Adjacent Ordered Heap: An Extremely Flat Tree Structure That Requires No Balancing

JuFan Wang[[1]](#footnote-1)\*  
Independent Researcher, Qingdao University of Science and Technology Tianjin China, [wangjufan@126.com](mailto:wangjufan@126.com)

The heap data structure is a foundational element in computer science, with binary heaps and Fibonacci heaps serving as pivotal contributions to the field. Despite decades of research yielding numerous important achievements, a fundamental challenge persists: the inefficiency of the delete-min operation and its associated stability issues continue to pose significant bottlenecks. In this paper, we introduce a novel heap structure—the Adjacent Order Heap (AOP)—which extends the partial order relation between parent and child nodes to include relationships among sibling nodes. The AOP achieves an exceptionally flat tree structure that requires no balancing, leveraging arrays for efficient management of tree root nodes and employing a binary heap to facilitate batch consolidation. The amortized time complexity of its four core operations—insert, decrease-key, increase-key, and delete-min—achieves , where represents the average number of sibling nodes. Notably, depends solely on the array capacity rather than the heap size N. Our benchmarks on synthetic datasets reveal significant performance advantages of the AOP compared to pairing heaps, Fibonacci heaps, and d-ary heaps, particularly in the delete-min operation.

CCS CONCEPTS • Theory of computation → Design and analysis of algorithms → Data structures design and analysis

**Additional Keywords and Phrases:** Heap, Partial order relation, Time complexity, Scalability

1. Introduction

Given a set of elements, if a parent-child restriction relation exists among them without forming any cycles, and there is a unique node from which every other node in the set is reachable, then the data structure defined by these elements and their relations is a tree. In computer science, it is common to augment such structures with additional attributes and constraints to develop novel tree data structures with specialized properties.

Assign a key to each node in the tree such that the keys of parent and child nodes satisfy a partial order relation (either ≥ or ≤). Under this constraint, the tree is specialized into a heap. It has an irreplaceable position in algorithm design, system optimization and interdisciplinary applications, and is one of the core basic algorithms in computer science.

The primary operations [1,2,3] supported by a heap include meld, find-min, insert, decrease-key, increase-key and delete-min. When the partial order relation is ≥, the heap is a max-heap, whereby the key of each parent node is greater than or equal to that of its child nodes. Conversely, when the partial order relation is ≤, the heap is a min-heap, in which the key of each parent node is less than or equal to that of its child nodes.

A heap may comprise one or more trees. For instance, pairing heaps [3], binary heaps [4] and d-ary heaps [5] consist of a single tree, whereas Fibonacci heaps [2] are composed of multiple trees. The partial order relation among nodes in a heap can be maintained either by adjusting indices or by rearranging the elements themselves. Heaps represented as a single tree typically employ contiguous storage and enforce order constraints by moving element contents. This method improves spatial locality, thereby enhancing operational efficiency—an effect that is especially significant when the heap contains a relatively small number of nodes. Heaps such as pairing heaps and Fibonacci heaps utilize non-contiguous storage and maintain order constraints through indices (such as pointers). Although this approach sacrifices the benefits of spatial locality, it eliminates the need to copy or move element contents, thereby offering greater flexibility for algorithmic optimization.

Binary heaps do not support direct key modifications. Their operational efficiency primarily stems from the locality of memory access. However, as the heap size increases, the overhead of maintaining the tree's structural balance gradually diminishes this advantage. In contrast, heap structures that utilize non-contiguous storage offer greater flexibility through index-based operations; however, they sacrifice the benefits of spatial locality and do not fully leveraged the computational advantages of modern hardware architectures, such as extremely high CPU frequencies and their integrated computational capabilities.

1. Related Work

In the standard comparison model, the amortized time complexity for priority queue operations maintains perdelete-min operation for both Fibonacci heaps and binary heaps. Although this logarithmic upper bound can be surpassed by employing more advanced operations beyond binary comparisons, the computational models underlying these operations often impose significant restrictions, thereby limiting their practical applicability [5,6].

In 1964, J.W.J. Williams first systematically introduced the binary heap data structure [4]. This pioneering work established the theoretical foundation for modern research on heap algorithms. The design of binary heap is both simple and effective, serving as the basis for numerous algorithms in computer science. A binary heap is a complete binary tree that enforces a partial order relation between parent and child nodes. To maintain the heap order property, a binary heap does not require the complex dynamic adjustments characteristic of self-balancing trees [3]. The binary heap employs a contiguous storage model, with elements stored in an array and accessed via indices. While the array is contiguous, elements may move within the array during heap operations to maintain the heap property and ensure the correct structure. The binary heap does not natively support efficient key modification operations; such changes require locating the element with an time complexity of , where is the number of items in the heap. The d-ary heap, a generalization of the binary heap, similarly maintains tree balance by moving element values and can support key modifications with efficiency comparable to that of Fibonacci heaps and pairing heaps.

To enable efficient key modifications, Fredman and Tarjan introduced the Fibonacci heap in 1987 [2]. The amortized time complexity [8] for delete-min and increase-key operations in a Fibonacci heap is , while the other two core operations achieve an amortized time complexity of The Fibonacci heap employs a pointer-based, discrete memory model but still requires maintaining certain tree balance properties. While its decrease-key operation delivers optimal performance, the frequent restructuring necessitated by its operations, coupled with inherent constant factors in its implementation, often leads to significantly slower performance and limited scalability, thereby rendering optimization particularly challenging.

As a pointer-based data structure, Fibonacci heaps sacrifice cache locality and incur additional overhead. Decrease-key and delete-min operations in Fibonacci heaps can trigger extensive cascading cuts and restructuring, which can disrupt locality of reference and hinder efficient memory access through direct indexing. Maintaining certain structural properties—such as bounding the degree of nodes—is essential for achieving amortized time complexity guarantees. However, compared to the resources required by these structural maintenance mechanisms, their contribution to the fundamental heap order property is often insufficient.

In traditional heap data structures, maintenance operations are generally regarded as necessary overhead incurred to preserve structural invariants. For example, binary heaps maintain heap invariants through sift-up and sift-down operations, which swap elements within an array. While these operations incur computational costs, their primary objective is to ensure the correctness and efficiency of the most recently executed heap operations. However, the data structure does not inherently preserve partial order relation or correctness in a robust manner. Maintenance operations—such as those enforcing structural invariants—often modify the structure solely for correctness, rather than to support specific operations like insertion, key updates, or delete-min. Consequently, these adjustments introduce overhead without optimizing future operations, leaving little to no opportunity for performance improvements.

Fibonacci heaps hold greater theoretical significance than practical impact [1,2]. They paved the way for further advances in heap data structures, and since their introduction, numerous heap implementations [9–18] with diverse characteristics have been proposed, including several variants achieving sub-logarithmic time complexities [19–23]. These new heap algorithms aim to match the time complexities of Fibonacci heaps while being simultaneously simpler and more efficient in practice. Whether a new heap algorithm can achieve the theoretical time bounds of Fibonacci heaps has become a standard criterion for evaluating such data structures. However, whether the time bound for any operation in heap data structures can be achieved below within the pointer machine model remains an open question, driving research into new heap structures.

