^{D+1}-1}{m-1}$ if $m > 1$ while it equals $D \leq \mathrm{N}_0$ otherwise. Assuming that $m > 1$, note that

$$m^{D+1} = m^2 m^{\lceil \log_{1/r}(\mathrm{N}_0) \rceil} \leq m^3 m^{\log_{1/r}(\mathrm{N}_0)} = m^3\,\mathrm{N}_0^{\overline{\frac{1}{\log_m\left(\frac{1}{r}\right)}}}$$

where since $\frac{1}{r} > m$ gives us $\log_m\left(\frac{1}{r}\right) > 1$, it follows that $m^3\,\mathrm{N}_0^{\overline{\frac{1}{\log_m\left(\frac{1}{r}\right)}}} < m^3\,\mathrm{N}_0$. Thus the condition in line 3 is checked no more than $\frac{m^3}{m-1}\,\mathrm{N}_0$ times.

Note that the number of times that line 7 is executed is necessarily strictly less than the number of times that the condition in line 3 is checked. The number of times that line 12 is executed, which is 1 larger than the number of times that the condition in line 8 is checked, is necessarily strictly less than $m+1$ multiplied by the number of times that the condition in line 3 is checked This means that if we let $B=(3+2m)\frac{m^3}{m-1}$ then the operations in lines 3, 7, 8, and 12 are performed no more than $B\,\mathrm{N}_0$ times. Thus, this functions performs a grand total of no more than $C\left[\frac{m}{1-mr}+B\right]\mathrm{N}_0$ operations. ∎

Corollary 16.3. Let $A>1$ be an integer. Suppose that we weaken the hypotheses of lemma 16.4 by replacing the inequality $\mathrm{N}'\leq r\,\mathrm{N}$ with the inequality $\mathrm{N}'\leq r\,\mathrm{N}+A$ and we replace the number 2 the condition "if$(\mathrm{N}<2)$" in F's definition with a constant $T>1+\left\lceil\frac{A}{1-r}\right\rceil$ so that F is now:

```
void F(N, Args)
  if (N < T)
    return
  int i = 0
  while (i < m)
    F(L(i, N, Args))
    i = i + 1
  return
```

Then F is an $O(\mathrm{N})$ function that uses at most $O(\log \mathrm{N})$ additional memory.

Proof. We may assume without loss of generality that a, b, and m are all strictly greater than 1. By sufficiently increasing C if necessary, we may assume without loss of generality that $C>1+\frac{A}{1-r}$ is such that for all $N>0$, any call to $L(\mathrm{N},\mathrm{Args})$ performs no more than $C\,\mathrm{N}-\frac{A}{1-r}$ total operations for all possible inputs to i and Args. Continuing with the terminology defined in the proof of lemma 16.4, note that if $(\mathrm{newN},\mathrm{newArgs})=L(i,\mathrm{N},\mathrm{Args})$ is called while current_recursive_depth $=d$ then we are now guaranteed that $\mathrm{newN}\leq r^d\,\mathrm{N}_0+A\left(1+\cdots+r^d\right)=r^d\,\mathrm{N}_0+A\frac{1-r^{d+1}}{1-r}$ instead of $\mathrm{newN}\leq r^d\,\mathrm{N}_0$ as in the original proof. Note that there exists an integer $D>1$ such that $r^{D-1}\,\mathrm{N}_0<1+\left(\left\lceil\frac{A}{1-r}\right\rceil-\frac{A}{1-r}\right)$, so that $\mathrm{newN}<r^{D-1}\,\mathrm{N}_0+\frac{A}{1-r}<1+\left\lceil\frac{A}{1-r}\right\rceil<T$, which implies that current_recursive_depth is never greater than D so that in particular, our initial call to uses at most $O(\log\mathrm{N})$ additional memory. For concreteness, we will take D to be $D:=1+\left\lceil\log_{1/r}\left(\frac{(1-r)\,\mathrm{N}_0}{rA}\right)\right\rceil$, where one derives this formula in a similar way to how the formula for D was derived in the proof of lemma 16.4. Note that the total number of operations performed by all calls to L whose depth is $d\geq 1$, denote this by L_d, is not more than $m^d\left[Cr^{d-1}\,\mathrm{N}_0-\frac{A}{1-r}\right]\leq Cm^dr^{d-1}\,\mathrm{N}_0$. As before, we may now conclude that $\sum_{d=1}^{\infty}L_d<\infty$.

