

Fibonacci vs. Binary Heaps: A Performance Comparison

Authors: Miller Fourie, Owen Zgurski, Shunsuke Kobayashi

1. Introduction

Priority queues are important in many algorithmic applications – from graph traversal, shortest-path computations to scheduling and simulation. Among the various structures used to implement priority queues, in this paper we will compare Binary Heaps and Fibonacci Heaps. Binary Heaps with a array-based implementation, offer $O(\log n)$ operations for insertion, extraction, and decrease-key. In contrast, Fibonacci Heaps have a amortized constant-time insertions and decrease-key operations, along with constant-time merge capabilities, making them attractive for algorithms such as Dijkstra's, where frequent key updates are common.

However, theoretical complexity does not always translate directly into practical performance. Implementation details and language-specific overhead can greatly influence actual performance. In this paper, we present a comprehensive empirical comparison between Binary Heap Sort and Fibonacci Heap Sort. Our evaluation spans both isolated operations (insertion, extraction, decrease-key, and merge) and integrated applications (specifically Dijkstra's Algorithm), using a rigorously designed benchmarking framework implemented in Python. Testing was performed on a text-based Linux virtual machine to minimize background overhead, and unnecessary services, enforced further by CPU pinning to a single core.

The experimental results reveal a fundamental trade-off between time and space efficiency. Fibonacci Heaps deliver superior runtime performance in operations that are critical to Dijkstra's algorithm, particularly in scenarios with frequent key adjustments and merges. However, this comes at the cost of increased memory usage due to their pointer-based, doubly linked structure. Conversely, our optimized Binary Heap—designed to use a lean, list-based position mapping and memory-efficient data objects—consistently consumes less memory, though its runtime performance is comparatively slower in key-intensive workloads.

By bridging the gap between theoretical analysis and practical performance, this study not only validates the expected asymptotic behavior of both data structures but also provides valuable insights into their real-world trade-offs. Ultimately, our work aims to guide practitioners in selecting the most appropriate heap implementation based on the specific demands of their applications.

2. Experimental Design

2.1 How We Compare the Graphs

To compare the performance of Binary Heaps and Fibonacci Heaps, we designed a benchmarking framework implemented in Python. Each heap implementation was tested independently across multiple core operations: insert, extract, decrease_key, and merge. Additionally, we benchmarked the performance of each heap when used as the priority queue in Dijkstra's algorithm.

For each operation, we generated a randomized input data using a fixed seed per trial to ensure repeatability. Each input size was tested across multiple runs (default: 5 runs per size), and timing were averaged to reduce variant caused by transient system effects. The heap operations were executed using custom benchmark functions that measured total elapsed time using Python's `time.perf_counter()` for high-precision timing.

To prevent garbage collection overhead from interfering with measurements, we explicitly triggered garbage collection with `gc.collect()` before timing and temporarily disabled it using `gc.disable()` during execution. For memory benchmarks, Python's `tracemalloc` module was used to record peak memory usage after inserting all elements into the heap.

For each operation:

- **Insert** benchmarks measured the time to insert a list of randomly generated integers into a empty heap.
- **Extract** benchmarks measured the time to repeatedly extract the minimum element from a pre-filled heap.
- **Decrease-Key** benchmarks generated random keys and performed decrements on each in-place key using either internal node handles (Fibonacci) or value-based mapping (Binary)
- **Merge** benchmarks measured the time to combine two separate heaps of equal size.

This benchmarking setup ensures consistency and fairness in our testing.

2.2 Performance Metrics

To evaluate and compare the performance of Binary Heaps and Fibonacci Heaps, we employed a set of metrics designed to capture both theoretical alignment and practical efficiency. Each metric was chosen to isolate a specific aspect of heap behavior and support empirical validation of known asymptotic bounds.

1. Total Runtime

For each core operation – insert, extract, decrease_key, and merge – we measured the total execution time needed to perform the operation across an input size of n . This metric captures the real-world performance cost of executing a full sequence of operations, providing insight into constant factors and implementation overheads beyond asymptotic complexity.

2. Average Time per Operation

To evaluate how the time cost of individual operations scales with input, we processed the data to get the average time per operation:

3. Operation Analysis

3.1 Insertion Analysis

Explanation of Functions

Before diving into the details of insertion, let's first break down the stages of both insertion heap methods.

The insertion operation in a Binary Heap begins by adding the new element to the end of the underlying array, which represents a complete binary tree. This placement maintains the structural integrity of the heap since the tree must remain complete. This placement maintains the structural integrity of the heap since the tree must remain complete. For instance, imagine we have a Binary Heap stored as an array in level order:

[2, 4, 6, 8, 10]

If we want to insert a new element, for this example, 1, it is first appended to the array, resulting in:

[2, 4, 6, 8, 10, 1]

At this point, although the tree is complete, the min-heap property may be violated since the new element might be smaller than its parent.

This leads us to the restoration phase, which we will refer to as heapify-up going forward. In this phase, the newly added element is compared with its parent. The parent of an element at index i is found at $\lfloor (i-1)/2 \rfloor$. If the new element is smaller a swap occurs. This comparison and swap process continues recursively until the new element is in a position where it is larger than or equal to its parent or it reaches the root. For clarity, we will walk through the example in a tabulated summary below.

Table 3.1.1: Heapify Process Example

Step	Heap Array	Index of New Element	Parent's Index	Values Compared	Swap Made	Notes
1	[2, 4, 6, 8, 10, 1]	5	2	1 vs 6	Yes	1 is swapped upward
2	[2, 4, 1, 8, 10, 6]	2	0	1 vs 2	Yes	1 becomes the new root
3	[1, 4, 2, 8, 10, 6]	0	-	-	-	Heapify complete

Next let's look at Fibonacci Heap insertion. Fibonacci Heap is very simplistic. Much like Binary Heap insertion, it begins by adding the new element to the collection; however, unlike Binary Heap, we don't heapify. Instead, the process is composed of two main phases: the addition of the new node to the root list and update the min pointer. For clarity, we walk through an example of this in a tabulated summary below:

Table 3.1.2: Fibonacci Insertion Example

Step	Heap Root List	Operation	Min Pointer
1	21→(5, 13)	Starting heap	5
2	21→(5, 13), 3	Insert new node 3	5
3	21→(5, 13), 3	Update min point (since 3<5)	3

Proof of Correctness

Proof of Correctness for Binary

The first line of the function,

```
self.heap.append(value)
```

adds a new element to the end of the heap's internal array. This maintains the shape property of binary heaps, which require the tree to remain a complete binary tree. Williams describes this property in the original formulation of heaps as:

“The active elements of the array are maintained and densely packed, as elements $A[1]$ to $A[n]$ ” [1, p.347]

Newly inserted values are placed in the leftmost available position on the last level, ensuring that the array-based tree remains complete.

However, appending the element in this way can violate the heap-order property, which requires that every parent node is less than or equal to its children. So to restore this invariant, the next line in the function invokes:

```
self._heapify_up(len(self.heap) - 1)
```

This method performs upward heapification by comparing the newly inserted element to its parent and swapping them if the parent is greater, where the parent node at index i is located at $(i-1)//2$, and its children are located at $2i + 1$ and $2i + 2$. These relationships follow directly from the array-based binary tree representation described by Williams:

“The elements are normally so arranged that $A[i] \leq A[2i]$ for $2i \leq n$, and $A[i] \leq A[2i + 1]$ for $2i + 1 \leq n$ ” [1, p.347]

The `_heapify_up` process continues recursively until either the newly inserted element is no longer smaller than its parent or it reaches the root (index 0). This guarantees that the heap-order property is restored locally and that no violations remain in the path from the inserted element up to the root.

These two steps ensure that the tree remains complete and that the heap-order is maintained.

In addition to the base implementation, the code also handles two edge cases:

1. If the inserted value is larger than its parent, no swaps occur, and the heap remains valid
2. If the inserted value becomes the new minimum, it bubbles up to the root, ensuring constant-time access in future `extract_min` calls.

Therefore, the insertion procedure conforms to the properties and design principles of Binary Min Heaps as originally defined by Williams. It both preserves heap invariants and operates on the minimal path required for restoration.

Proof of Correctness for Fibonacci

The first line of the function,

```
node = FibonacciHeapNode(key)
```

Correctly implements Fredman and Tarjan's definition of Insert, which states:

"To carry out insert (i, h), we create a new heap consisting of one node containing i " [2, p.599]

This line constructs a new heap-ordered node with the inserted key. The FibonacciHeapNode constructor ensures that the newly created node satisfies all required F-heap invariants; no parent, no children, a rank of zero, and is unmarked. It is initialized in a circular doubly linked list ($\text{left}=\text{self}$, $\text{right}=\text{self}$), preparing it for root list.

These properties exactly align with the structural assumptions of Fibonacci Heaps as described by Fredman and Tarjan:

"Each node contains a pointer to its parent (or to a special node null if it has no parent) and a pointer to one of its children. The children of each node are doubly linked in a circular list." [1, p. 598].

After creating the node, the next step is to determine where it should be placed in the heap. Since Fibonacci Heaps consist of multiple trees stored in circular doubly linked list [1, p.598], the insertion process depends on whether the heap is empty or already contains nodes.

Case 1: Inserting into an Empty Heap

If the heap is empty, the new node becomes the first and only root:

```
if self.min_node is None:  
    self.min_node = node
```

This follows directly from Fredman and Tarjan's specification:

"If the heap is empty, its minimum node is null; in that case, the newly inserted node becomes the one-node root list of the new heap." [2, p.598]

By setting `self.min_node = node`, we ensure that:

- The new node is the only element in the heap
- The F-heap structure remains valid since a one-node heap trivially satisfies all Fibonacci Heap invariants.

Case 2: Inserting into a Non-Empty Heap

If the heap is not empty, the new node must be spliced into the existing circular doubly linked root list. This is achieved through the following pointer assignments:

```
node.right = self.min_node.right  
node.left = self.min_node  
self.min_node.right.left = node  
self.min_node.right = node
```

These steps ensure that the new node is properly integrated into the root list whilst preserving the circular doubly linked structure:

- | | |
|--|--|
| 1. <code>node.right = self.min_node.right</code> | The new node points to what was previously the right neighbor of <code>self.min_node</code> . |
| 2. <code>node.left = self.min_node</code> | The new node's left pointer is set to <code>self.min_node</code> , preserving bidirectional linkage. |
| 3. <code>self.min_node.right.left = node</code> | The previous right neighbour of <code>self.min_node</code> updates its left pointer to reference the new node. |
| 4. <code>self.min_node.right = node</code> | The new node becomes the immediate right neighbour of <code>self.min_node</code> , completing the insertion. |

By performing these pointer updates, the new node integrates into the Fibonacci Heap's circular root list. This process is equivalent to the meld operation defined by Fredman and Tarjan as:

"To carry out `meld` (h_1, h_2) we combine the root list of h_1 and h_2 into a single list, and return as the minimum node of the new heap either the minimum node of h_1 or the minimum node of h_2 , whichever contains the item of the smaller key. (In the case of a tie, the choice is arbitrary.)" [1, p.599]

Since inserting a new node can be viewed as melding a one-node heap into an existing F-heap, the code correctly performs the meld operation in constant time, preserving all F-heap invariants.

After insertion, the final step is to update the global minimum pointer (`self.min_node`) if necessary:

```
if node.key < self.min_node.key:  
    self.min_node = node
```

This directly follows Fredman and Tarjan's requirement:

"Return as the minimum node of the new heap either the minimum node of h_1 or the minimum node of h_2 , whichever contains the item of the smaller key." [1, p.599].