1. Problem Definition

The underlying tree structures of these two heap implementations differ significantly: the binary heap enforces a strict complete binary tree invariant, whereas the Fibonacci heap consists of a flexible forest of min-heap-ordered trees that are neither necessarily complete nor strictly balanced. A common characteristic shared by these heap variants is the maintenance of a partial order relation between parent and child nodes. This inherent single-dimensional order constraint, along with the requirement to maintain a certain level of balance—such as limiting the number of children per node in Fibonacci heaps—fundamentally restricts the time complexity of existing heap operations, preventing them from exceeding the upper bound of .

To overcome this limitation, a more efficient heap data structure must incorporate multidimensional order constraints. This approach enables the algorithm to eliminate any balance requirements traditionally imposed on tree nodes, thereby leveraging structural invariants induced by the multidimensional order constraints to surpass the amortized time complexity upper bound of . Critically, the structural invariant—combined with the use of arrays for efficient management of tree root nodes and the tree batch consolidating mechanism—ensures scalability while forming the foundation for the algorithm's computational efficiency.

This paper extends the fundamental heap property—the partial order relation between parent and child nodes—to encompass child nodes themselves, and proposes a high-performance heap data structure termed theAdjacent Ordered Heap (AOH). AOH manages the root nodes of the trees via an array, complemented by a binary heap that facilitates efficient batch consolidation of trees. Leveraging array-based management of root nodes can effectively reduce and defer memory access overhead in its four core operations. Furthermore, the binary heap-assisted batch consolidation accelerates tree consolidation by substantially increasing the number of sibling nodes while simultaneously reducing tree height. By leveraging longer brother chains—where the tree height no longer restricts the number of nodes it can accommodate—this approach enhances the structural order of heap elements, creating additional opportunities to improve algorithmic performance.

The use of arrays allows AOH to fully harness the capabilities of modern hardware, particularly its extremely high clock speeds. By imposing no restrictions on the number of child nodes and maintaining order among linked nodes, memory access is restricted to a very narrow range (a few memory locations closely related to the current operating node), resulting in a significant reduction in access frequency and leading to exceptional performance efficiency. By explicitly addressing the dual challenges of computational efficiency and memory access, AOH achieves a significant reduction in the time complexity of its four core heap operations. The integration of an auxiliary array and a binary heap, combined with a tree consolidating technique that constructs a bidirectional linked list via bro and pre pointers, transforms maintenance operations from mere overhead for preserving its structural invariants into a foundation for further heap optimization.

The core logic of AOH is intuitive and straightforward, facilitating ease of understanding, implementation, and maintenance. Notably, it eliminates the traditional requirement for heaps to maintain tree structures that rely on a certain degree of balance. This innovation enables AOH to surpass the conventional logarithmic upper bound on time complexity while preserving the simplicity of fundamental operations. The following sections provide a detailed discussion of the design and implementation of AOH, which adheres to the partial order relation ≤ using algorithmic steps.

1. Alogorithm Design
   1. Data Structure

Our novel tree representation utilizes three fundamental pointers (*pre*, *next*, *bro*) for tree node linkage.

**The pre pointer is a dual-function pointer that connects to either the parent node or the left sibling of the current node, as a node can have at most one such connection through the pre pointer. This design enforces the partial order constraint ∀x ∈ {parent, left sibling}, key(x) ≤ key(current) in the reverse direction. The** next **pointer** points to the child node with the smallest key among all child nodes of the current node, preserving parent-child ordering, key(current) ≤ key(child). **The** bro **pointer connects to the right sibling of the current node,** forming sibling chains while preserving the monotonicity condition key(current) ≤ key(**right** sibling).

The manuscript employs these consistent naming conventions:

Let *T* denote a tree structure.

Let ***X*** denote a key or a node.

Let *P* denote a pointer to a node.

Let pre(*T/X/P*), next(*T/X/P*), bro(*T/X/P*)denote the pre, next, or bro pointer of node T/X/P.

Let location(*T/X/P*)denote the position of node *T/X/P* in an array.

Node *P* refers to the node pointed to by pointer *P***, and vice versa**. Key *P* refers to the key of the node pointed to by pointer *P***, and vice versa**. For tree *P*, pointer *P* references the root node of the tree**, and vice versa**. **If X is a key, we can refer to it as node *X* when there is no** ambiguity**, and vice versa. Similarly, if *X* is a key or a node, we can refer to it as tree *X* (a tree rooted at node *X*) when there is no ambiguity, and vice versa. Thus, node *T* means the root node of tree *T*, or key *T* means the key of the root node of tree *T*, when there is no ambiguity. Therefore, pointer, key, node, and tree achieve a certain level of uniformity.** When storing a node or key, the actual stored value is the pointer to the node. When storing a tree, the stored value is the pointer to its root node.

In Figure 1, Key/Node/Pointer/Tree 1 has child nodes 2 and 3, and both its pre and bro pointers are NULL (pre(1) == NULL, bro(1) == NULL). Node 1 is the root node of the tree and points to child node 2 through the next pointer (next(1) == key/node/pointer/tree 2). There is no direct pointer connection between node 1 and its child node 3; the partial order relation is maintained through node 2. Node 2 has no child nodes. It points to its parent node 1 through the pre pointer, where pre(2) == node 1, and it points to its right sibling node 3 through the bro pointer, where bro(2) == node 3. Node 3 has no sibling nodes and child nodes with smaller key; both its next and bro pointers are NULL, next(3) == bro(3) == NULL. It points to its left sibling node 2 through the pre pointer, pre(3) == node 2.

In Figure 1, there are a total of three possible trees. Tree 1 consists of three nodes: nodes 1, 2, and 3. Disconnecting the links between nodes 1 and 2 results in Tree 2, which contains two nodes: nodes 2 and 3. Disconnecting the links between nodes 3 and 2 yields Tree 3, which has only one node. In this paper, when there is no ambiguity, Tree 2 and Tree 3 can be referred to directly, thereby omitting the description of the link management.

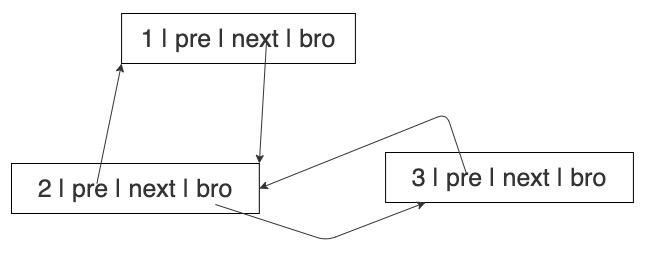


Figure 1: A standard subtree with three nodes

In our tree classification hierarchy, the following properties hold invariantly:

1. The pre(*T*) is guaranteed to be NULL.
2. A node forms a *Trivial-Subtree* *T* if , , and (indicating that all its pointers—pre, next, and bro—are NULL).
3. A *Standard-Subtree* *T* represents the canonical form, characterized by . This general form includes the Trivial-Subtrees as degenerate cases where all pointers (pre, next, bro) are NULL.
4. Additionally, we define a *B-Subtree T*as a tree structure whose root node satisfies the following conditions: (1) (the next pointer is NULL); (2) (the bro pointer is non-NULL).
5. A *Subtree-Group* *T* denotes a tree whose root node maintains active connections through both its next and bro pointers，specifically and .

Throughout all figures (including Figures 2-3), our diagrammatic convention employs: Horizontal double-headed arrows represent pre and bro pointer connections. Vertical double-headed arrows indicate pre and next pointer relations. This consistent visual encoding applies universally across all illustrations in this work.