Let $S=\frac{rA}{1-r}$. Recall that the condition in line 2 is checked no more than $m+m^2+\cdots+m^D$ times, where since $m>1$, this equals $-1+\frac{m^{D+1}-1}{m-1}$. Note that

$$m^{D+1}=m^2m^{\left\lceil\log_{1/r}(\mathrm{N}_0/S)\right\rceil}\leq m^3m^{\log_{1/r}(\mathrm{N}_0/S)}=m^3\left(\mathrm{N}_0/S\right)^{\frac{1}{\log_m\left(\frac{1}{r}\right)}}$$

where $\log_m\left(\frac{1}{r}\right)>1$ now gives us $m^3\left(\mathrm{N}_0/S\right)^{\frac{1}{\log_m\left(\frac{1}{r}\right)}}<m^3\left(1/S\right)^{\frac{1}{\log_m\left(\frac{1}{r}\right)}}\mathrm{N}_0$. Thus the condition in line 2 is checked no more than $\frac{m^3}{m-1}\left(1/S\right)^{\frac{1}{\log_m\left(\frac{1}{r}\right)}}\mathrm{N}_0$ times. The rest of the proof proceeds as in the final part of the proof of lemma 16.4. ∎

Lemma 16.4. Let $m>0$ be an integer and let $T>1$ be an integer. Let $L(i,\mathrm{N},\mathrm{Args})$ be an $O(\mathrm{N})$ function taking as input a non-negative integer $i<m$ and a positive integer N along with some (possibly empty) finite list of arguments, collectively denoted by Args, each of which uses $O(\text{constant})$ memory and suppose that its return value, $\left(\mathrm{N}',\mathrm{Args}'\right)$, along with any non-negative integer $i'<m$ may be passed as inputs to L. In addition, assume that N' is dependent solely on the values of N and i and that if L accepts $(i_0,\mathrm{N}_0,\mathrm{Args}_0)$ as input then it also accepts $(i,\mathrm{N}_0,\mathrm{Args}_0)$ as input for all integers $0\leq i<m$.

Let $C>\frac{m}{r}(3m+1)+1$ be such that for all $\mathrm{N}>0$, any call to $L(i,\mathrm{N},\mathrm{Args})$ performs no more than $\left(\frac{rC}{m}-(3m+1)\right)\mathrm{N}$ total operations for all possible inputs to i and Args. Let $r_0,\ldots,r_{m-1}$ be distinct positive real numbers summing to $r = \sum_{l=0}^{m-1} r_l < 1$ such that for all N and Args that form a possible input $(i, \mathrm{N}, \mathrm{Args})$ into L, if $(\mathrm{N}', \mathrm{Args}') = L(i, \mathrm{N}, \mathrm{Args})$ then $\mathrm{N}' \leq r_i \mathrm{N}$.

Define the following function F, which takes as input any pair $(\mathrm{N}, \mathrm{Args})$ that may be forwarded as the last two inputs to L:

```
void F(N, Args)
  if (N <= T)
    return
  int i = 0
  while (i < m)
    F(L(i, N, Args))
    i = i + 1
  return
```

Then F is an $O(\mathrm{N})$ function that uses at most $O(\log \mathrm{N})$ additional memory.