Since `self.min_node` always tracks the current global minimum, this logic check ensures that if the new key is smaller than `self.min_node.key` we update `self.min_node` to reference the new node. Otherwise, `self.min_node` is unchanged, maintaining correctness. This guarantees that `self.min_node` always references the root with the smallest key, preserving the Fibonacci Heap invariant.

Stress Tests

To validate the robustness of our code, we implemented a comprehensive set of stress tests for insert.

The **Bulk Insertion Test** repeatedly inserts a large number of elements in descending order – the worst-case scenario – to force maximal upward reordering with `_heapify_up`. After each insertion, the heap structure is immediately validated, and the current heap size is compared against the expected value, although by setting the inter This continuous validation ensures that the complete binary tree property and the heap-order invariant are maintained throughout the insertion process.

The **Duplicate Insertion Test** focuses on stress-testing the heap’s handling of non-unique keys. In this test, we insert a large number of duplicate values – using keys 0, 1000, -1, `float('inf')` – to challenge the robustness of the comparison logic and the integrity of the heap structure. Validation was performed per insertion but unlike the other stress tests for insertion, can be changed to ensure that the test does not take unreasonable times to complete. Validation ensures that the insertion procedure correctly maintains the heap invariants even when the key comparisons are ambiguous. Tests were completed

The final test, **Random Order Insertion Test**, simulates typical usage by inserting elements in a random sequence. This test verifies that, regardless of insertion order, the heap-property and heap-order invariant are maintained, and that the internal size of the heap matches the number of insertions.

Additionally, our worst-case benchmarking function for insertion, is also designed to work as a worst-case insertion, similar to Bulk Insertion Test. During each measurement, the heap is constructed in descending order. This ensures that each constructs a heap by inserting elements in descending order and its results are displayed later.

```
start_value = random.randint(size * 10, size * 20)
return [start_value - i * 10 for i in range(size)]
```

This ensures that each benchmarked insertion reflects the theoretical worst-case behavior of the data structure.

Worst Case Time Complexity

Insertion Time Complexity Analysis for Fibonacci Heap

The insert operation in a Fibonacci Heap begins by creating a new heap-ordered node containing the given key. This is done by calling the `FibonacciHeapNode` constructor, which initializes the node with no parent or children, a degree of zero, an unmarked state, while also settings its left and right pointers to itself to form a circular doubly linked list. This is done in accordance with Fredman and Tarjan’s requirement, “to carry out `insert(i, h)`, we create a new heap consisting of one node containing `i`” [1, p.599]. This single-node heap trivially satisfies all the required invariants.

Once the node is created, the insertion procedure checks whether the heap is empty by examining the pointer to the minimum node, `min_node`. If the heap is empty, the new node is assigned as `min_node`. Otherwise, the node is inserted into the existing circular doubly linked root list. This is done through a fixed series of pointer assignments: the new node’s left pointer is set to the current `min_node`, its right pointer is set to `min_node.right`, and the adjacent pointers in the list are updated accordingly. These operations are all performed in constant time as they involve a fixed number of pointer updates, independent of the number of elements in the heap - no loops or recursive calls are involved. Thus re-linking is bounded to constant time.

After inserting the new node into the root list, a single comparison is performed between the new node’s key and the current `min_node`’s key. If the new key is smaller, `min_node` is updated to reference the new node.

The final comparison is also completed in constant time as we are comparing are only ever comparing two values.

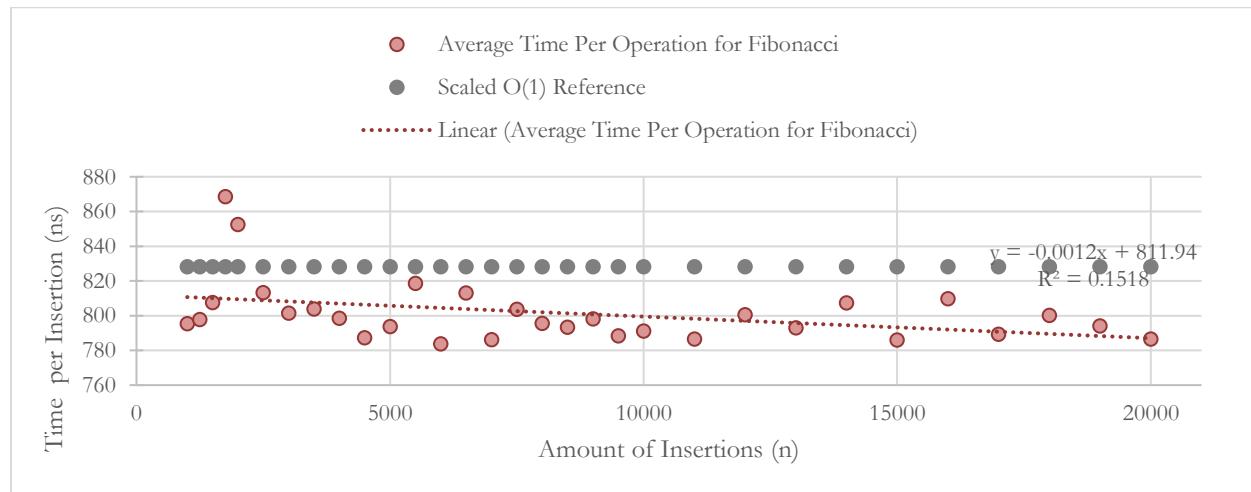
Finally, the overall node count is incremented, and the new node is returned. Since each individual step – node creation, pointer manipulation, key comparison, and counter updated – is executed in constant time, the entire insertion process runs in $O(1)$ worst-case time. Moreover, because the insertion does not defer any structural adjustments for later operations, its amortized time complexity remains $O(1)$.

Insertion Time Complexity Analysis for Fibonacci Heap Sort

The insert operation in a Binary Heap begins by appending the new element to the underlying array, preserving the shape property of the heap. This approach ensures that the array stays packed, as described by Williams: “the active elements of the array are maintained and densely packed, as elements $A[1]$ to $A[n]$ ” [2, p.347]. After the new element is added the insertion function invokes heapify up from the index of the newly appended element. In the heapify up, the inserted element is compared to its parent (located at index $(i - 1) // 2$). If the new element is smaller, it is swapped with its parent. This process is repeated until either the new element is no longer less than its parent or it reaches the root of the heap. In the worst case, the new element might bubble all the way from the lowest level to the root, which takes $O(\log n)$ comparisons and swaps, where n is the total number of elements in the heap. Thus, the worst-case time complexity of insertion in a Binary Heap is $O(\log n)$.

Benchmark Results

Figure 3.1.1: Average Time Per Insertion for Fibonacci (Worst Case)



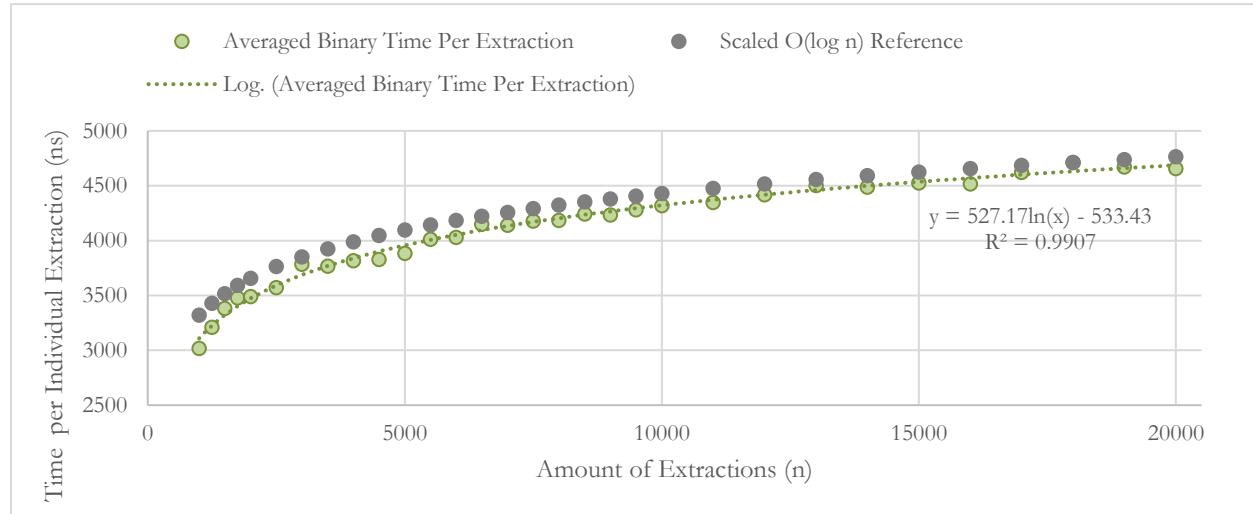
The empirical performance of Fibonacci Heap’s insertion operation exhibits near-constant growth. The measured values (red) consistently hover around 800 nanoseconds across a broad range of input sizes.

The fitted linear model $y = -0.0012x + 811.94$ with a low R^2 value of 0.1518 indicates that the variation in n contributes negligibly to change in performance. This observation matches the theoretical expectation of constant-time insertion, as $O(1)$ time, by definition, means that the time per operation is independent from the amount of insertions.

Additionally, scaled $O(1)$ reference points (gray) remain in close proximity to the measured data, reinforcing the notion that insertion does not scale upward with input size. Although not strictly fitted, these reference markers reflect the same flat behavior as the empirical values, providing a straightforward, visual comparison.

Taken together, the near-alignment of the measured data, linear regression, and scaled reference curve strongly suggests that Fibonacci insertion is effectively constant-time. This result aligns well with the established theoretical properties of Fibonacci heaps.

Figure 3.1.2: Average Time Per Extraction for Binary (Worst Case)

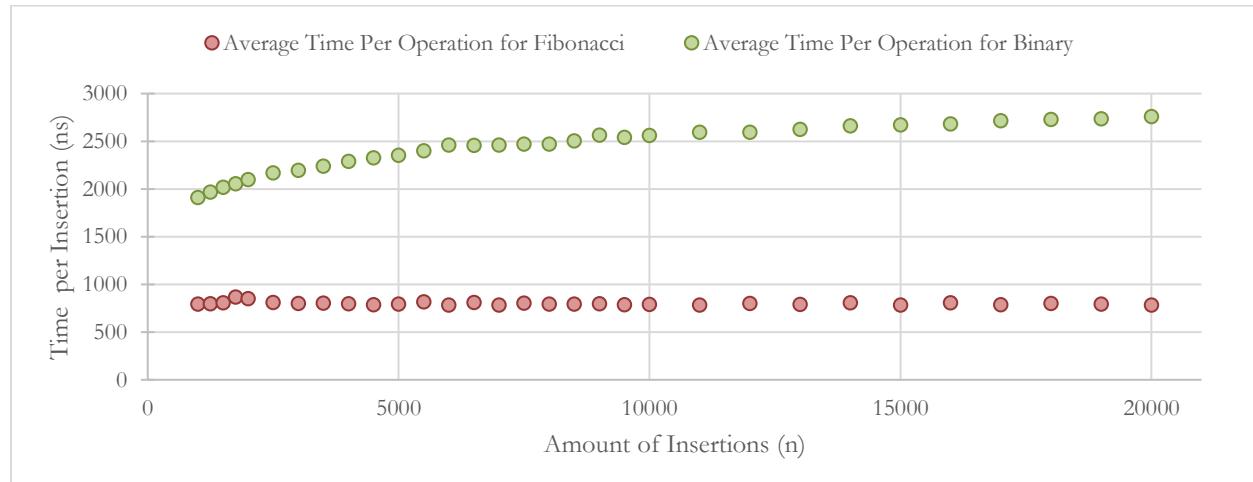


The empirical performance of Binary Heap's insertion operation exhibits sublinear growth. The measured values (green) show a clear upward trend that closely follows the fitted logarithmic model of $y = 527.17\ln(x) - 533.43$, with a high R^2 value of 0.9907. This indicates that as n grows, the average insertion time increases in a manner consistent with the theoretical $O(\log n)$ expectation for binary heaps.