By strategically breaking the next–pre pointer pairs at the root nodes of the trees, any subtree-group can be systematically decomposed into two distinct components: a well-formed B-Subtree and an associated residual tree, which may be a Standard-Subtree, a B-Subtree, or another Subtree-Group. Standard-Subtrees, B-Subtrees and Subtree-Groups are collectively called trees. These three categories collectively encompass all valid tree structures in this paper.

In Figure 2, Subtree-Group 3 is split into B-Subtree 3 and a residual tree 4 by breaking the next-pre pointer pairs between node 3 and node 4. Through a straightforward pointer exchange operation at the root node, any B-Subtree can be equivalently represented as a Standard-Subtree, thereby unifying the tree representation. In Figure 3, by exchanging the bro and next pointers of node 3, B-Subtree 3 is transformed into Standard-Subtree 3.

* 1. Merged-Height

The minimum number of Standard-Subtrees and B-Subtrees produced during tree decomposition—achieved through the decomposition of a Subtree-Group—is formally defined as the tree's *merged-height*. The merged-height for both Standard-Subtrees and B-Subtrees is defined as 0. Each next-pre pointer pair contributes exactly one unit to the merged-height calculation, while bro-pre pointer pairs maintain merged-height neutrality.

The decomposition of a Subtree-Group involves separating it into a B-Subtree and a residual tree by severing the next-pre pointer pair between the root node of the Subtree-Group and its child. If the resulting residual tree remains a valid Subtree-Group, the process is repeated iteratively. The merged-height H(Subtree-Group)satisfies the recurrence relation:

(1)

In Figure 2, the merged-height of the Subtree-Group 3 is 1; the merged-height of the B-Subtree 3 is 0; the merged-height of the residual tree 4 is 0. In Figure 3, the merged-height of the Standard-Subtree 3 is 2.

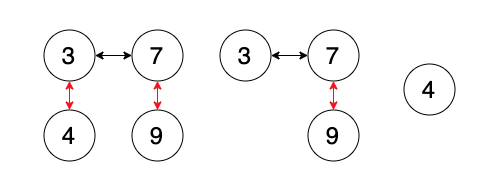


Figure 2: Transformation of Subtree-Group 3 into B-Subtree 3 and residual tree 4

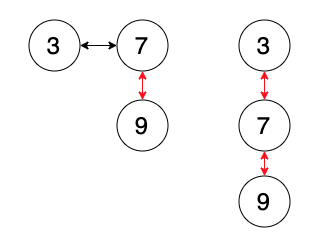


Figure 3: B-Subtree 3 is converted to Standard-Subtree 3

* 1. Operations

AOH data structure consists of four core components: (1) an array with capacity *L* storing the root nodes of trees, (2) two critical indices - *MinPtr* marking the minimum-key node and *FreePtr* indicating the next available slot, (3) a collection of trees, and (4) a binary heap as a consolidation-auxiliary structure.

AOH's operational logic centers on two fundamental mechanisms: (1) dynamic array maintenance through continuous updates of *FreePtr* (tracking available slots) and *MinPtr* (identifying the current minimum-key node), and (2) on-demand tree consolidation via the binary heap. The find-min algorithm simply returns the element indexed by *MinPtr*.

The construction of an empty AOH involves two critical steps:

ALGORITHM 1: Make-Heap

Reserve an array ARR[0..L-1] // Storage for root nodes of L trees

FreePtr ←0, MinPtr ← -1 // Initialize FreePtr to 0

To insert tree *T*, the algorithm stores *T* （a pointer to the root node of tree *T*） in the *FreePtr* slot of ARR. The efficiency of insert operation is determined by the amortized time complexity of tree consolidation. The meld operation of two heaps is carried out through tree insert operations. The procedure for the insert operation is outlined below:

ALGORITHM 2: Insert

ARR[FreePtr] ← tree T // Store tree T in array ARR at index pointed to by FreePtr

IF key T < key ARR[MinPtr] // key T is less than current minimum

MinPtr ← FreePtr

ENDIF

FreePtr++ // Advance FreePtr to next available memory location

IF FreePtr == L // L denotes the capacity limit of array ARR

Consolidate the L trees into a single tree T1

FreePtr  ← 1, MinPtr ← 0

ARR[MinPtr] ← tree T1

ENDIF

When key *X* is changed (either increased or decreased), the partial order relation between parent and child, or among siblings, may be disrupted. Adjustments are required to ensure that the partial order relation between parent and child, as well as among siblings, comply with the invariants of AOH. Various implementation approaches can be applied to achieve this. For example, one approach is to prioritize disconnecting node *X* with decreased key from node pre(*X*) while preserving its link with nodes bro(*X*) and next(*X*). The steps for implementing the key decrease algorithm are outlined as follows:

ALGORITHM 3: Decrease-Key

IF pre(X) == NULL && key X < key ARR[MinPtr]

MinPtr ← location(X)

ENDIF

IF pre(X) ≠ NULL && key X < key pre(X) && node next(pre(X)) == node X

next(pre(X)) ← NULL

pre(X) ← NULL

Insert tree X into ARR

ENDIF

IF pre(X) ≠ NULL && key X < key pre(X) && node bro(pre(X)) == node X

bro(pre(X)) ← NULL

pre(X) ← NULL

Insert tree X into ARR

ENDIF

For example, one approach prioritizes disconnecting node X with increased key from node bro(*X*) and node next(*X*) while preserving its links with node pre(*X*). The steps for implementing the key increase algorithm are outlined as follows:

ALGORITHM 4: Increase-Key

IF bro(X) ≠ NULL && key bro(X) < key X

pre(bro(X)) ← NULL

bro(X) ← NULL

Insert tree bro(X) into ARR

ENDIF

IF next(X) ≠ NULL && key next(X) < key X

pre(next(X)) ← NULL

next(X) ← NULL

Insert tree next (X) into ARR

ENDIF

IF location(X) == MinPtr

update MinPtr to point to the minimum node in the heap

ENDIF

The partial order relation remain unchanged when the minimum node X is deleted. This operation is straightforward, as it simply involves removing the node, reinserting the resulting tree, and updating the *MinPtr*. The minimum node delete operation implements the following procedure:

ALGORITHM 5: Delete-Min

Remove node X from the heap structure

IF bro(X) ≠ NULL pre(bro(X)) ← NULL ENDIF

IF next(X) ≠ NULL pre(next(X)) ← NULL ENDIF

IF bro(X) ≠ NULL && next(X) != NULL

ARR[MinPtr] ← tree bro(X)

insert tree next(X) into ARR

ENDIF

IF bro(X) ≠ NULL && next(X) == NULL

ARR[MinPtr] ← tree bro(X)

ENDIF

IF bro(X) == NULL && next(X) != NULL

ARR[MinPtr] ← tree next(X)

ENDIF

IF bro(X) == NULL && next(X) == NULL

FreePtr--

ARR[MinPtr] ← tree ARR[FreePtr]

ENDIF

Update MinPtr to point to the minimum node in the heap

* 1. Consolidation

The data structure exhibits time complexity for all operations when the array contains fewer than *L* elements. Beyond this threshold, the amortized time complexity of its four core operations is governed by the cost of tree consolidation. When *FreePtr* reaches the array capacity (*FreePtr* == *L*), the implementation comprises these stages:

ALGORITHM 6: Consolidation

Create an empty binary heap AHP

MinPtr ← 0, FreePtr ← 1

Sort ARR in ascending order according to keys of root nodes

bro ← nextMin(AHP, ARR) // retrieve a Standard-Subtree with the minimum root key

WHILE bro ≠ NULL

min ← nextMin(AHP, ARR)

bro→bro ← min // assign bro pointer of node bro to tree

tree→pre ← bro

bro ← min

ENDWHILE

To efficiently consolidate trees, the following procedure is applied when extracting the Standard-Subtree with the minimum root key from the binary heap AHP and the array ARR: (1) If the extracted tree is a B-Subtree, convert it directly into a Standard-Subtree; (2) If the extracted tree is a Subtree-Group, first extract a B-Subtree from the group, then transform this B-Subtree into a Standard-Subtree while reinserting the remaining structure back into the binary heap AHP.