Proof. Since the case of $m = 1$ is trivial, assume that $m > 1$. To aid in the exposition of the proof, we assume that we have a global variable current_recursive_depth storing a non-negative integer, initially set to 0, and we define a function G by

```
void G(N, Args)
  current_recursive_depth = current_recursive_depth + 1
  if (N < 2)
    current_recursive_depth = current_recursive_depth - 1
    return
  int i = 0
  while (i < m)
    (newN_i, newArgs_i) = L(i, N, Args)
    G(newN_i, newArgs_i)
    current_level = depth
    i = i + 1
  current_recursive_depth = current_recursive_depth - 1
  return
```

We will continue using, without repeating their meaning, the terminology and conventions that were used in the proof of lemma 16.4 where as in that lemma, we note that to prove the conclusion about F, it suffices to prove the same conclusion about G while ignoring all operations on current_recursive_depth when calculating the total operation cost.

We now introduce some additional terminology. Suppose that a call is made to G and execution has reached the while loop in G. By *a child call* (of this call) we mean any one of the subsequent m calls that are necessarily made within this call's while loop and we will refer to this call as the child's *parent call*. If a call to G has a parent call, then we will refer to this parent as the call's *first ancestor (call)*. Continuing by induction, having defined the n^{th} ancestor of a call to G, if this has itself a parent then we will refer to this parent as the $(n+1)^{th}$ *ancestor (call)*. If one call to G is the n^{th} ancestor of another call to G, then we will refer to the latter call as an n^{th} *descendant* of the former call. Given any input $(\mathrm{N}, \mathrm{Args})$ for G, by *the call tree depth of the call* $G(\mathrm{N}, \mathrm{Args})$ we mean one more than the largest integer n (if it exists, which we will show it does) such that this call has an n^{th} descendant. Less rigorously, the call tree depth of $G(\mathrm{N}, \mathrm{Args})$ is the largest number local variables that that are ever placed on the stack as a result of this call, where we assume that the initial call to G places variables on the stack so that if the initial call to $G(\mathrm{N}, \mathrm{Args})$ immediately returns (due to the if condition evaluating as true), then this call would have a call tree depth of 1 and otherwise, its call tree depth would be greater than 1.

Suppose that the initial call $G(\mathrm{N}, \mathrm{Args})$ has a call tree depth of 2. This is only possible if for each $i = 0, \ldots, m-1$, upon recursing into a call $G(\mathrm{N}_i, \mathrm{Args}_i)$, the if statement is immediately evaluated as true. The if statement is evaluated once when the current_recursive_depth = 1 and m times when current_recursive_depth = 2. The counter i is incremented m times and the while loop condition is also checked m times so that these three lines of code contribute $3m+1$ operations. For each $i = 0, \ldots, m-1$, $L(\mathrm{N}, \mathrm{Args})$ performs no more than $\left(\frac{rC}{m} - (3m+1)\right)\mathrm{N}$ operations so that the total number of operations performed is no greater than $3m+1+\sum_{i=0}^{m-1}\left(\frac{rC}{m} - (3m+1)\right)\mathrm{N} \leq \sum_{i=0}^{m-1}\frac{rC}{m}\mathrm{N} \leq rC\,\mathrm{N}$.

We proceed by induction on the call tree depth, where we have already proved the case $D = 2$. Suppose that we've shown that whenever $(\mathrm{N}, \mathrm{Args})$ is an input such that $G(\mathrm{N}, \mathrm{Args})$ has call tree depth $d \leq D$, then the number of operations performed by $G(\mathrm{N}, \mathrm{Args})$ is no greater than $C\left(r + \cdots + r^{d-1}\right)\mathrm{N}$. Let $G(\mathrm{N}, \mathrm{Args})$ be an input such that $G(\mathrm{N}, \mathrm{Args})$ has a call tree depth equal to $D+1$. There are m calls made to L when current_recursive_depth = 1, which perform no more than $\left(\frac{rC}{m} - (3m+1)\right)\mathrm{N}$ operations each so that as before, these L's together with the total number of calls to the if statement when current_recursive_depth = 1 and 2 together with the m check of the while loop and m increments of i while current_recursive_depth = 1, contribute no more than $rC\,\mathrm{N}$ operations to the total.