Additionally, the scaled $\log(n)$ reference points (gray) demonstrate the same curvature, serving as a practical overlay for asymptotic comparison.

The near-alignment between the empirical data, regression fit, and scaled theoretical model provides strong evidence that Binary heap insertion conforms to the expected $O(\log n)$ complexity in worst-case scenarios, as described by William.

Figure 3.1.3: Average Time Per Insertion for Comparison (Worst Case)



Both Binary and Fibonacci Heaps exhibit empirical data that align well with their respective theoretical bounds for the insertion operation. When comparing them side by side, we can see a difference between their scaling behavior, however minimal.

As shown, Fibonacci Heaps data points (red) remain nearly flat at around 800 nanoseconds, whereas Binary Heaps data points (green) rise roughly 1000 nanoseconds, from around 2000 to just under 3000 nanoseconds.

The difference stems from their structural implementations. Fibonacci Heaps require minimal linking steps to attach a new node to the root list which keeps insertion effectively constant time. On the other hand, Binary Heaps must “sift up” (heapify up) the inserted values through the heap’s levels, a process which naturally grows with $\log n$.

Thus, the evidence supports the conclusion that while both heaps preserve their theoretical complexities, Fibonacci Heaps maintain a practical edge for insertion-intensive workloads due to their minimal overhead.

3.2 Extraction Analysis

Explanation of Functions

Before proceeding into comparative analysis, we first outline the mechanics of extraction in both Binary and Fibonacci Heaps.

First let's look at how Extraction works for Binary heap.

The extract_min operation in Binary Heap follows two stages: removal and restoration. The Binary Heap is a complete binary tree maintained in array form, enforcing min-heap property – the key at every node is less than or equal to the keys of its children.

The removal phase begins by deleting the root node – located at index 0 – which contains the minimum value. To preserve completeness of the binary tree, the last element in the array is moved into the root's position. This step preserves the binary tree's shape but can violate the min-heap property. To restore correctness, the heap will move into restoration.

To illustrate the restoration phase, consider a Binary heap stored as an array in level order:

[2, 4, 6, 8, 10]

The root node 2 is removed, and the last element 10 is moved to index 0:

[10, 4, 6, 8]

This completes the removal phase and transitions us into restoration.

The restoration phase begins with heapify-down, a recursive process which starts from the root. At each step, the current node is compared with its left and right children. If one of the children has a smaller key, we swap it with the smallest of the two. This continues until the node is smaller than both children or reaches a leaf.

The table below continues our example and demonstrates how Heapify-Down functions:

Table 3.2.1: Heapify-Down Restoration

Step	Heap Array	Index Compared	Children	Swap Made	Notes
1	[10, 4, 6, 8]	0	4 (L), 6 (R)	10 ↔ 4	Swap with smallest child
2	[4, 10, 6, 8]	1	8 (L)	10 ↔ 8	Swap again
3	[4, 8, 6, 10]	3	-	-	No children; heap property restored

Once heapify-down completes, Binary Heap satisfies min-heap property again. This completes the Binary Heap extraction method.

Next, let's focus on the Fibonacci variant of extract_min.

The extract_min operation in a Fibonacci Heap is composed of three stages: removal, consolidation, and finalization. The heap maintains a dedicated pointer to the minimum element, min_node, which allows constant-time access to the smallest key.

The removal phase begins by extracting min_node from the root list. If min_node has any children, each is promoted to the root list. These children have their parent pointers cleared, and their marked flags reset to False, as roots nodes should not be marked. This step ensures all formerly nested subtrees are exposed at the root level.

To illustrate, consider a heap in which min_node = 13, and we have the following nodes 2, 8, 233, 34→55, and 13→(89→144, 21)

Table 3.2.2: Child Promotion During Removal

Node	Parent Before	Marked After	Notes
89	13	None	Promoted to root list; still has child 144
21	13	None	Promoted to root list

Note: Children of min_node are promoted to the root list. Each has its parent pointer cleared and marked flag reset to maintain heap invariants.

From this point forward, the root list now contains the following top-level nodes:

2, 8, 233, 34→55, 89→144, 21

This completes the removal phase and allows us to transition the heap into the consolidation phase, where trees of equal degrees will be merged and a new min_node will be selected.

The goal of consolidation is to ensure that there are no more than one tree in the root list has the same degree. This is accomplished using an auxiliary array indexed by degree. As the heap scans the root list, each tree is placed into the array at the index corresponding to its degree. As the heap scans the root list, each tree is placed into the array at the index corresponding to its degree. If a collision occurs (another tree already exists at that degree), then the two trees are linked: the one with the smaller root key becomes the new parent, and its degree increases by one. The process continues until all degrees are “unique”.

The table below tracks each step of the consolidation process and will show how trees are progressively merged and reinserted.

Table 3.2.3: Consolidation Process

Step	Degree 0	Degree 1	Degree 2	Degree 3
0	2, 8, 233, 21	34→55		
		89→144		
Merge 2 and 8 of Degree 0 and				
1	233, 21	2→8, 34→55, 89→144		
		Merge 2→8 and 34→55 of Degree1		
2	233, 21	89→144	2→(8, 34→55)	
	Merge 233 and 21 of Degree 0			
3		233→21 89→144		
		Merge 233→21 and 89→144 of Degree 1		
4			2→(8, 34→55) 21→(233, 89→144)	
			Merge 2→(34→55, 8) and 21→(233, 89→144) of Degree 2	
5				2→(8, 34→55, 21→(233, 89→144))

Once consolidation completes, the heap contains a newly structured root list composed of unique degree trees. The final phase of extract_min is to scan this root list and select the new min_node. This operation is done to ensure that the heap maintains fast access to its minimum element for future operations.

From our example, after consolidation, the root list contains:

2→(8, 34→55, 21→(233, 89→144))

To finalize, the heap compares the root keys of all remaining trees. So in this case, node 2 is the only root, so it is chosen by default.

This completes the full trace and explanation of a single extract_min operation.

Proof of Correctness

Insertion proofs of correctness are separated into Binary and Fibonacci. Both Binary and Fibonacci proofs of correctness for Insertion will compare the implementation of these functions with the accepted expectations of the implementations.

Proof of Correctness for Binary

The first line of the function,

```
min_value = self.heap[0]
```

retrieves the root of the binary heap, which is always the minimum value in a min-heap. This behavior follows from the heap-order property defined by Williams, who states that:

“The element are normally so arranged that $A[i] \leq A[2i]$ for $2i \leq n$, and $A[i] \leq A[2i+1]$ for $2i + 1 \leq n$ ” [2, p.347]

This ensures that each node is less than or equal to its children, and therefore, the minimum element is always located at the root (i.e., index 0)

The next line of our code:

```
self.heap[0] = self.heap.pop()
```

Replaces the root with the last element o the heap and removes the last item. This mains the shape property of the heap, which requires that the heap remains a complete binary tree (*that all levels are filled except possibly the last, which is filled from the left to the right*). This is done to maintain the array-based representation described by Williams:

“The active elements of the array are maintained and densely packed, as elements $A[1]$ to $A[n]$ ” [2, .347],

However, swapping can potentially violate the heap-order property. To restore it, we run:

```
self._heapify_down(0)
```

This function restores the min-heap invariant by percolating down the new root. At every step, it compares the current lement to its left and right children (indices $2i + 1$ and $2i + 2$), and swaps it with the smaller of the two, if necessary, This process matches the restoration strategy described by Williams’s OUTHEAP procedure:

“Given array A, elements 1 to n of which form a heap, OUTHEAP assigns to out the value of $A[1]$, the least member of the heap, and rearranges the remaining members … to form a heap.” [1, p.348].

We continue this process until the heap property is restored or the node becomes a leaf. Since the heap is a complete binary tree, its heigh is $\lceil \log_2 n \rceil$.

Finally, our code handles two edge cases explicitly:

1. If the heap is empty, the line:

```

if not self.heap:
    raise HeapError("Heap is empty")

```

ensures robust behavior by preventing invalid access.

2. If the heap contains only one element, the branch:

```

if len(self.heap) == 1:
    return self.heap.pop()

```

Directly returns the only time since trivially, one node satisfies the heap-order and shape properties as the heap becomes empty.

Proof of Correctness for Fibonacci

The first line of the extract_min function,

```

z = self.min_node

```

correctly identifies the node with the minimum key in the Fibonacci Heap. According to Fredman and Tarjan,

"To carry out delete min (h), we remove a node of minimum key from the root list" [1, p.599]

By storing this node in the variable z, the implementation prepares to remove it and return its key. If the heap is empty (self.min_node is None), the function raises a HeapError, in accordance with the fact that:

"If the heap is empty, its minimum node is null; in that case, there is no node to remove" [1, p.598]

Once it is confirmed that z is not None, the function proceeds to promote z's children (if any) to the root list:

```

if z.child:
    children = []
    c = z.child
    while True:
        children.append(c)
        c = c.right
        if c == z.child:
            break
    for child in children:
        child.parent = None
        child.left.right = child.right
        child.right.left = child.left
        child.left = self.min_node
        child.right = self.min_node.right
        self.min_node.right.left = child
        self.min_node.right = child

```

This segment of code directly follows the description in Fredman and Tarjan's specification:

"We take each of the children of the minimum node and add them to the root list." [1, p. 599]

The children are detached from the child list of z, their parent pointers are set to None, and they are inserted into root list via pointer manipulation. This maintains the Fibonacci Heap invariants:

1. Null parent – newly promoted children are roots and thus must have no parent.
2. Circular root list – the children are linked into the doubly linked root list, preserving its structure.

Once all the children of z have been promoted to the root list, the minimum node itself is removed :

```
z.left.right = z.right  
z.right.left = z.left
```

This step unlinks z from the circular root list, ensuring it is no longer part of the heap. This aligns with Fredman and Tarjan's definition of Delete-Min:

"We remove the minimum node from the root list..." [1, p. 599]

After the removal, the algorithm checks whether z was the only node in the root list:

```
if z == z.right:  
    self.min_node = None
```

If z was the sole node, the heap is now empty, and the self.min_node pointer is cleared. Otherwise, a new tentative minimum is selected (z.right), and the consolidation process is initiated:

```
else:  
    self.min_node = z.right  
    self._consolidate()
```

This corresponds to the consolidation step in Fibonacci Heap theory, which Fredman and Tarjan describe as follows:

"We repeatedly link any two trees of the same rank until at most one root remains of each rank." [1, p. 599]

The _consolidate function reduces the number of trees in the root list by linking roots of equal rank into a single tree, always attaching the higher-key root under the lower-key root.