The algorithm achieves optimal time complexity through three key mechanisms: (1) maximizing data locality through coordinated use of array and binary heap structures, (2) leveraging hardware acceleration (when available) by increasing the parameter*L*, and (3) maintaining a bidirectional linked list via the bro and pre pointers. These optimizations yield dual benefits through improving theoretical time complexity bounds and enhancing practical runtime performance. The data locality optimization reduces cache misses through two complementary strategies: (1) employing compact array structures to enhance spatial locality and leverage the computational advantages of modern hardware, and (2) utilizing small-scale binary heaps to enable efficient, localized memory access. This approach is particularly effective for large-scale datasets, where traditional methods—used to maintain tree balance and heap order—often suffer from poor spatial locality.

When specialized hardware (such as an L-way extremum selector) is available, the algorithm can further improve the performance of the four core operations through parallel subtree processing and hardware-optimized comparison operations. After obtaining a Standard-Subtree, the algorithm employs an efficient sibling chain management strategy through bidirectional linking. Upon Standard-Subtree returned, it dynamically constructs and maintains an optimized sibling hierarchy via bro and pre pointers. The complete linkage of sibling pointers ensures maximal connectivity while avoiding unnecessary structural adjustments.

This combined approach achieves an optimal balance between memory efficiency and computational performance, which is particularly advantageous for large-scale priority queue implementations. Moreover, it ensures scalable behavior and robust performance across diverse hardware configurations and dataset sizes. The efficiency of four core operations fundamentally depends on the tree consolidation mechanism. This process serves as the computational foundation governing the algorithm's time complexity and practical performance. The maintenance operations of conventional heaps fundamentally suffer from poor locality due to their chain-processing paradigm, with two canonical examples: (1) In Fibonacci heaps, cascading cuts necessitate recursive parent traversals, as theoretically proven by Fredman and Tarjan [2]; (2) In pairing heaps, multi-pass merging inherently induces sequential node-pair processing, as both experimentally and analytically demonstrated in [20,23].

The innovation transforms what was traditionally considered mere overhead into the main driver of both theoretical and practical performance. The key breakthrough is that each structural adjustment actively improves subsequent access efficiency by enforcing a more orderly node organization, allowing maintenance operations to enhance performance rather than merely guaranteeing correctness. The maintenance operations in AOH demonstrate strong locality by decomposing a Subtree-Group into a minimal set of Standard-Subtrees during consolidation, modifying only the currently processed node and its directly linked neighbors. This localized behavior: (1) forms the foundation of the algorithm's stability and computational efficiency, and (2) progressively improves data quality for subsequent operations. Increased tree consolidation operations enhance both data organization and algorithmic efficiency, thereby elevating structural maintenance operations to fundamental optimization opportunities.

This work establishes a fundamental shift in priority queue architecture, redefining structural maintenance as an active performance accelerator rather than a passive overhead. It transforms tree consolidation into an active process where each tree consolidation contributes to improved amortized time complexity and optimizes real-world execution profiles.

* 1. Tree Consolidation Example

Let *L*=4 to concretely demonstrate the tree consolidation process:

Figure 4 illustrates four trees: 1, 3, 0, and 8 (left to right).

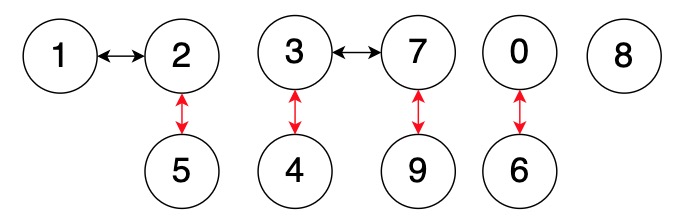


Figure 4: Four unordered trees

Figure 5 shows the four trees from Figure 4 after sorting by root key in ascending order (left-to-right). The trees are stored consecutively in the array at indices 0 through 3, with Tree 0 at index 0, Tree 1 at index 1, Tree 3 at index 2, and Tree 8 at index 3.

Figure 6 presents the consolidation result of Tree 0 and Tree 1. The Standard-Subtree 0 and Standard-Subtree 1 (converted from B-Subtree 1 from Figure 5) are selected based on minimal root keys. The pointer of node 1 is assigned to the bro pointer of node 0 (bro(node 0) ← pointer 1), and the pointer of node 0 is assigned to the pre pointer of node 1 (pre(node 1) ← pointer 0), forming a doubly linked list.

In Figure 7, the Subtree-Group 3 from Figure 6 is decomposed into a B-Subtree 3 and a residual tree 4. The residual tree 4 is inserted into the binary heap, while the B-Subtree 3 is converted into a Standard-Subtree 3. Subsequently, the Standard-Subtree 3 is consolidated into tree 0.

In Figure 8, Standard-Subtree 4 is removed from the binary heap and consolidated into Tree 0. Subsequently, Standard-Subtree 8 (located at array index 3) is also consolidated into Tree 0.

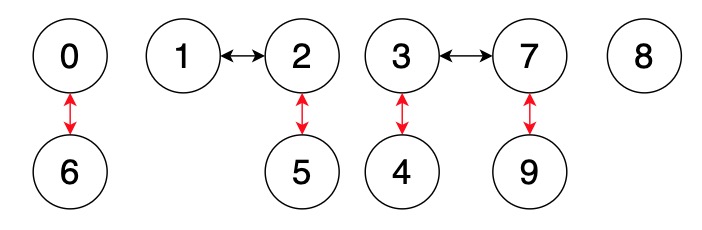


Figure 5: Ascending-order sorted trees

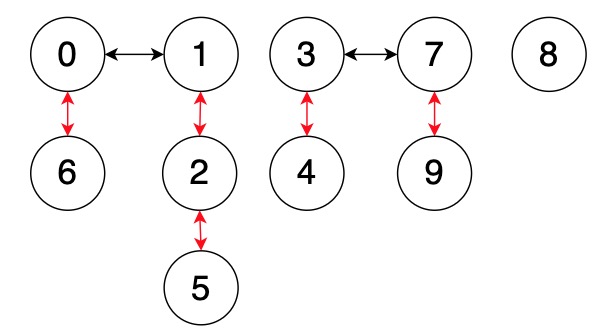


Figure 6: Converting tree 1 into a Standard-Subtree and consolidating it into tree 0