For each $i = 0, \ldots, m-1$, the total number of operations performed by the call $G(\mathrm{newN}_i, \mathrm{newArgs_i})$ (called when current_recursive_depth = 1) is no more than $C\left(r + \cdots + r^{D-1}\right)r_i\,\mathrm{N}$. Note that for since $r = \sum_{l=0}^{m-1} r_l$, for any positive integer p

$$\sum_{i=0}^{m-1} r^p r_i = \sum_{i=0}^{m-1}\left(\sum_{l=0}^{m-1} r_l\right)^p r_i = \left(\sum_{l=0}^{m-1} r_l\right)^{p+1} = r^{p+1}.$$

It follows that these calls to G perform no more than $C\left(r^2 + \cdots + r^D\right)\mathrm{N}$ operations. Thus the total number of operations performed us $C\left(r + \cdots + r^D\right)\mathrm{N}$, as desired.

We've thus shown that if $G(\mathrm{N}, \mathrm{Args})$ is a call whose call tree depth is finite, then this call performs no more than $C\left(r + \cdots + r^D\right)\mathrm{N} \leq \frac{C}{1-r}\mathrm{N}$ operations. It remains to show that every call $G(\mathrm{N}, \mathrm{Args})$ has a finite tree depth, which is equivalent to showing that every such call halts.

Note the value of newN_i that is returned by any call to L when current_recursive_depth = 1 such call to L satisfies $\mathrm{newN}_i \leq r_i \mathrm{N}_0 \leq R\,\mathrm{N}_0$. Assume that we've shown that for all $k = 1, \ldots, d$, the value of newN_i that is returned by a call to $L(i, \mathrm{N}, \mathrm{Args})$ when current_recursive_depth = k satisfies $\mathrm{newN}_i \leq R^k\,\mathrm{N}_0$. When current_recursive_depth = $k+1$, for any $i = 0, \ldots, m-1$, the value of N that is passed to $L(i, \mathrm{N}, \mathrm{Args})$ must, by the inductive hypothesis,satisfy $\mathrm{N} \leq R^d\,\mathrm{N}_0$ so that $\mathrm{newN}_i \leq r\,\mathrm{N} \leq rR^d\,\mathrm{N}_0 \leq R^{d+1}\,\mathrm{N}_0$. We've thus shown that for all positive integers d, the value of newN_i that is returned by a call to $L(i, \mathrm{N}, \mathrm{Args})$ when current_recursive_depth = d satisfies $\mathrm{newN}_i \leq R^k\,\mathrm{N}_0$.

A call to L, say $(\mathrm{newN}, \mathrm{newArgs}) = L(i, \mathrm{N}, \mathrm{Args})$, when current_recursive_depth = $d+1$ will return a value of newN satisfying $\mathrm{newN} \leq r_i\,\mathrm{N}$ and the value of N that was passed to this call of L would (by the inductive hypothesis) necessarily satisfy $\mathrm{N} \leq r^d\,\mathrm{N}_0$. Let j_0 be any integer such that r_{j_0} is the maximum of $r_0, \ldots, r_{m-1}$ and let $R = r_{j_0}$.

Let D be the smallest integer greater than 1 such that $R^{D-1}\,\mathrm{N}_0 < T$, so that $D = 1 + \left\lceil \log_R\left(\frac{T}{\mathrm{N}_0}\right)\right\rceil = 1 + \left\lceil \log_{1/R}\left(\frac{\mathrm{N}_0}{T}\right)\right\rceil$. Observe that if execution enters $G(\mathrm{N}, \mathrm{Args})$ when current_recursive_depth = D then the condition in the if statement is necessarily satisfied so that execution the leaves the function. In particular, current_recursive_depth never stores a value greater than D_R and the call to $G(\mathrm{N}, \mathrm{Args})$ halts.