After consolidation, the node count is decremented and the key of the removed node is returned:

```
self.num_nodes -= 1  
return z.key
```

This final step reflects Fredman and Tarjan's description:

"We then return the item of the node removed, decrementing the total number of items in the heap." [1, p. 599]

Through these steps, the extract_min function adheres to the formal definition of Delete-Min in Fibonacci Heaps. It promotes all children of the minimum node to the root list, removes the minimum node, consolidates the root list to enforce unique degrees, updates the minimum pointer, and returns the correct value.

Stress Test

To validate the robustness of our code, we implemented a comprehensive set of stress tests for `extract_min`.

The **Bulk Extraction Test** inserts a large amount of element in a strictly descending order and then extracts them all. This is designed to trigger maximal restricting for both heap types and serves as a worst-case input sequence. During execution, the extracted values are checked to ensure ascending order, and the internal structure is validated periodically to detect corruption or invariant violations.

The **Duplicate Extraction Test** stresses the heaps' handling of identical keys. A large number of duplicate values are inserted and then extracted, with each result validated for correctness. This test detects edge cases in heap traversal logic and ensures stability when key comparisons cannot distinguish between elements.

The **Interleaved Churn Test** aims at introducing a more realistic workload by interleaving partial extractions with reinsertion of new elements, followed by a complete extraction of the remaining heap. This simulates dynamic usage patterns and challenges the correctness of lazy restructuring mechanisms.

Finally, the **Empty Heap Test** confirms that attempting to extract from an empty heap raises the correct exception rather than resulting in undefined behavior. This ensures robustness in boundary cases.

In addition to these stress tests, the primary worst-case benchmarking function is also designed to work as a worst-case extraction pattern. During each measurement, the heap is fully constructed and then completely emptied through repeated calls to `extract_min`:

```
for value in data:  
    heap.insert(value)  
  
while heap.num_nodes > 0:  
    heap.extract_min()
```

This ensures that each benchmark reflects the theoretical worst-case behavior of the data structure, especially during consolidation phases in Fibonacci Heaps.

Together, the stress and benchmark tests confirm that both implementations behave correctly and maintain their structural invariants across a wide range of conditions.

Worst Case Time Complexity

Extraction Time Complexity Analysis for Fibonacci Heap Sort

The `extract_min` operation begins by identifying and storing the current minimum node, which is a constant-time access. If the heap is empty, an exception is raised, terminating the operation in constant time.

Otherwise, the algorithm proceeds to remove the minimum node from the heap and restructure the data to preserve heap invariants.

If the minimum node has children, the algorithm iterates over its child list once, detaching each child and promoting it to the root list. Each child is processed individually: its parent pointer is cleared, it is removed from its sibling list, and then it is inserted into the root list of the heap. These are all constant-time operations per child, and since each child is handled once, this is bounded by $\log n$ where n is the number of nodes in the heap. Thus, the run time for this portion of the code is $O(\log n)$.

After promoting all children to the root list, the algorithm removes the minimum node from the circular list of roots. This operation involves a fixed number of pointer updates and completes in constant time. If the removed node is only a root, the heap becomes empty and no further restructuring occurs. Otherwise, the

algorithm proceeds to a consolidation phase, which links trees of equal degree in the root list to enforce a bound on the number of trees. This is achieved by iterating over the root list and applying pairwise linking. Since each linking operation is constant time and the number of distinct tree degrees is $O(\log n)$, the consolidation step also runs in $O(\log n)$ time.

Therefore, because `extract_min` never performs full traversal over all nodes in the heap, and the only operations that scale with input size are those dependent on the number of distinct tree degrees or the degree of removed nodes, the worst-case and amortized time complexities of `extract_min` are both $O(\log n)$, consistent with the bounds established by Fredman and Tarjan for Fibonacci Heaps.

Extraction Time Complexity Analysis for Binary Heap Sort

The `extract_min` operation in Binary Heap begins by accessing the root node which stores the minimum value due to heap-order property. This is a constant-time operation. If the heap is empty, the function raises an exception, terminating in $O(1)$ time. If the heap contains only one element, it is removed and returned immediately with `pop()`, which also completes in constant time.

When the heap has more than one element, the algorithm removes the root by replacing it with the last element in the array. This ensures the complete binary tree structure (*shape property*) is preserved, as removing the last element from the array maintains density. This reassignment and `pop` operations are both constant time operations.

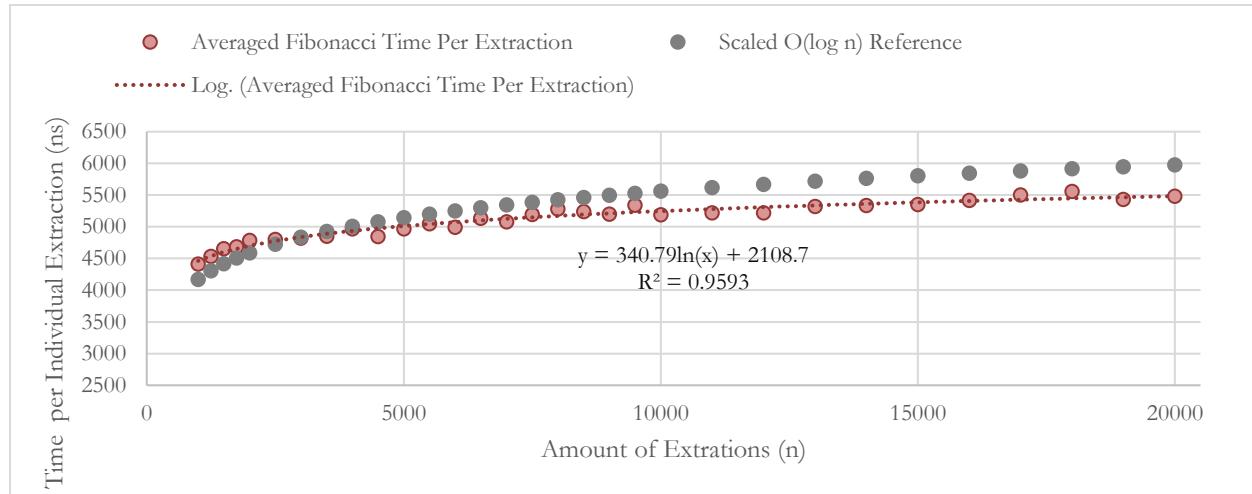
After substitution, it is possible that heap-order property is violated and requires re-heapification. To restore the invariant, the algorithm performs a downward percolation from the root using the `_heapify_down` function. For each step, the node is compared to two children (if present), and it is swapped with the smaller child if it violates the min-heap condition. This process continues recursively until the node is correctly positioned, or it reaches a leaf node.

Since Binary Heaps are complete binary trees, the height of the heap is at most $\lfloor \log_2 n \rfloor$, where n is the number of elements. Thus, in the worst case a node may be moved from the root all the way to the lowest level, performing $O(\log n)$ comparisons and swaps.

Therefore, because no other part of the algorithm iterates over the heap or processes multiple elements, and no auxiliary structures are used, the overall worst-case time complexity for binary extraction is $O(\log n)$.

Benchmark Results

Figure 3.2.1 Average Time Per Extraction for Fibonacci Heaps (Worst Case)

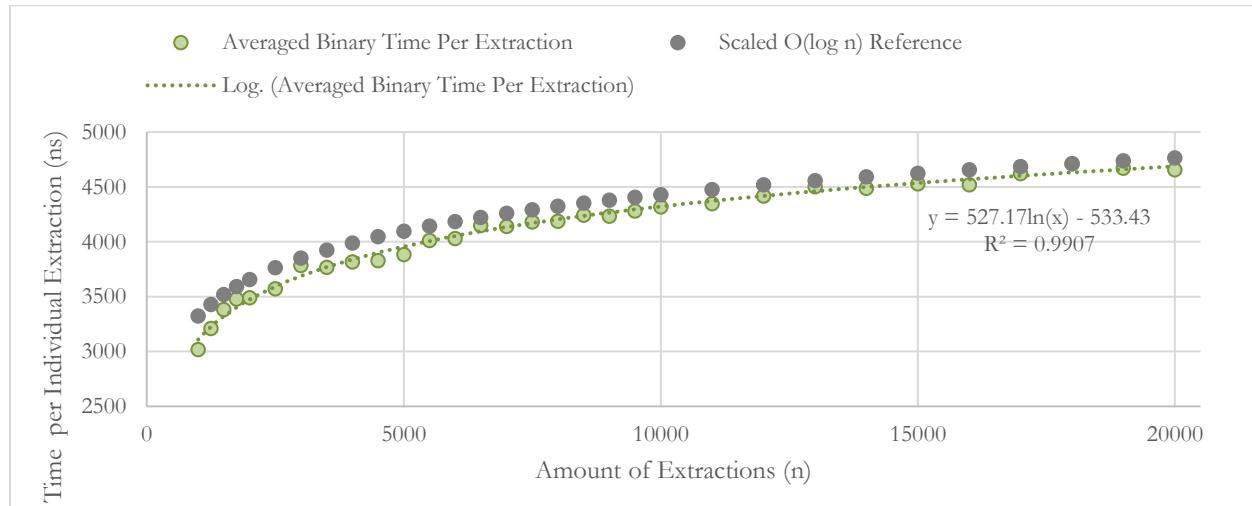


The empirical performance of Fibonacci Heap's extract_min operation demonstrates strong sublinear growth. The measured values (red) closely follow the fitted logarithmic model with a regression curve of the form $y = 340.79\ln(n) + 2108.7$, and a high R^2 value of 0.9593.

Additionally, scaled reference points for $\log n$ (gray) reinforce the theoretical expectation. Although not fitted, they demonstrate the same curvature as the measured data and serve as a practical overlay for asymptotic comparison.

The near-alignment between the empirical data, regression fit, and scaled theoretical model provides strong evidence that Fibonacci extract_min conforms to the expected $O(\log n)$ complexity in worst-case scenarios, as described by Fredman and Tarjan.

Figure 3.2.2 Average Time Per Extraction for Binary (Worst Case)

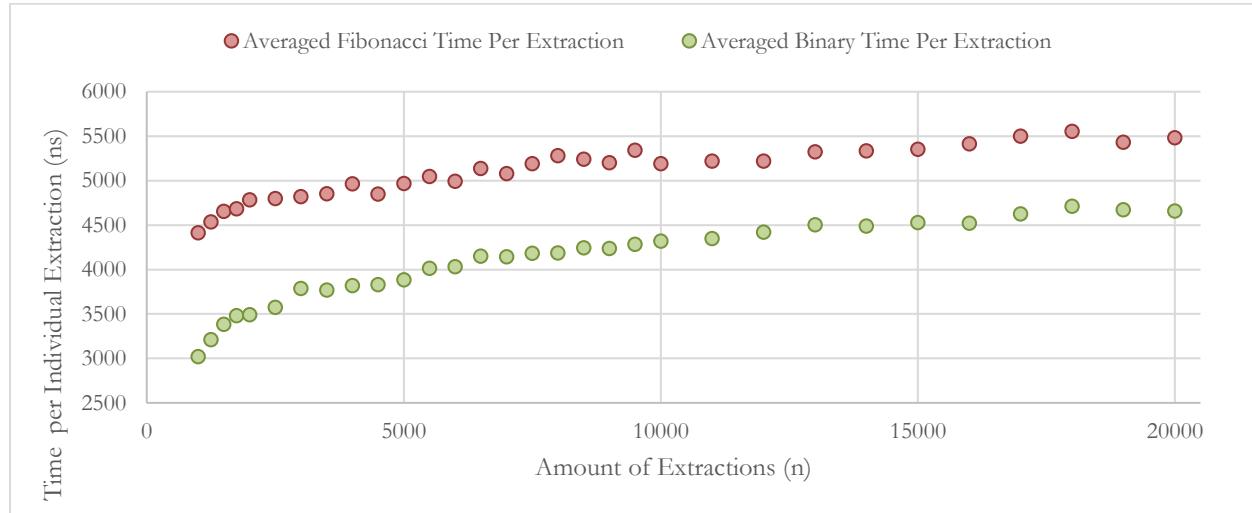


The empirical performance of Binary Heap's extract_min operation strongly aligns with the expected logarithmic growth. The measured average times (green dots) follow the fitted logarithmic regression curve $y = 527.17\ln(n) - 533.43$ closely, with a high R^2 value of 0.9907.