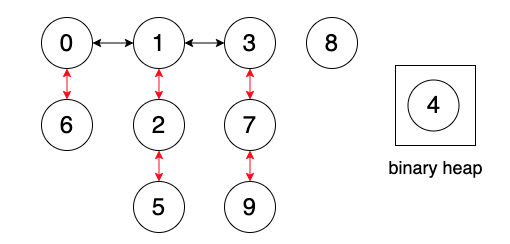


Figure 7: Decomposition and Consolidation of Subtree-Group 3

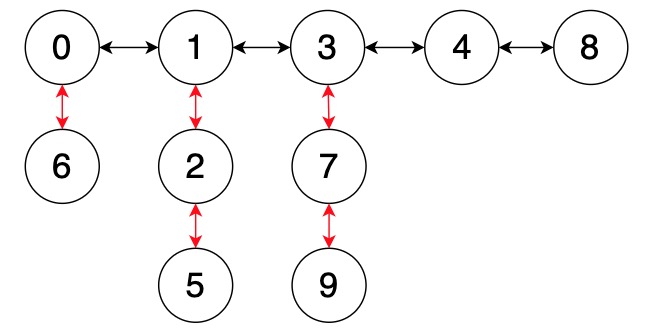


Figure 8: Consolidation of Standard-Subtrees 4 and 8 into Tree 0

The length of the doubly linked list constructed by the Standard-Subtrees through the pre and bro pointers must be at least *L*. This consolidation method significantly enhances the horizontal growth of the trees, reduces its height, and optimizes the algorithm’s time complexity. If *L* becomes too large, the sorting and traversal of the array will consume significant computing resources, creating a bottleneck in processing power. Using Standard-Subtree structures, batch tree consolidation improves data organization while significantly reducing—and in some cases eliminating—content access bottlenecks. Both issues are inherent trade-offs of the algorithm’s design given the current hardware constraints.

AOH must carefully balance the computational overhead incurred by array and binary heap operations with the memory bandwidth requirements imposed by batch consolidation. Reducing the traversal and sorting latency of these data structures allows for scaling up the array capacity, effectively postponing or diminishing the incidence of memory-bound operations.

1. Time Complexity Analysis

The average merged-height is defined as:

(2)

where: ∑(merged-height of trees processed during each consolidation) is the total sum of the merged-height of trees involved in consolidation, number of core operations processed is the total count of core operations carried out during all consolidation processes.

When implementing AOH using arrays and binary heaps, the amortized time complexities of increase-key and delete-min is , and the amortized time complexity of  insert and decrease-key is , where is the time complexity to traverse the array, is the amortized cost of binary heap operations, and is the amortized time complexity of obtaining Standard-Subtrees during consolidation. The minimum extraction operation can achieve amortized time complexity when implemented with dedicated hardware support—for example, using L-way extremum selector hardware modules—eliminating the requirements for either array traversal or binary heap maintenance. Under this hardware-accelerated paradigm, the actual running efficiency of AOH and the amortized time complexity of its four core operations can achieve significant improvements.

**Transformation Axiom.**In AOH, all newly added doubly linked connections are implemented via bro-pre pointer pairs; all next-pre pointer pairs are derived through transformations of bro-pre pointer pairs during consolidation; reverse transformations do not occur.

The links (either the next-pre pointer pair or the bro-pre pointer pair) that are eliminated by the decrease-key and increase-key operations are restored in the subsequent consolidation operation as bro-pre pointer pairs. Consequently, only the following links affect the total link count of the heap: (1) the bro-pre pointer pair count during the consolidation increases by the number of newly inserted trees into the heap, plus the number of link pairs contained in those trees. When the heap is empty, it increases by the number of newly inserted trees into the heap, plus the number of link pairs contained in those trees, minus 1; and (2) all link pairs that are disconnected during the delete-min operation, which collectively decrease the link count by 1. More intuitively, each non-root node corresponds to a link pair, so deleting a node inevitably reduces the number of link pairs by one. During consolidation, when a B-Subtree is identified, its bro-pre pointer pair is converted to a next-pre pointer pair, and thus does not affect the total link count in the heap.

**Reduction Axiom.** In AOH, when ignoring tree consolidations, insert operations preserve the merged-height of trees, while delete-min, decrease-key, and increase-key operations may reduce the merged-height of trees.

In AOH, insert simply appends an element to the array without altering the merged-height of any constituent tree.

During delete-min operations, AOH enforces the following structural transformation for any tree *T*: (1) If *T* is a Subtree-Group, it decomposes into two residual trees; (2) If *T* is a Standard-Subtree or a B-Subtree, it yields a single residual tree. In both cases, the merged-height of the residual trees remain unchanged. The merged height of the tree T reduces to 0. The total merged-height sum of the trees in the heap decreases by 1 if the heap is not empty after the node is deleted. More intuitively, the deleted node corresponds to a unit of merged-height, provided the heap remains non-empty after the deletion.

For decrease-key operations on non-root node *X*, if key *X* becomes smaller than key pre(*X*) and next(pre(*X*)) == node *X* (node pre(*X*) is node *X*’s parent node), then the merged-height of tree pre(*X*) decreases by (1 + the merged-height of tree *X*), equals to 0. In all other cases, decrease-key operations preserve the merged-height of all trees in the heap.

When performing an increase-key operation on node *X*, if the operation requires disconnecting *X* from node next(*X*), the merged-height of tree *X* decreases by (1 + the merged-height of tree next(*X*)), equals to 0.

**Increment Axiom.** In AOH, increases merged-height occur only during consolidation phases, resulting from the transformation of a bro link to a next link. If next(pre(tree T)) == tree T, the merged-heigh of tree pre(T) satisfies:

(3)

According to the Reduction Axiom, when the array is not full, all operations of AOH either reduce the merged-height or have no effect on it. When the number of elements in the array reaches *L*, AOH will split tree T into a B-Subtree and an associated residual tree when tree *T* is a Subtree-Group. The merged-height of the B-Subtree decreased to 0 after the splitting operation. The merged-height of the residual tree remains unchanged after the splitting operation. After the B-Subtree is converted to a Standard-Subtree *T1*, its merged-height increase to (one plus the merged-height of tree next(*T1*)).

**Single-Contribution Axiom.** In AOH, a next-pre pointer pair contributes its merged-height to the time complexity at most once during its lifecycle.

According to the Transformation Axiom, the bro-pre pointer pairs are established during consolidation operations. In subsequent consolidation, some bro-pre pointer pairs are converted into next-pre pointer pairs, while others are removed during delete-min, decrease-key and increase-key operations—with no reverse transformation occurring.

Furthermore, according to the Reduction Axiom and Increment Axiom, the connection, disconnection, and conversion of bro, next, and pre pointers all correspond to specific operational behaviors, and there is no meaningless semantic conversion—such as the transformations used to maintain tree balance in Fibonacci heaps—between the bro and next pointers. During the tree consolidation phase, each disconnected next-pre pointer pair identified for a Subtree-Group contributes one unit to the time complexity.

Notably, only the disconnected next-pre pointer pairs during consolidation influence the algorithm's time complexity. According to Transformation Axiom, next-pre pointer pairs originate exclusively during B-Subtree conversion in the consolidation process. Once generated, these links are ultimately consumed through exactly three scenarios: (1) minimum node deletion, (2) key modification operations, or (3) construction of bro-pre chains during consolidation - the only case where their merged-height contributes to time complexity. This production-consumption dynamic between bro-pre and next-pre pointer pairs serves as the foundation for the algorithm’s efficiency and overall time complexity analysis.

Suppose AOH undergoes *I* consolidation operations, during which the total number of four core operations is *Ops*，the amortized merged-height of the four core operations is . The merged-height of the *j-th* tree in the array of the heap in the *i-th* consolidation is . In the *i-th* consolidation, *L* trees correspond to Standard-Subtrees. This expression represents the number of next-pre pointer pairs broken during the consolidation, where represents the merged-height of tree .