■

17. FusedBiHeapify **With** N mod 3 = 2**.** Let us continue with the notation from the section defining FusedBiHeapify and assume that N mod 3 = 2. In this case, the BiHeap $\mathrm{B_N}$ has a double headed arrow where none, one, or both of this edge's nodes could be in F. We must consider each case.

```
void FusedBiHeapifySiftFromMinToMaxWithDHA(Node V, int N, int pos_hc,
                                          int last_node_in_biheap_hc, int first_F_hc, int last_F_hc)
  int heap_size                  = HeapSize(N)
  int first_in_node              = N - heap_size
  int pmin_double_arrow_end_hc = (N - 2) / 3 //The double arrow node in the pure min heap ...
  int pmax_double_arrow_end_hc = 2 * pmin_double_arrow_end_hc + 1 //... and the pure max heap.
  int minh_parent_of_pmin_double_arrow_end_hc = Parent(pmin_double_arrow_end_hc)
  int maxh_parent_of_pmax_double_arrow_end_hc = Flip(minh_parent_of_pmin_double_arrow_end_hc)
  if (first_F_hc <= pos_hc and pos_hc <= last_F_hc) //This can happen with a double arrow node.
    return
  while (pos_hc < first_in_node)
    int  left_child_hc  = LeftChild(pos_hc)
    int  right_child_hc = RightChild(pos_hc)
    bool is_right_valid = false
    if (right_child_hc <= last_node_in_biheap_hc and right_child_hc < heap_size)
      is_right_valid = true
    bool is_left_valid = false
    if (left_child_hc <= last_node_in_biheap_hc and left_child_hc < heap_size)
      is_left_valid = true
    if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)//If the left child is in F
      left_child_hc = Flip(Parent(Flip(left_child_hc))) //then "skip over it."
      //If the left child is still in F then it must be the end of the double arrow that
      // belongs to the pure max heap so skip over it.
      if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)
        left_child_hc = maxh_parent_of_pmax_double_arrow_end_hc
    if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
      right_child_hc = Flip(Parent(Flip(right_child_hc)))
      if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
        right_child_hc = maxh_parent_of_pmax_double_arrow_end_hc
    if (is_right_valid and right_child_hc <= last_node_in_biheap_hc) is_right_valid = true
    else                                                               is_right_valid = false
    if (is_left_valid and left_child_hc <= last_node_in_biheap_hc) is_left_valid  = true
    else                                                             is_left_valid  = false
    if (!is_left_valid and !is_right_valid)
      return
    Node pos_node         = V + pos_hc
    Node left_child_node  = V + left_child_hc
    Node right_child_node = V + right_child_hc
    Node smaller_node
    if (!is_left_valid or (is_right_valid and *right_child_node < *left_child_node))
      smaller_node = right_child_node
      pos_hc       = right_child_hc
    else //Here, the left child is valid.
      smaller_node = left_child_node
      pos_hc       = left_child_hc
    if (*pos_node <= *smaller_node)
      return
    SwapValues(pos_node, smaller_node)
  //It's now impossible to be in F, in the pure max heap, and incident to the double arrow.
  if (pos_hc == pmin_double_arrow_end_hc)
    //If the other end of the double arrow is in F.
    if (first_F_hc <= pmax_double_arrow_end_hc and pmax_double_arrow_end_hc <= last_F_hc)
      Node maxh_parent_of_pmax_double_arrow_end_node = V + maxh_parent_of_pmax_double_arrow_end_hc
      Node pos_node = V +  pos_hc
      //Perform one iteration of sifting up the min heap while skipping the node in F.
      if (maxh_parent_of_pmax_double_arrow_end_hc <= last_node_in_biheap_hc and
          *maxh_parent_of_pmax_double_arrow_end_node < *pos_node)
        SwapValues(pos_node, maxh_parent_of_pmax_double_arrow_end_node)
      else return
      pos_hc = maxh_parent_of_pmax_double_arrow_end_hc
  SiftUpMaxHeap(V, N, Flip(pos_hc), Flip(last_node_in_biheap_hc))
  return
}
```

```
void FusedBiHeapifySiftFromMaxToMinWithDHA(Node V, int N, int pos_mc,
                              int first_node_in_biheap_hc, int first_F_hc, int last_F_hc)
  int pos_hc                   = Flip(pos_mc)
  int heap_size                = HeapSize(N)
  int first_in_node            = N - heap_size
  int pmin_double_arrow_end_hc = (N - 2) / 3
  int pmax_double_arrow_end_hc = 2 * pmin_double_arrow_end_hc + 1