Additionally, like with Fibonacci, we have added scaled $\log n$ reference points (red dots) to visualize the theoretical behavior in optimal environments with no noise.

The Binary Heap's tightly clustered data and high statistical fit support the conclusion that its `extract_min` operation scales as $O(\log n)$ in the worst case, consistent with classical heap theory as originally formulated by Williams.

Figure 3.2.3 Binary and Fibonacci Average Time Per Extraction Comparison(Worst Case)



Both Binary and Fibonacci Heaps exhibit empirical data behavior consistent with their theoretical $O(\log n)$ time bounds for the `extract_min` operation. When comparing them side-by-side we can see that there is a consistent advantage for Binary Heaps. As shown in the graph, the Binary data points (green), remain below the Fibonacci data points (red) across all input sizes. This supports that Binary Heaps scale more efficiently than Fibonacci heapsort.

The difference here stems from the structural implementations: Binary Heap rely on in-place array swaps during percolation, which results in minimal memory overhead and tight cache locality. In contrast, Fibonacci Heaps involve promoting all children of the minimum node to the root list and performing a consolidation step, which introduces additional point updates and traversal costs – even if these operations were run in amortized $O(\log n)$ time.

Thus, the evidence supports the conclusion that while Fibonacci Heaps preserve logarithmic complexity, Binary Heaps are better suited for workloads dominated by `extract_min`.

3.3 Decrease Key Analysis

Explanation of function

Before proceeding into comparative analysis, we first outline the mechanics of decrease_key in both Binary and Fibonacci Heaps.

First let's look at how decrease_key works for Binary heap.

The decrease_key operation in Binary Heap follows two stages: update key and heapify-up. The Binary Heap is a complete binary tree maintained in array form, enforcing min-heap property – the key at every node is less than or equal to the keys of its children.

During the update key stage, the element is searched by its current value, and once found, its key is replaced with the new, lower value in the heap array. This step preserves the binary tree's shape but can violate the min-heap property. To restore correctness, the heap will move into the heapify-up stage.

The _heapify_up process continues recursively until either the newly inserted element is no longer smaller than its parent or it reaches the root (index 0). This guarantees that the heap-order property is restored locally and that no violations remain in the path from the inserted element up to the root.

To illustrate the decrease key process, consider a following Binary heap:

[2, 5, 8, 10, 15, 20]

We want to decrease key at index 4 (value 15) → new value = 1

Table 3.3.1: Binary Heap Decrease-key Example

Step	Heap Array	Index of updated Element	Parent's Index	Values Compared	Swap Made	Notes
1	[2, 5, 8, 10, 1, 20]	4	1	5vs 1	Yes	1 is swapped upward
2	[2, 1, 8, 10, 5, 20]	1	0	2 vs 1	Yes	1 becomes the new root
3	[1, 2, 8, 10, 5, 20]	0	-	-	-	Heapify complete

Next, let's focus on the Fibonacci variant of decrease key. The decrease key operation in a Fibonacci Heap is composed of three stages: update key, cut, and cascading_cut. The process required depends on the value of the key after the update, whether the parent node is marked, etc.

The update key phase begins by comparing the current key value and the new key value. New value must be less than the current value. If this condition is met, we update the key of the node. At this point, the min-heap property may be violated since the key value might be smaller than its parent.

This leads us to the cut and cascading_cut phases. In cut phase, the node which key was just updated is detached from its parent node and added to the root list. The cascading_cut phase ensures that the heap maintains its structural properties when a child is cut from its parent.

After the initial cut, we check whether the parent of the removed node is marked. If it is not marked, we simply mark it. However, if it is already marked, we recursively cut the parent as well and repeat this process up the tree. This mechanism is designed to limit the number of children a node can lose before being removed from its parent. By doing so, the amortized time complexity for decrease key operations remains

constant. The cascading_cut phase may thus result in a sequence of cuts that propagate up the heap, ultimately preserving the heap's balance and ensuring efficient operations.

The table below tracks each step of decreasing the key 18 to 3.

Table 3.3.2: Fibonacci Decrease-key Example

Step	Heap Root List	Operation	Min Pointer
1	5→(12(marked)→18),50	Starting point	5
2	5→(12(marked)→3),50	Decrease key 18 to 3	5
3	5→12(marked),50,3	Cut 3 to root (since 3<12)	5
4	5,50,3,12	Cascade cut to root (since 12 is marked)	5
5	5(marked),50,3,12	Parent 5 is unmarked → mark it	5
6	5(marked),50,3,12	Update min pointer (since 3<5)	3

Proof of Correctness

Proof of Correctness for Binary

The first three lines of the function,

```
def decrease_key(self, uid: int, new_value: T) -> None:
    """Decreases the value of an element in the heap."""
    if uid not in self.position_map:
        raise HeapError("Element not found")
    index = self.position_map[uid]
    if new_value > self.heap[index].value:
        raise ValueError("New value is greater than the old value")

    self.heap[index].value = new_value
```

correctly decreases key from the node in Binary Heap. According to Weiss,

“The decreaseKey(p) operation lowers the value of the item at position p by a positive amount” [3, p.254]

By comparing new value and old value, we enforce to lower the value of the item.

```
self._heapify_up(index)
```

This method performs upward heapification by comparing the newly updated element to its parent and swapping them if the parent is greater. This step follows instruction described by Weiss:

“Since this might violate the heap order, it must be fixed by a percolate up” [3, p.254]

Proof of Correctness for Fibonacci

The first three lines of the function,

```
if new_value > node.key:  
    raise HeapError("New value is greater than the current value")  
node.key = new_value
```

correctly decreases key from the node in the Fibonacci Heap. According to Fredman and Tarjan,

"To carry out decrease key (A, i, b), we subtract A from the key of i , find the node x containing i , and cut the edge joining x to its parent $p(x)$." [1, p.597]

"Decrease the key of item i in heap b by subtracting the nonnegative real number A ." [1, p.601]

Subtracting a non-negative value is equivalent to updating the key to a smaller value than the current one.

After updating the key, the function moves to the cut phase, where the node is detached from its parent and added to the root list.

```
node.key = new_value  
parent = node.parent  
if parent and node.key < parent.key:  
    self._cut(node, parent)  
    self._cascading_cut(parent)
```

```
def _cut(self, x: FibonacciHeapNode[T], parent: FibonacciHeapNode[T]) -> None:  
    """Cuts the link between x and its parent."""  
    if parent.child == x:  
        if x.right != x:  
            parent.child = x.right  
        else:  
            parent.child = None  
        x.left.right = x.right  
        x.right.left = x.left  
        parent.degree -= 1  
  
        x.parent = None  
        x.left = self.min_node  
        x.right = self.min_node.right  
        self.min_node.right.left = x  
        self.min_node.right = x  
        x.marked = False
```

This segment of code directly follows the description in Fredman and Tarjan's specification:

"This requires removing x from the list of children of $p(x)$ and making the parent pointer of x null. The effect of the cut is to make the subtree rooted at x into a new tree of b , and requires decreasing the rank of $p(x)$ and adding x to the list of roots of b ." [1, p.601]

x is detached from the child list of its parent and added to the root list. Also, the necessary pointers and node states are updated to maintain the Fibonacci properties:

1. Child pointer – since the parent points to one of its children, this pointer must be updated or cleared if the removed node was referenced.
2. Null parent – the cut node becomes a new root and must have no parent pointer.
3. Circular root list – the children are linked into the doubly linked root list, preserving its structure.

```
def _cascading_cut(self, node: FibonacciHeapNode[T]) -> None:
    """Performs cascading cut on the parent of y."""
    while node.parent:
        parent = node.parent
        if not node.marked:
            node.marked = True
            break
        self._cut(node, parent)
        node = parent
```

This segment of code directly follows the description in Fredman and Tarjan's specification:

“After a root node x has been made a child of another node by a linking step, as soon as x loses two of its children through cuts, we cut the edge joining x and its parent as well, making x a new root as in decrease key. We call such a cut a cascading cut” [1, p.603]

“The purpose of marking nodes is to keep track of where to make cascading cuts. When making a root node x a child of another node in a linking step, we unmark x . When cutting the edge joining a node x and its parent $p(x)$, we decrease the rank of $p(x)$ and check whether $p(x)$ is a root. If $p(x)$ is not a root, we mark it if it is unmarked and cut the edge to its parent if it is marked.” [1, p.603]

Through this step, the cascading cut function enforces the structural discipline required by Fibonacci Heaps. It ensures that nodes do not lose multiple children without being promoted, maintaining balance in the tree.

Stress tests

To validate the robustness of our code, we implemented a comprehensive set of stress tests for decrease key.

The **Bulk Decrease-Key** Test repeatedly applies decrease_key operations to randomly selected elements after a large number of insertions. Each key is reduced by a small random value to simulate dynamic priority updates. After each operation, the heap is immediately validated to ensure that the heap-order invariant and structure remain intact throughout the update process.

The **Duplicate Decrease-Key** Test examines the heap’s handling of initially identical keys that are later made distinct. A large number of duplicate values are inserted, and each is then updated to a unique, smaller key using decrease_key. This stresses the heap’s repositioning logic and ensures structural correctness when transitioning from uniform to unique key values.

The **Decrease-Key-To-Min** Test reduces all keys in the heap to a shared minimum value (e.g., 0), forcing maximal structural change. In Fibonacci Heaps, this triggers widespread cascading cuts; in Binary Heaps, it causes maximum upward movement. Continuous validation confirms that the heap structure and ordering are preserved under this extreme reordering.

Worst case time complexity

Worst case time complexity for Binary Heap

After the element is updated the insertion function invokes heapify up from the index of the newly updated element. In the heapify up, the updated element is compared to its parent (located at index $(i - 1) // 2$). If the element is smaller, it is swapped with its parent. This process is repeated until either the updated element is no longer less than its parent or it reaches the root of the heap. In the worst case, the updated element might bubble all the way from the lowest level to the root, which takes $O(\log n)$ comparisons and swaps, where n is the total number of elements in the heap. Thus, the worst-case time complexity of insertion in a Binary Heap is $O(\log n)$.

Worst case time complexity for Fibonacci Heap

The decrease_key operation updates the key of a given node to a smaller value and may trigger structural adjustments to maintain the min-heap property. If the updated key becomes smaller than that of its parent, the min-heap property is violated. In response, the node is cut from its parent and added to the root list.