**Equivalence Theorem.** In AOH, Ops \* is equal to the sum of the next-pre pointer pairs broken in I times consolidation operations.

Proof: The sum of the next-pre pointer pair broken in I times consolidation operations is:

(4)

According to Single-Contribution Axiom, the merged-height of a next-pre pointer pair affects the algorithm’s time complexity at most once. Therefore, the sum of the next-pre pointer pair broken in I times consolidation operations equals the sum of the merged-heights of the trees processed in I merge operations. Based on Formula (2) and Formula (4), AOH achieves the following amortized time complexity bounds per core operation:

(5)

The linear combination in Eq. (5) assigns uniform coefficients to the corresponding terms, regardless of their structural differences. The structural units represented by a next-pre pointer pair can be distributed across any tree in the data structure without altering the algorithm's amortized time complexity. The average merged-height of the tree accurately represents the algorithm's amortized time complexity, ensuring that the average time complexity evaluation for each core operation based on remains precise and reliable.

**Distribution-Independence Corollary.** *The amortized time complexity of the four core operations depends solely on the number of next–pre pointer pairs disconnected when decomposing Group-Subtrees, and is independent of the specific trees on which these pointer pairs reside.*

Consider a collection of *L* trees partitionable into Standard-Subtrees, where denotes the average sibling cardinality per node, and *N* represents the total number of heap elements. It holds that . The following relation is established:

(6)

Formula (6) yields a counterintuitive breakthrough, overcoming the well-established complexity barrier. The derivation quantitatively demonstrates how the time complexity decreases monotonically with increasing . The Distribution-Independence Corollary provides the theoretical foundation for justifying the use of as a valid measure of amortized time complexity. The parameter reflects AOH's fundamental design principle that all decomposition trees during consolidation are topologically equivalent. This assumption enables the complexity bound for its four core operations.

AOH eliminates traditional tree-balancing overhead through two key innovations: merged-height tolerance and asymmetric processing. The merged-height tolerance mechanism effectively handles extreme structural variations, accommodating both high-density trees with a merged height of *H* = 0 containing up to O(*N*) nodes, and low-density trees with *H* = *k* > 0 containing only O(1) nodes. This ensures efficient operations are maintained regardless of the distribution of nodes among the trees. Asymmetric processing utilizes next-pre pointer pairs to actively reduce merged-height and flatten the trees, while bro-pre pointer pairs preserve the tree topology without incurring costly restructuring. This strategy eliminates the need for explicit rebalancing and paves the way for further optimizations.

AOH yields flat structures as measured by the average merged-height, regardless of the distribution of merged-heights among the *L* trees. In other words, the particular tree to which a next-pre pointer pair contributes does not influence the overall time complexity. Consequently, the assumption on the parameter is both straightforward and effective, providing a natural and intuitive evaluation of the tree structures.

1. Experimental Results
   1. Environment

The experiment device is a MacBook Pro 16-inch 2019 model, and the system version was macOS Monterey 14.7.4 (23H420). The device is equipped with a 2.6 GHz six-core Intel Core i7 processor and 32 GB 2667 MHz DDR4 memory, providing powerful computing power and smooth multitasking experience. In terms of storage, Macintosh HD was used as the startup disk to ensure stable access to data.

* 1. data set

This study employs systematically constructed datasets to evaluate heap performance across the four core operations. The experimental framework comprises: 5 sets of data sets for insert, delete-min, decrease-key and increase-key operations, and the number of records of the corresponding operation type in each set is 10², 103, 104, 105, 106, 107 and 5\*107 respectively. In the record sets for the decrease-key, increase-key, and delete-min operations, the first half of the records consist of insertions, while the second half comprises key decreases, key increases, or deletions. Therefore, the actual number of record sets for these operations is twice the previously mentioned count. This paper generates (1) the initial keys for heap elements, which are randomly sampled from the range [1,N] (where N is the cardinality of the corresponding record set), and (2) all subsequent key modifications are also randomly generated within the range [1,N]. The generated dataset is classified as "random-type".

* 1. Experimental Methodology

To enable fair comparison, we designed a unified interface that supports three heap implementations from the Boost.Heap 1.88.0 library (Fibonacci heap, pairing heap, and d-ary heap) while maintaining compatibility with AOH. In this study, the parameter *L* was set to 96, and the branching factor d in the d-ary heap was set to 3 throughout our experiments. For efficient element access, we employ a hash table to manage node pointers, enabling O(1) lookup by node identifier.

The benchmarking process iteratively reads and processes batches of 10 million operation records until all data is exhausted. We measure computational performance by precisely recording the clock cycles consumed during execution of each target operation type, ensuring statistically reliable results through repeated trials. For each operation, the testing algorithm is executed twice across five distinct datasets, and the average clock cycle count is computed to ensure the statistical validity of the results.

* 1. Insert

In Figure 9 (a), Fibonacci, pairing and d-ary heaps exhibit particularly pronounced growth rates at larger scales. Although theoretically optimal for insertion, the Fibonacci heap performs worse than both the pairing and d-ary heaps at small scales. The pairing heap maintains a competitive growth rate and demonstrates superior scalability compared to others, especially in smaller datasets. Although Fibonacci performs best at 107, its growth factor is significantly higher than that of AOH, demonstrating that AOH is most stable at extreme scales. The actual clock cycle consumption for the insert operation of each algorithm can be found in Appendix A, Table 1.

The marginally slower insertion performance in AOH at small scale dataset reflects an intentional design tradeoff, where structural maintenance incorporating multiple ordering constraints ensures optimal efficiency for subsequent key modification and delete operations. This additional structural information enables the system to achieve amortized time complexity surpassing for all four core operations, representing a significant advancement over conventional heap structures with less rigorous maintenance requirements. By storing more effective and stable relationships between nodes within the heap, AOH will exhibit increasingly better scalability as the data size continues to expand.

* 1. Decrease-Key

In Figure 9 (b), AOH heap is the most efficient for small datasets, while the Fibonacci heap outperforms the others at larger scales. AOH, pairing, and d-ary heap implementations show significantly poorer performance when processing large datasets. Although Fibonacci heaps demonstrate the best performance on most datasets, their actual scalability is the worst when scaling from 107 to 5×107. The actual clock cycle consumption for the decrease-key operation of each algorithm can be found in Appendix A, Table 2.

The Fibonacci heap excels in decrease-key operations due to several optimizations, including lazy consolidation, marking, and cascading cuts. Its flexible structure allows nodes to be quickly cut and relocated to the root list during decrease-key, avoiding immediate rebalancing. The extreme optimization of decrease-key operations is achieved at the expense of degraded performance in other operations. Moreover, its scalability is the poorest on large-scale datasets, which aligns with the performance trend observed during the insert operation.

Similar to insert operations, the decrease-key operation in AOH incurs performance overhead due to structural maintenance requirements. However, AOH can achieve significant efficiency improvements by utilizing larger *L* values on higher frequency CPUs, minimizing the number of Standard-Subtrees during consolidations, or through other methods, thereby substantially reducing the costs of consolidation.

* 1. Increases-Key

Benchmark results in Figure 9 (c) reveal complex performance trends for increase-key operations. AOH heap excels with small-sized datasets, while pairing and d-ary heaps dominate mid-sized datasets. At large sizes, the d-ary heap initially leads, but the pairing heap ultimately achieves the lowest runtime. Notably, AOH heap maintains intermediate efficiency, while the Fibonacci heap shows consistent but the worst performance across almost all scales. Pairing heaps exhibit optimal scalability and absolute performance at extreme scales (5×10⁷). The actual clock cycle consumption for the increase-key operation of each algorithm can be found in Appendix A, Table 3.