  int minh_parent_of_pmin_double_arrow_end_hc = Parent(pmin_double_arrow_end_hc)
  if (first_F_hc <= pos_hc and pos_hc <= last_F_hc)
    return
  while (pos_mc < first_in_node)
    int  left_child_mc  = LeftChild(pos_mc)
    int  right_child_mc = RightChild(pos_mc)
    int  left_child_hc  = Flip(left_child_mc)
    int  right_child_hc = Flip(right_child_mc)
    bool is_right_valid = false
    if (right_child_mc < heap_size and right_child_hc >= first_node_in_biheap_hc)
      is_right_valid = true
    if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)//If the left child is in F
      left_child_hc = Parent(left_child_hc) //then "skip over it."
      //If the left child is still in F then it must be the end of the double arrow that
      // belongs to the pure min heap so skip over it.
      if (first_F_hc <= left_child_hc and left_child_hc <= last_F_hc)
        left_child_hc = minh_parent_of_pmin_double_arrow_end_hc
      left_child_mc = Flip(left_child_hc)
    if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
      right_child_hc = Parent(right_child_hc)
      if (first_F_hc <= right_child_hc and right_child_hc <= last_F_hc)
        right_child_hc = minh_parent_of_pmin_double_arrow_end_hc
      right_child_mc = Flip(right_child_hc)
    if (is_right_valid and right_child_hc >= first_node_in_biheap_hc) is_right_valid = true
    else                                                              is_right_valid = false
    bool is_left_valid = false
    if (left_child_hc >= first_node_in_biheap_hc) is_left_valid = true
    if (!is_left_valid and !is_right_valid)
      return
    Node pos_node         = V + pos_hc
    Node left_child_node  = V + left_child_hc
    Node right_child_node = V + right_child_hc
    Node larger_node
    if (!is_left_valid or (is_right_valid and *right_child_node > *left_child_node))
      larger_node = right_child_node
      pos_hc      = right_child_hc
      pos_mc      = right_child_mc
    else //Here, the left child is valid.
      larger_node = left_child_node
      pos_hc      = left_child_hc
      pos_mc      = left_child_mc
    if (*pos_node >=  *larger_node)
      return
    SwapValues(pos_node, larger_node)
  //It's now impossible to be in F, in the pure min heap, and incident to the double arrow.
  if (pos_mc == pmin_double_arrow_end_hc) //if and only if pos_hc = pmax_double_arrow_end_hc
    if (first_F_hc <= pmin_double_arrow_end_hc and pmin_double_arrow_end_hc <= last_F_hc)
      Node minh_parent_of_pmin_double_arrow_end_node = V + minh_parent_of_pmin_double_arrow_end_hc
      Node pos_node = V + pos_hc
      //Perform one iteration of sifting up the min heap while skipping the node in F.
      if (minh_parent_of_pmin_double_arrow_end_hc >= first_node_in_biheap_hc and
          *minh_parent_of_pmin_double_arrow_end_node > *pos_node)
        SwapValues(pos_node, minh_parent_of_pmin_double_arrow_end_node)
      else return
      pos_hc = minh_parent_of_pmin_double_arrow_end_hc
  SiftUpMinHeap(V, N, pos_hc, first_node_in_biheap_hc)
  return
```

```
void FusedBiHeapifyWithDHA(Node V, int N, int first_F_hc, int last_F_hc)
  if(N <= 2)
    if (N == 2 and last_F_hc < first_F_hc)
      Node V_0 = V + 0
      Node V_1 = V + 1
      if (*V_0 > *V_1) SwapValues(V_0, V_1)
    return
  int heap_size     = HeapSize(N)
  int first_in_node = N - heap_size
  if (first_F_hc > first_in_node and last_F_hc < heap_size - 1)
    //If we don't have to worry about skipping over either one of the end nodes of the double
    // headed arrow, then we may as well use the more efficient FusedBiHeapify algorithm.
    FusedBiHeapifyNoDHA(V, N, first_F_hc, last_F_hc)
    return
  if (first_F_hc < first_in_node) first_F_hc = first_in_node
  if (last_F_hc >= heap_size)     last_F_hc  = heap_size - 1 //The last In node.
  int last_node_in_biheap_hc  = heap_size - 2
  int first_node_in_biheap_hc = Flip(last_node_in_biheap_hc)
  while (first_node_in_biheap_hc > 0)
    first_node_in_biheap_hc--
    FusedBiHeapifySiftFromMinToMaxWithDHA(V, N, first_node_in_biheap_hc,
                                          last_node_in_biheap_hc, first_F_hc, last_F_hc)
    last_node_in_biheap_hc++
    FusedBiHeapifySiftFromMaxToMinWithDHA(V, N, Flip(last_node_in_biheap_hc),
                                         first_node_in_biheap_hc, first_F_hc, last_F_hc)
  return
```