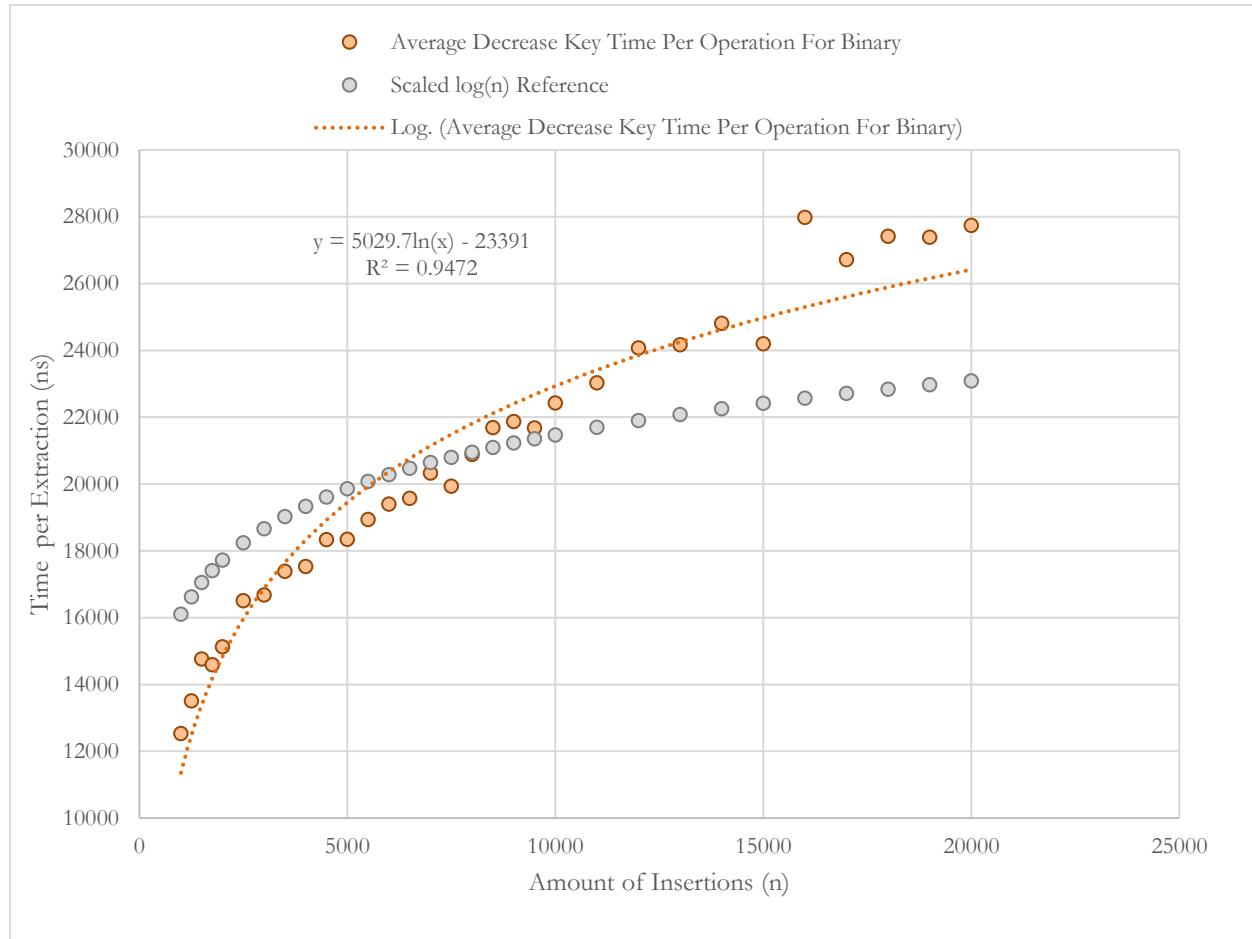
If the parent node is already marked, it too is cut and moved to the root list. This cascading cut process may continue recursively up the tree, cutting parents as long as they are marked.

In the worst case, a single decrease_key operation may trigger a sequence of cascading cuts along a path of up to n nodes, where n is the total number of nodes in the heap. Such a scenario can occur if the heap has become highly unbalanced, forming a long chain of parent-child relationships. Thus, the worst case time complexity is $O(n)$.

However, the design of Fibonacci Heaps restricts the frequency of such cascades through cutting a node which loses two children from its parent node. As a result, the amortized time complexity of decrease_key remains $O(1)$, ensuring that the operation is efficient on average over a sequence of operations.

Benchmark Results

Figure 3.3.1: Average Time Per Decrease Key for Binary (Worst Case)

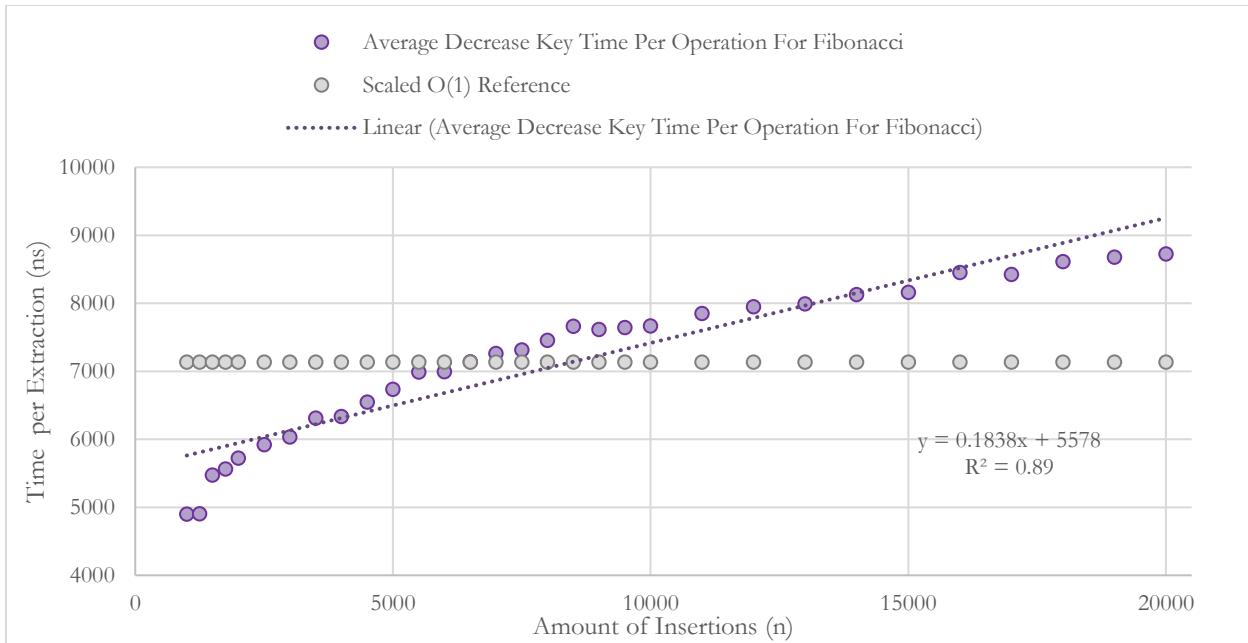


The empirical performance of Binary Heap's decrease_key operation exhibits sublinear growth. The measured values (orange) follow a clear upward trend that closely matches the fitted logarithmic model of $y = 5029.7\ln(n) - 23391$, boasting a high R^2 value of 0.9472.

As n increases, the average time per operation rises from around 12'000 nanoseconds to nearly 30'000 nanoseconds, reflecting the theoretical $O(\log n)$ complexity. The scaled $\log(n)$ reference points (gray) display the same curvature and serve as a straightforward comparison for asymptotic behavior.

This near alignment between the empirical data and the regression curve provides strong evidence to support that decrease_key in Binary Heap scales logarithmically under worst-case conditions.

Figure 3.3.2: Average Time Per Decrease Key operation for Fibonacci (Worst Case)



The empirical performance of Fibonacci Heap's decrease_key operation exhibits a gentle upward trend that remains close to constant time. The measured values (purple) start near 5'000 nanoseconds and rise to around 9'000 nanoseconds as n approaches 20'000, closely following the fitted linear model of $y = 0.1838x + 5578$ with an R^2 value of 0.89.

While the slight upward trend suggests some incremental overhead – likely due to pointer updates and internal housekeeping – the operation still performs near-constantly on average, which aligns with the theoretical $O(1)$ amortized expectation for decrease_key in Fibonacci Heaps. The scaled $O(1)$ reference (gray) serves as a visual comparison of theoretical behavior and actual behavior.

Overall the empirical data suggests that, despite small practical overheads, that Fibonacci Heaps maintain an effectively constant-time performance for decrease_key operations in worst-case conditions.

While both benchmarks validate the theoretical performance of decrease_key for their respective heap structures, the testing methodologies to exhibit worst case behavior differ significantly. For Binary Heaps, the worst-case scenario is simulated by selecting the deepest node from a freshly built heap. For Fibonacci Heaps, the test forces a consolidation through an extract_min and then selects a non-root node as the candidate for decrease_key. These distinct setups mean that a direct head-to-head comparison could be misleading; each benchmark is tailored to highlight the nuances of the respective heap's worst-case behavior rather than offering a uniform metric for comparison.

3.4 Merge Analysis

Explanation of Functions

Now let's investigate the functionality of the merge operations for both Binary and Fibonacci.

To start let's examine Binary Heap Merge,

The merge function in a Binary Heap combines two heaps by inserting each element of one heap (other) into the existing heap (self). Binary heaps are typically implemented as arrays or lists, where the structure maintains the heap property: every parent node is less than or equal to its child nodes in a minimum-heap (or greater than or equal to its children in a maximum-heap).

The merge process begins by first verifying that the input is a valid Binary Heap object. If not, a `HeapError` is raised to prevent undefined behavior. Once validated, the algorithm iterates through the array representation of the other heap.

For each element, it calls the `insert` method on the current heap (self). This `insert` operation appends the element to the end of the array and then bubbles it up through the heap until the heap property is restored. The bubbling up process involves repeatedly swapping the element with its parent if it is smaller (in a minimum-heap).

By repeating this process for each element in the other heap, the merge operation successfully integrates all of its elements into the current heap while preserving the binary heap property.

To visualize this, consider two heaps, A and B.

Heap A (self): [3,5,8], Heap B (other): [2,4]

Table 3.4.1: Binary Heap Merge Process

Step	Inserted Value	Heap A After Insert	Notes
1	2	[2,3,8,5]	2 bubbles up past 3 to become the new root
2	4	[2,3,8,4,5]	4 inserted at the end, bubbles up past 4 but not 3.

This results in the final merged Heap A being: [2,3,8,4,5].

Next we will investigate the functionality of the Fibonacci Heap Merge.

The merge function in a Fibonacci Heap combines two heaps by joining their root lists and updating metadata like the minimum node and node count. Fibonacci heaps are built using circular, doubly linked lists of tree roots, allowing them to efficiently support operations like merge in constant time. This makes them particularly powerful for applications like Dijkstra's algorithm, where frequent merges can be performance critical.

The function begins by verifying that the input `other` is indeed a `FibonacciHeap`. If not, a `TypeError` is raised to prevent undefined behavior. The function then handles two edge cases: if the other heap is empty (i.e., `min_node` is `None`), there is nothing to merge, so the function returns immediately. If the current heap (self) is empty but `other` is not, it simply adopts the `min_node` and `num_nodes` from `other`, effectively becoming a copy of `other`.

If both heaps are non-empty, the core logic of the merge is performed by calling `_concatenate(self.min_node, other.min_node)`. This operation joins the root lists of the two heaps into a single circular, doubly linked list. Since the roots are not sorted or structured like in a binary heap, no reordering is required—just a simple pointer adjustment, which makes the operation very fast ($O(1)$).