The pairing heap delivers superior performance on large-scale datasets. Although its theoretical analysis is less well-defined compared to Fibonacci heaps and d-ary heaps, it is a highly efficient alternative, particularly in scenarios where increase-key operations dominate. From AOH’s perspective, the four core heap operations are nearly identical, and it processes both increase-key and decrease-key uniformly. However, AOH is not optimal for the two operations due to the considerable overhead incurred by breaking the next-pre pointer pairs and reconstructing the bro-pre pointer pairs, leaving substantial room for further optimization. It is evident that the pairing heap exhibits optimal scalability for key modification operations on large-scale datasets.

* 1. Delete-Min

Benchmark results in Figure 9 (d) demonstrate that the AOH heap maintains consistent and substantial performance advantages in the delete-min operation. In contrast, d-ary, Fibonacci, and pairing heaps all exhibit significant performance degradation. Notably, while the performance differences for insert, increase-key, and decrease-key operations are small, delete-min emerges as the primary computational bottleneck across all heap data structure algorithms. Detailed clock cycle measurements for these delete-min operations are presented in Appendix A, Table 4.

AOH demonstrates a significant advantage in delete-min operations compared to Fibonacci, pairing, and d-ary heaps, highlighting fundamental architectural distinctions between these data structures. This advantage arises from its dual-order structure, which inherently facilitates more efficient deletions than traditional heap implementations.

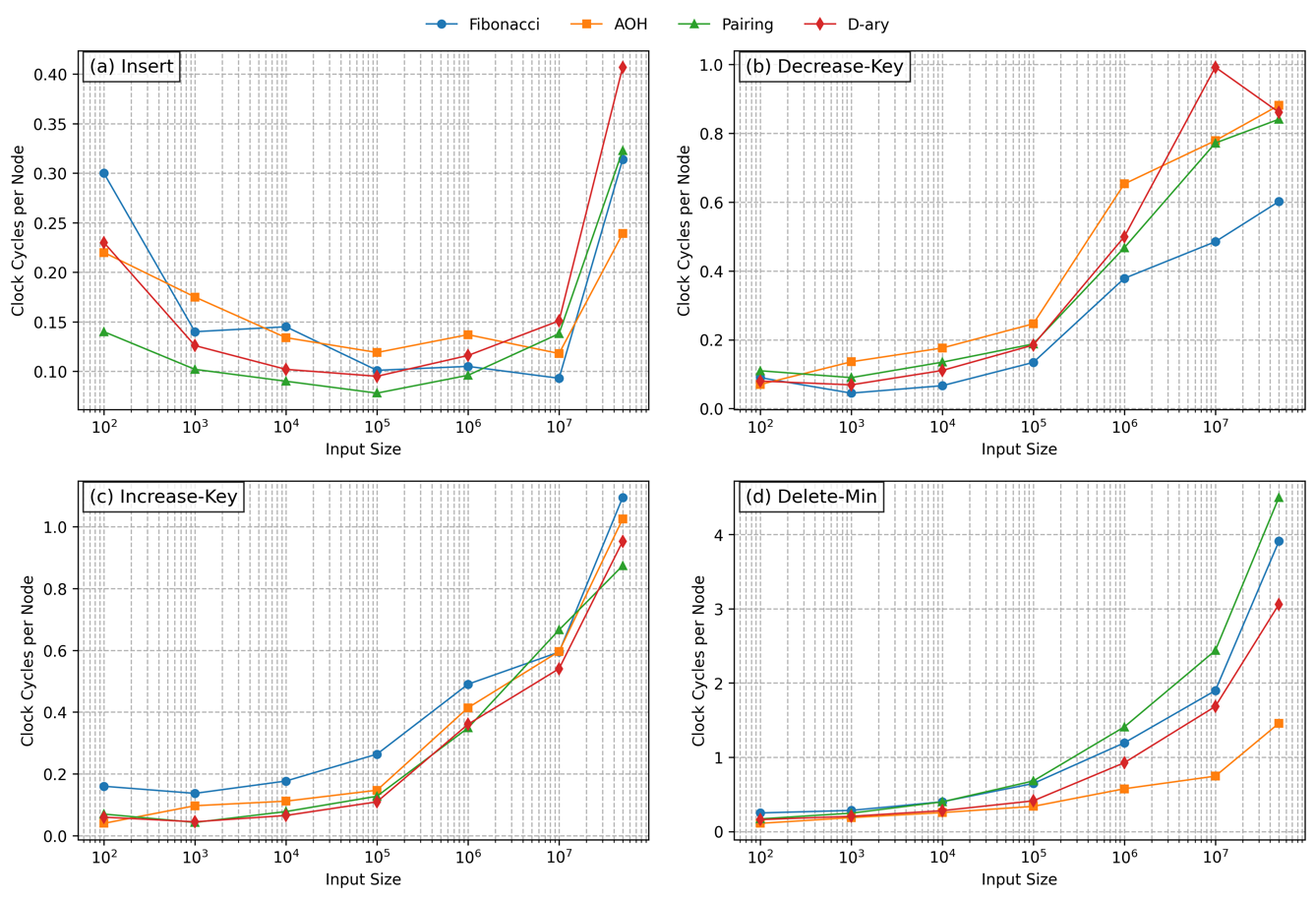


Figure 9: (a) Clock cycles used per node during insert operations; (b) Clock cycles used per node during decrease-key operations; (c) Clock cycles used per node during increase-key operations; (d) Clock cycles used per node during delete-min operations.

* 1. Effect of L on

An intuitive conclusion derived from the design of AOH is that the larger the parameter *L*, the smaller the average merged-height , indicating a significant negative correlation between the two. During consolidation of *L* trees, a larger *L* may results in a longer bro-pre bidirectional list. If this observed characteristic can be preserved, or even optimized and amplified through data structure design during algorithm execution, then increasing *L* naturally leads to a reduction in .

We have formally established this relationship through a theoretical time-complexity analysis based on . Subsequently, we will demonstrate the correlation between *L* and through experimental evaluation. The results conclusively verify this design intuition, showing that increased *L* values consistently yield lower . To investigate the effect of *L* on , a distinct set of datasets is randomly selected for each operation type. For AOH running on the dataset, we record both the merged-height and operation count, then compute using Equation 3.

In Figure 10, the number of operations for insert and delete-min is twice the input size, while the number of operations for increase-key and decrease-key is three times the input size. The value of *L* is 16, 64, or 256.