18. Flip-Ordered BiHeaps.

Definition 18.1. Say that an ordered collection of nodes $\mathrm{V}, \ldots, \mathrm{V}+(\mathrm{N}-1)$ is *flip-ordered* if $^*(\mathrm{V}+i) \leq {}^*(\mathrm{V}+\mathrm{Flip}(i))$ for all $0 \leq i < \left\lfloor \frac{\mathrm{N}}{2} \right\rfloor$. Call this collection a *flip-ordered BiHeap* if it is both flip-ordered and a BiHeap.

The following algorithm forms a flip-ordered BiHeap from any collection of nodes. This algorithm is a modification of the BiHeapify () algorithm with code added to ensure that at every step of the biheapification process, the condition $^*(\mathrm{V}+i) \leq {}^*(\mathrm{V}+\mathrm{Flip}(i))$ is satisfied for all i such that $0 \leq i < \left\lfloor \frac{\mathrm{N}}{2} \right\rfloor$ with $\mathrm{V}+i$ is in the current BiHeap. It is straightforward to see that like the BiHeapify algorithm, this algorithm has $O(\mathrm{N})$ complexity and that it forms a flip-ordered BiHeap.

```
//Assumes that the node pos_hc belongs to the min heap and that
// pos_hc <= last_node_in_biheap_hc.
void SiftFromMinToMaxFlipOrdered(Node V, int N, int pos_hc, int last_node_in_biheap_hc)
  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.
  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
    Node flip_pos_node    = V + Flip(pos_hc)
    if (*pos_node > *flip_pos_node)
```

```
      SwapValues(pos_node, flip_pos_node)
    bool is_right_child_valid = right_child_hc <= last_node_in_biheap_hc and
                                right_child_hc < heap_size
    Node smaller_node
    if (is_right_child_valid and *right_child_node < *left_child_node)
      smaller_node = right_child_node
      pos_hc       = right_child_hc
    else
      smaller_node = left_child_node
      pos_hc       = left_child_hc
    if (*pos_node <= *smaller_node)
      return
    SwapValues(pos_node, smaller_node)
  if (N % 3 != 2 or pos_hc != (N - 2) / 3) //If the node is not the end of a double arrow.
    Node pos_node         = V + pos_hc
    Node flip_of_pos_node = V + FLIP(pos_hc)
    if ((pos_hc <  N / 2 and *pos_node > *flip_of_pos_node) or
        (pos_hc >= N / 2 and *pos_node < *flip_of_pos_node))
      SwapValues(pos_node, flip_of_pos_node)
  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
// Flip(pos_mc) >= first_node_in_biheap_hc.
void SiftFromMaxToMinFlipOrdered(Node V, int N, int pos_mc, int first_node_in_biheap_hc)
  int heap_size                  = HeapSize(N)
  int first_node_in_mirror_heap = N - heap_size
  int pos_hc                     = Flip(pos_mc)
  //Sift down the max heap while not yet in the min heap.
  while (pos_mc < first_node_in_mirror_heap)
    int  left_child_mc    = LeftChild(pos_mc)
    int  right_child_mc   = RightChild(pos_mc)
    int  left_child_hc    = Flip(left_child_mc)
    int  right_child_hc   = Flip(right_child_mc)
    Node pos_node         = V + pos_hc
    Node left_child_node  = V + left_child_hc
    Node right_child_node = V + right_child_hc
    Node flip_pos_node    = V + pos_mc
    if (*pos_node < *flip_pos_node)
      SwapValues(pos_node, flip_pos_node)
    //Note that right_child_hc >= first_node_in_biheap_hc necessarily holds.
    bool is_right_child_valid = right_child_mc < heap_size