After concatenation, the function compares the `min_node` keys of the two heaps and updates `self.min_node` to point to the smaller one, ensuring the heap's minimum value remains accurate. Finally, the `num_nodes` of the current heap is increased by the number of nodes in the other heap to reflect the updated total.

This lazy merge strategy—deferring most structural work until later operations like `extract_min()`—is what gives Fibonacci heaps their theoretical performance advantages, especially in algorithms that do many merges and insertions. The example below outlines the process of the Fibonacci Heap Merge.

Heap A: Key 5, Heap B: Keys 2,8

3.4.2: Before Merge

Heap A Root List	Heap B Root List
[5]	[2]<->[8]
min_node: 5	min_node: 2
Num_nodes: 1	Num_nodes:1

3.4.3: Step by Step

Step	Action	Result
1	Check that B is a Fibonacci Heap	Yes
2	Check if B. <code>min_node</code> = none	No (2)
3	Check if A. <code>min_node</code> = none	No (5)
4	Concatenate root lists	[5]<->[2]<->[8]
5	Compare min nodes	Update self. <code>min_node</code> = 2
6	Add node counts	Update self. <code>num_nodes</code> = 3

3.4.4: After Merge

Root List	Min Node	Node count
[2]<->[8]<->[5]	2	3

Proof of Correctness

Proof of Correctness for Binary

The merge function in the Binary Heap works correctly because it inserts each element from the other heap into the current heap (`self`) using the `insert` method. We know from the heap implementation that `insert(value)` appends the value to the end of the internal list (`self.heap`) and then performs a "bubble up" operation, comparing the inserted value to its parent and swapping if necessary. This process continues until the heap property is restored — that is, every parent is smaller than or equal to its children in a min-heap.

In the merge function:

```
def merge(self, other: "BinaryHeap[T]") -> None:
    """Merges the current heap with another heap."""
    if not isinstance(other, BinaryHeap): # Check if the other heap is an instance of BinaryHeap
        raise HeapError("Can only merge with another BinaryHeap")
    for item in other.heap: # Insert all the elements of the other heap into the current heap
        self.insert(item.value)
```

This line ensures that every value from the other heap is passed through insert, meaning the heap property is re-established after each insertion. Since the Binary Heap's internal structure (a list representing a binary tree) always maintains the heap invariant after insert, the merged heap is guaranteed to remain a valid Binary Heap.

The function also guards against incorrect usage by checking:

```
if not isinstance(other, BinaryHeap):  
    raise HeapError("Can only merge with another BinaryHeap")
```

This ensures we only attempt to merge valid heap structures. So, by relying on the correctness of insert, and ensuring we process each value individually, we can confidently say the merged heap is correct.

Proof of Correctness for Fibonacci

The merge function in the Fibonacci Heap is correct because it leverages the Fibonacci Heap's lazy structure, where the root lists of trees can be combined without immediate restructuring. This enables an O(1) time merge operation, which is one of the key advantages of Fibonacci heaps.

The function starts by verifying that the input is a valid FibonacciHeap, and exits early if the other heap is empty:

```
if not isinstance(other, FibonacciHeap):  
    raise TypeError("Can only merge with another FibonacciHeap")  
  
if other.min_node is None: # If the other heap is empty, return  
  
    return
```

If the current heap (self) is empty, the function simply adopts the other heap's structure:

```
if self.min_node is None: # If the current heap is empty, set the minimum node to the other heap's  
minimum node  
  
    self.min_node = other.min_node  
  
    self.num_nodes = other.num_nodes  
  
    return
```

This is valid, since a Fibonacci Heap with no nodes can become another heap just by pointing to its min_node and copying its node count.

If both heaps are non-empty, the root lists are joined using `_concatenate(self.min_node, other.min_node)`, which links their circular doubly-linked lists together. Because the Fibonacci Heap allows multiple unstructured trees in the root list, combining them doesn't violate any structural rules.

After joining the root lists, the function ensures the correct min_node is set by comparing the keys:

```
if other.min_node.key < self.min_node.key: # Update the minimum node if necessary  
  
    self.min_node = other.min_node
```

This comparison explicitly checks if the other heap's minimum is smaller than the current heap's minimum, and updates the `min_node` pointer accordingly. This ensures that `min_node` always points to the minimum element in the combined heap.

Finally, it updates the total node count:

```
self.num_nodes += other.num_nodes
```

Since no tree restructuring is needed at merge time, and all critical references (`min_node`, root list, and node count) are updated appropriately, the function maintains a valid Fibonacci Heap. Consolidation of the root list (which reduces the number of trees by linking roots with the same degree) is deliberately deferred until an `extract_min()` operation, which is where the structure is "cleaned up." Until then, the heap remains logically correct and fully functional.

The amortized time complexity of this merge operation is $O(1)$, as it only involves pointer adjustments and basic arithmetic operations, with no tree restructuring or traversal required.

Stress Tests

The **bulk merge test** (`generic_stress_test_merge_bulk`) evaluates the heap's ability to correctly merge with another large heap containing a significant number of elements. In this test, two separate heaps are created and populated with random integers, then one heap is merged into the other. After the merge, the resulting heap is validated to ensure it maintains a proper heap structure. This test is important because it simulates common real-world use cases—such as in priority queue operations or graph algorithms—where large heaps may need to be combined. It helps catch errors like lost elements, incorrect tree consolidation, or structural issues, especially in more complex data structures like Fibonacci heaps.

The **empty merge test** (`generic_stress_test_merge_empty`) checks how the heap handles merging when one of the heaps is empty. It tests both directions: merging an empty heap into a populated one, and merging a populated heap into an empty one. These are edge cases that can easily be overlooked during development but can lead to crashes or inconsistent behavior if not handled correctly. The test ensures that empty heaps do not cause exceptions, that the populated heap retains all its elements, and that the resulting heap remains valid. This is particularly important in systems that perform incremental merges or initialize with empty structures.

The **duplicate merge test** (`generic_stress_test_merge_duplicates`) verifies that the heap can handle the merging of two heaps that contain many identical values. Both heaps are filled with the same value (e.g., 500), and then merged. This test is designed to reveal issues related to value comparison logic, element uniqueness assumptions, or duplicate key handling, especially in heaps that use value-based comparisons for ordering. Ensuring that heaps can merge correctly even with duplicates is essential for robustness, as real-world data often includes repeated priority values.

Worst-Case Time Complexity

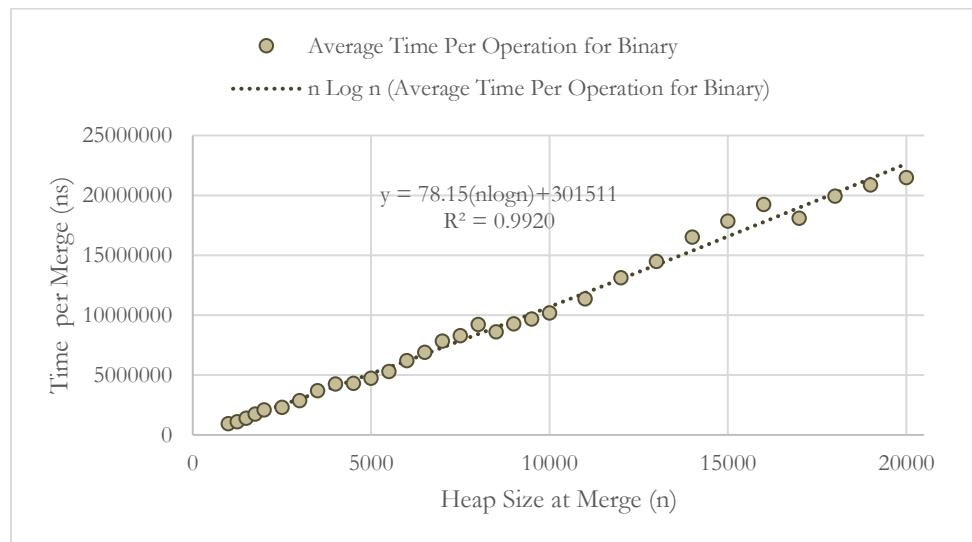
The worst-case time complexity for merging two Binary Heaps is $O(m \log n)$, where m is the number of elements in the heap being merged (i.e., the other heap), and n is the size of the current heap (`self`) before the merge. This is because the merge is implemented by inserting each element from the other heap into the current heap one at a time. Each insertion requires placing the new element at the end of the heap's internal

array and then performing a bubble up (heapify-up) operation to restore the heap property. In the worst case, this bubbling may traverse up to the root, which takes $O(\log n)$ time per insert. As a result, the total time to merge m elements is $O(m \log n)$. This makes Binary Heap merging relatively inefficient for large heaps.

In contrast, the merge operation in a Fibonacci Heap has a worst-case time complexity of $O(1)$. This efficiency is achieved by simply concatenating the circular doubly linked root lists of the two heaps. No reordering or restructuring of the heap is needed at the time of merge; instead, all structural adjustments are deferred to later operations such as `extract_min`. Since merging only involves updating a few pointers and comparing two `min_node` values, the time taken does not depend on the number of elements in either heap. This makes Fibonacci Heaps exceptionally well-suited for applications with frequent merges, such as graph algorithms like Dijkstra's or Prim's.

Benchmark Results

Figure 3.4.1: Average Time Per Merge for Binary (Worst Case)

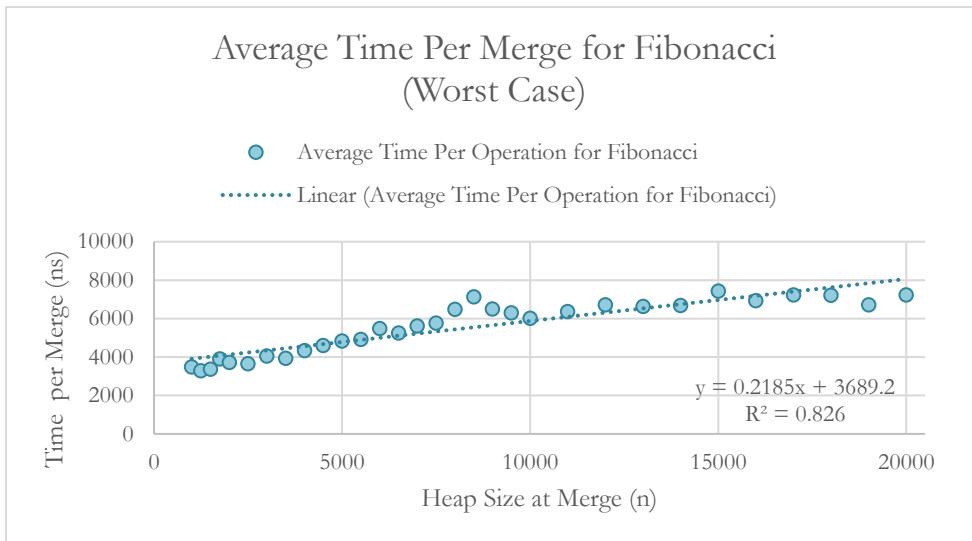


The empirical performance for Binary Heap's merge operation demonstrates strong superlinear growth consistent with its theoretical $O(m \log n)$ time complexity. The measured values (olive green) rise rapidly from under 1 million nanoseconds to over 21 million nanoseconds as n increases from 1'000 to 20'000, indicating substantial cost as input size grows.

The fitted model, $y = 78.15(n \log n) + 301'511$, captures this growth remarkably well, with an R^2 value of 0.9920, showing strong agreement between theory and observation. The curvature of the data also aligns with expectations.