    Node larger_node
    if (is_right_child_valid and *right_child_node > *left_child_node)
      larger_node = right_child_node
      pos_hc      = right_child_hc
      pos_mc      = right_child_mc
    else
      larger_node = left_child_node
      pos_hc      = left_child_hc
      pos_mc      = left_child_mc
    if (*pos_node >= *larger_node)
      return
    SwapValues(pos_node, larger_node)
  if (N % 3 != 2 or pos_mc != (N - 2) / 3) //If the node is not the end of a double arrow.
    Node pos_node         = V + pos_hc
    Node flip_of_pos_node = V + pos_mc
    if ((pos_hc <  N / 2 and *pos_node > *flip_of_pos_node) or
        (pos_hc >= N / 2 and *pos_node < *flip_of_pos_node))
        SwapValues(pos_node, flip_of_pos_node)
39    SiftUpMinHeap(V, pos_hc, first_node_in_biheap_hc)
      return
```

```
    void BiHeapifyFlipOrdered(Node V, int N)
2     if (N < 2)
        return
4     int heap_size      = HeapSize(N)
      int first_in_node = N - heap_size
6     for (int i = first_in_node i < N / 2 i++)
        Node i_node       = V + i
8       Node flip_i_node = V + Flip(i)
        if (*i_node > *flip_i_node)
10        SwapValues(i_node, flip_i_node)
      //Ignore all In nodes, even those belonging to a doubled-headed arrow.
12    int last_node_in_biheap_hc  = MinHeapCoordinateOfLastMinHeapNode(heap_size)
      int first_node_in_biheap_hc = Flip(last_node_in_biheap_hc)
14    while (first_node_in_biheap_hc > 0)
        first_node_in_biheap_hc-- //Sift the next pure min heap node.
16      Node first_node_in_biheap_node = V + first_node_in_biheap_hc
        Node last_node_in_biheap_node  = V + first_node_in_biheap_hc
18      if (*first_node_in_biheap_node > *last_node_in_biheap_node)
          SwapValues(first_node_in_biheap_node, last_node_in_biheap_node)
20      SiftFromMinToMaxFlipOrdered(V, N, first_node_in_biheap_hc, last_node_in_biheap_hc)
        last_node_in_biheap_hc++ //Sift the next pure max heap node.
22      SiftFromMaxToMinFlipOrdered(V, N, Flip(last_node_in_biheap_hc), first_node_in_biheap_hc)
      return
```

Of course, the algorithm produces a flip-ordered BiHeap (def. 18.1). Note that calling either BiHeapify () or BiHeapifyFlipOrdered () to a list of nodes that have just been passed as input to BiHeapifyFlipOrdered () does not change the value of any of the nodes. In particular, like the BiHeapify algorithm, the BiHeapifyFlipOrdered algorithm is idempotent, which makes this variant of the BiHeapify algorithm noteworthy and potentially important. However, the only benefit that the author has found for this algorithm is to form BiHeaps that have slightly nicer mathematical properties than those that are given by the BiHeapify algorithm.

Note that if one were to add between lines 5 and 6 of the BiHeapifyFlipOrdered algorithm the code

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

to recurse on the In nodes, then BiHeapifyFlipOrdered would become an $O(\mathrm{N})$ recursive algorithm that would no longer be idempotent. However, in this case one could then remove the for loop and still have an algorithm that produces a flip-ordered BiHeap.

REFERENCES

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