Overall the graph provides strong empirical confirmation that Binary Heap merging scales in accordance with $O(n \log n)$ behavior under worst-case conditions.

Figure 3.4.2: Average Time Per Merge for Fibonacci (Worst Case)

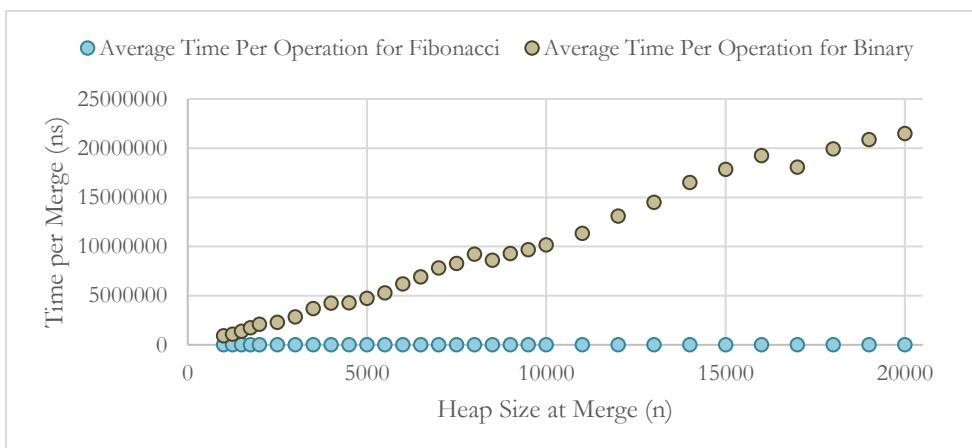


The empirical performance of Fibonacci Heap's merge operation displays near constant behavior with only a modest upward trend, aligning with its theoretical $O(1)$ time complexity. As shown in the graph, the average time per operation increases only slightly – roughly 3'500 nanoseconds to just over 7'000 nanoseconds – as the input size grows from 1'000 to 20'000. This result is consistent with the theoretical $O(1)$ time complexity of Fibonacci Heap merges, where the operation simply concatenates root lists without restructuring, regardless of the heap size.

The fitted linear model $y = 0.2185x + 3689.2$ and an R^2 value of 0.826 suggests that while some noise and overhead exist, the operation remains consistently lightweight in practice.

Overall this empirical data supports the theoretical claim that Fibonacci Heap merge operations are $O(1)$ in worst case.

Figure 3.4.3: Average Time Per Merge for Comparison (Worst Case)



The side-by-side comparison of merge performance in Binary and Fibonacci reveals a contrast in scalability. As shown in the graph, the average time per merge in Binary Heaps (olive green) rises significantly with increasing heap size – reaching over 21 million nanoseconds by size 20'000. In contrast, the Fibonacci Heap

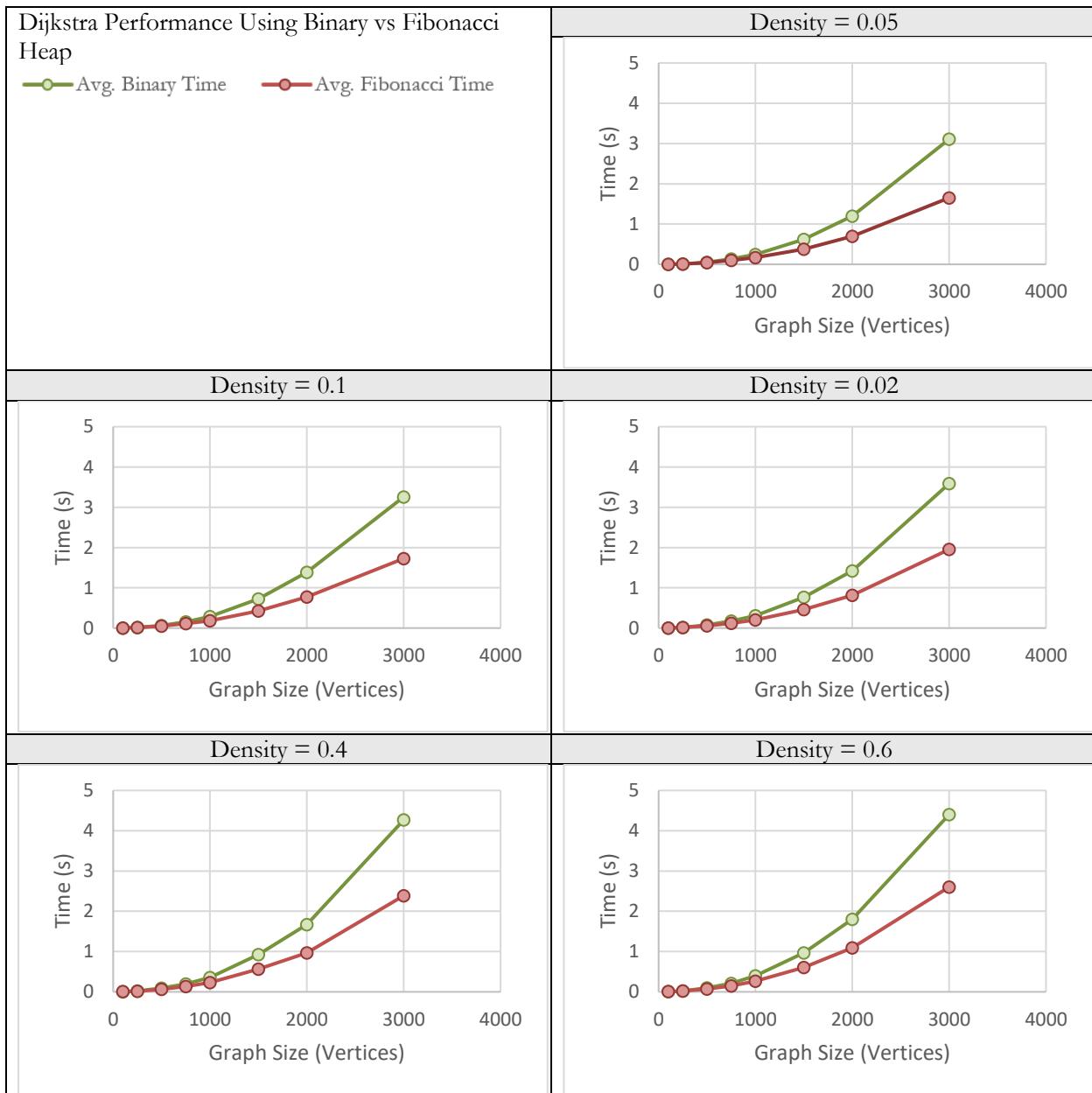
(blue) maintains a consistently low runtime, hovering between 3'000 and 7'000 nanoseconds regardless of heap size.

Overall, the benchmarks empirically confirm the theoretical expectations: Fibonacci Heaps offer significantly better scaling performance for merging operations, especially as input sizes grow, while Binary Heap merge times increase with input size due to their reliance on repeated insertions. This makes Fibonacci Heaps a superior choice in applications where frequent merges are needed.

4. Conclusion and Future Work

To demonstrate the practical implications of our heap performance analysis, we extend our evaluation to Dijkstra's algorithm. Our implementation of Dijkstra's algorithm was designed to use all the operations we have talked about – insertion, extraction, decrease-key, and merge – within the priority queue. In our implementation, the priority queue holds (distance, vertex) tuples, and rather than updating an existing entry when a vertex's tentative distance decreases, we simply insert a new pair and rely on lazy deletion to ignore outdated entries. This approach forces repeated evaluation of the merge functionality while ensuring the heap remains efficient. Our code uses two distinct heap implementations—a Binary Heap, which relies on a dynamic array and a position map to maintain index mappings, and a Fibonacci Heap, which uses circular, doubly linked lists to allow constant-time merges and amortized constant-time decrease-key operations. In addition to runtime measurements, memory usage data was captured in our integrated benchmarking environment. A driver script repeatedly executed both worst-case and Dijkstra's algorithm benchmarks, ensuring that memory and timing metrics were consistently recorded across multiple runs with controlled random seeding.

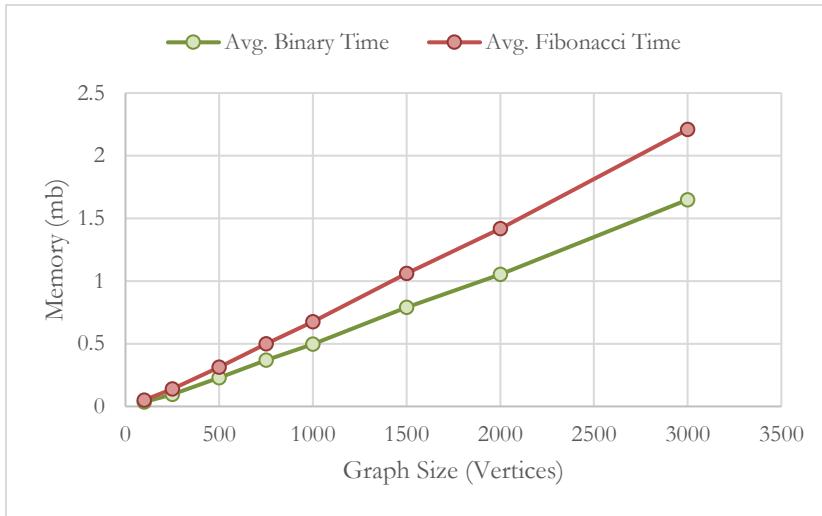
Figure 4.0.1: Dijkstra's Algorithm Runtime by Heap Type and Graph Density



Across all densities, we observe a consistent pattern: the Fibonacci Heap outperforms the Binary Heap in both low and high-density graphs, particularly as input size increases. While performance is comparable at lower sizes and sparse graphs, the gap widens significantly in larger or denser graphs where decrease-key and merge operations dominate execution time. This aligns with theoretical expectations, given that the BinaryHeap's decrease-key runs in linear time in our implementation, compared to the amortized constant time in FibonacciHeap.

While execution time is often the primary concern in algorithmic benchmarking, memory usage also plays a critical role – especially in constrained environments. To complement our run-time evaluation, we captured peak memory usage for both types during Dijkstra's algorithm execution.

Figure 4.0.2: Average Peak Memory Usage vs. Graph Size (Averaged over Densities)



Notably, our Binary Heap, which leverages a list-based position mapping and slots in its `HeapItem`, consistently consumes less memory than Fibonacci Heap. This observation confirms that the extra structural overhead of the Fibonacci Heap – necessary for its amortized $O(1)$ decrease-key and constant-time merge operations—incurs a higher memory cost, even though it delivers superior runtime performance in key-intensive scenarios.

Together we can conclude that there is a fundamental trade-off between time and space efficiency in heap implementations. For applications that demand rapid updates and frequent key adjustments, the Fibonacci Heap’s design offers substantial performance gains at the expense of increased memory consumption. In contrast, Binary Heap provides a leaner memory footprint, making it advantageous memory-constrained contexts.

Future work could explore hybrid approaches or further optimize these data structures – perhaps by implementing them in lower-level languages – to better balance these trade-offs and extend their practical applicability across a wider range of real-world tasks.

5. References

- [1] Fredman, M. L., & Tarjan, R.E. (1987). Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM (JACM)*, 34(3), 596-615.
- [2] Williams, J. W. J. (1964), Algorithm 232- Heapsort, *Communications of the ACM* 7 (6): 347-348
- [3] Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson Education.