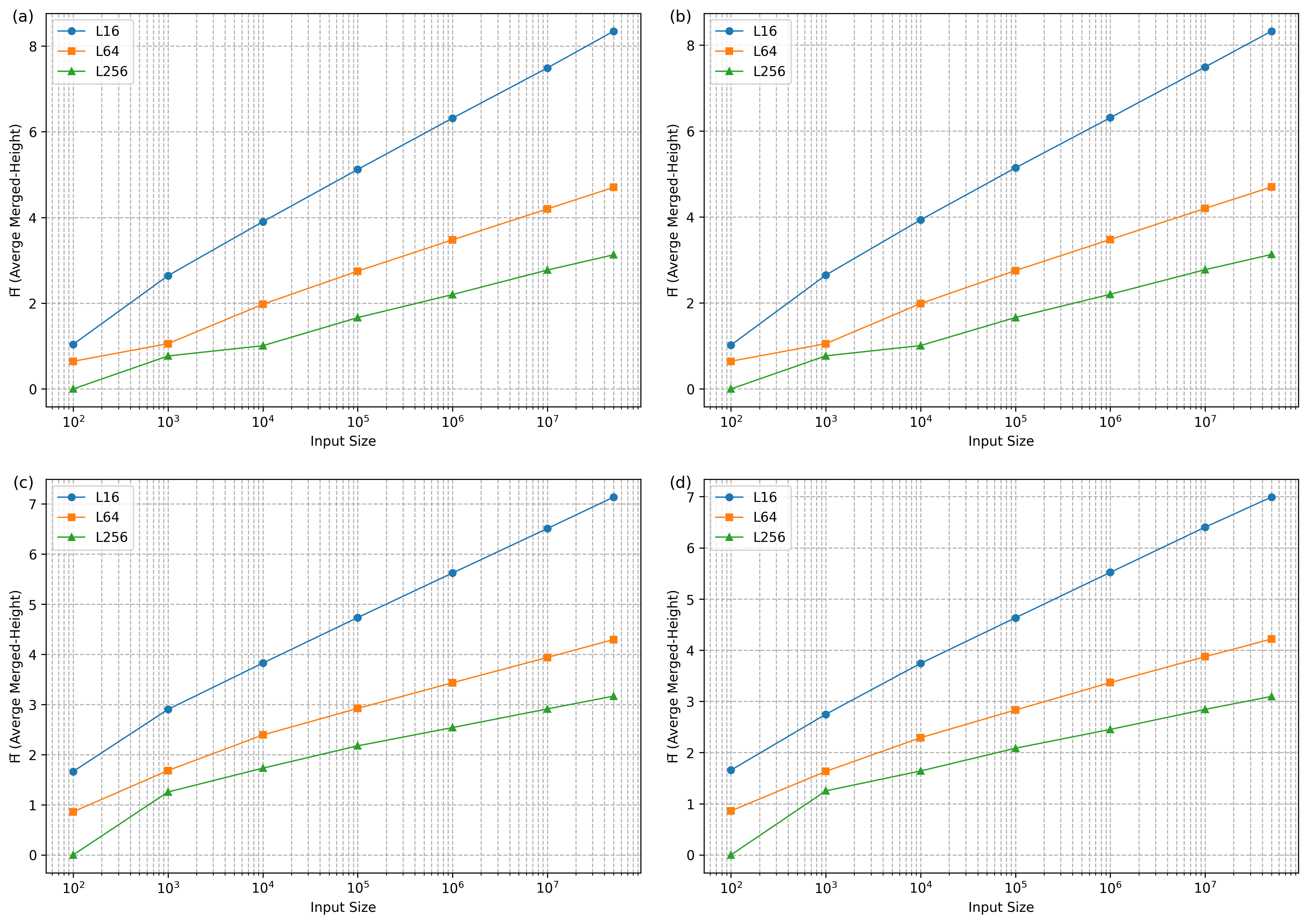


Figure 10: (a) Effect of L on during insert; (b) Effect of L on during delete-min; (c) Effect of L on during increase-key; (d) Effect of L on during decrease-key.

The experimental results demonstrate a consistent inverse relationship between *L* and the average merged-height across all four core operations. This finding substantiates AOH's remarkable scalability: enhancing computational resources—whether through increased CPU frequency or dedicated L-way extremum selector hardware modules—effectively reduces time complexity while substantially boosting performance. These results establish a robust positive correlation between computational capacity and algorithmic efficiency, where resource augmentation directly translates to reduced time complexity and improved operational performance. The experimental results confirm the logarithmic relationship between heap size and the average merged-height , thereby providing partial validation of the proposed time-complexity formulation (Eq. 6).

For N ≥ 1000, our experimental results indicate that is determined exclusively by *L* for each operation type and is statistically independent of the problem size *N*. The assumption concerning is empirically validated, as different *L* values for each operation type yield distinct reciprocals of log *C̅*, effectively explaining the scaling behavior observed in Equation (6). This work advances beyond the conventional bound by identifying the sibling node count as the fundamental time complexity determinant.

Structural innovation can indeed reduce the time complexity of the algorithm, and computing power investment will significantly improve the operating efficiency of the algorithm. This paper breaks through the upper limit with the help of structural innovation and hardware computing power. The computing power is abstracted as the number of sibling nodes, which becomes the essential problem of the heap algorithm and is reflected in the time complexity of AOH.

Complete experimental data and statistical analysis are provided in Appendix B.

1. Stability of Delete-Min Operations

To evaluate the stability of the four core operations of Pairing Heap, Fibonacci Heap, D-ary Heap, and AOH, this study generates test datasets with strictly descending keys (classified as "decreasing-type," see Appendix C) and strictly ascending keys (classified as "increasing-type," see Appendix D), using the same method as for generating random-type datasets. Stability is assessed by observing performance variations across the three different input patterns. Ideally, an algorithm should demonstrate consistent performance across various dataset types.

In Figure 11, AOH demonstrates relatively high stability across different input patterns, exhibiting smaller performance variations. In contrast, D-ary Heap, Fibonacci Heap and Pairing Heap show greater sensitivity to changes in input patterns. AOH heap is the ideal choice for handling unpredictable delete operations, while D-ary Heap, Fibonacci Heap, and Pairing Heap should be avoided in scenarios where delete operations occur frequently. Experimental data supporting these findings can be found in Appendix E.4.

All four core operations of AOH exhibit high stability. In terms of efficiency, its insert, decrease-key, and increase-key operations show no significant difference compared to the other three data structures. The crucial aspect is its substantial performance advantage and stability in delete-min operations, which effectively addresses the performance and stability issues associated with existing heap implementations. The experimental results in Appendices E.1-E.3 demonstrate consistent stability for the insert, decrease-key, and increase-key operations across all four data structures.

1. Discussion

The performance of AOH under high-frequency key-modification workloads can be significantly improved through lazy techniques, whether based on completely novel technical innovations or by leveraging inherent data characteristics. By strategically deferring both key updates and tree consolidation operations, AOH implementation would demonstrates superior performance compared to both Fibonacci and Pairing Heaps in dynamic key-update scenarios. For example, AOH can be optimized through controlled subtree consolidation by processing the fewest Standard-Subtrees per consolidation cycle according to a specific strategy, while maintaining the remaining unmerged trees in a binary heap or an array.

The computational complexities of the four core operations exhibit a super-linear relationship with heap size. The discrepancy stems may from our comprehensive measurement methodology, which incorporates the critical overhead of element localization within the heap structure. Therefore, for the purposes of algorithm evaluation and practical application, the testing methodology employed in this paper is more reliable. Specifically, our benchmarks account for the essential pointer resolution step that is inherently required in practical heap implementations but often abstracted away in theoretical analyses. This more realistic performance metric provides meaningful insights for real-world applications, where pointer resolution constitutes an unavoidable and often significant component of heap operations. The overall computational complexity converges to a unified pattern that may better reflect the actual system behavior. Regardless of whether the conflict arises from algorithmic factors, cache complexities, or the interplay between hash cache complexity and the inherent cache complexity of heap methods, our evaluation methodology offers greater theoretical and practical significance. Exploring the underlying causes and proposing optimization strategies will be a key focus of future research.

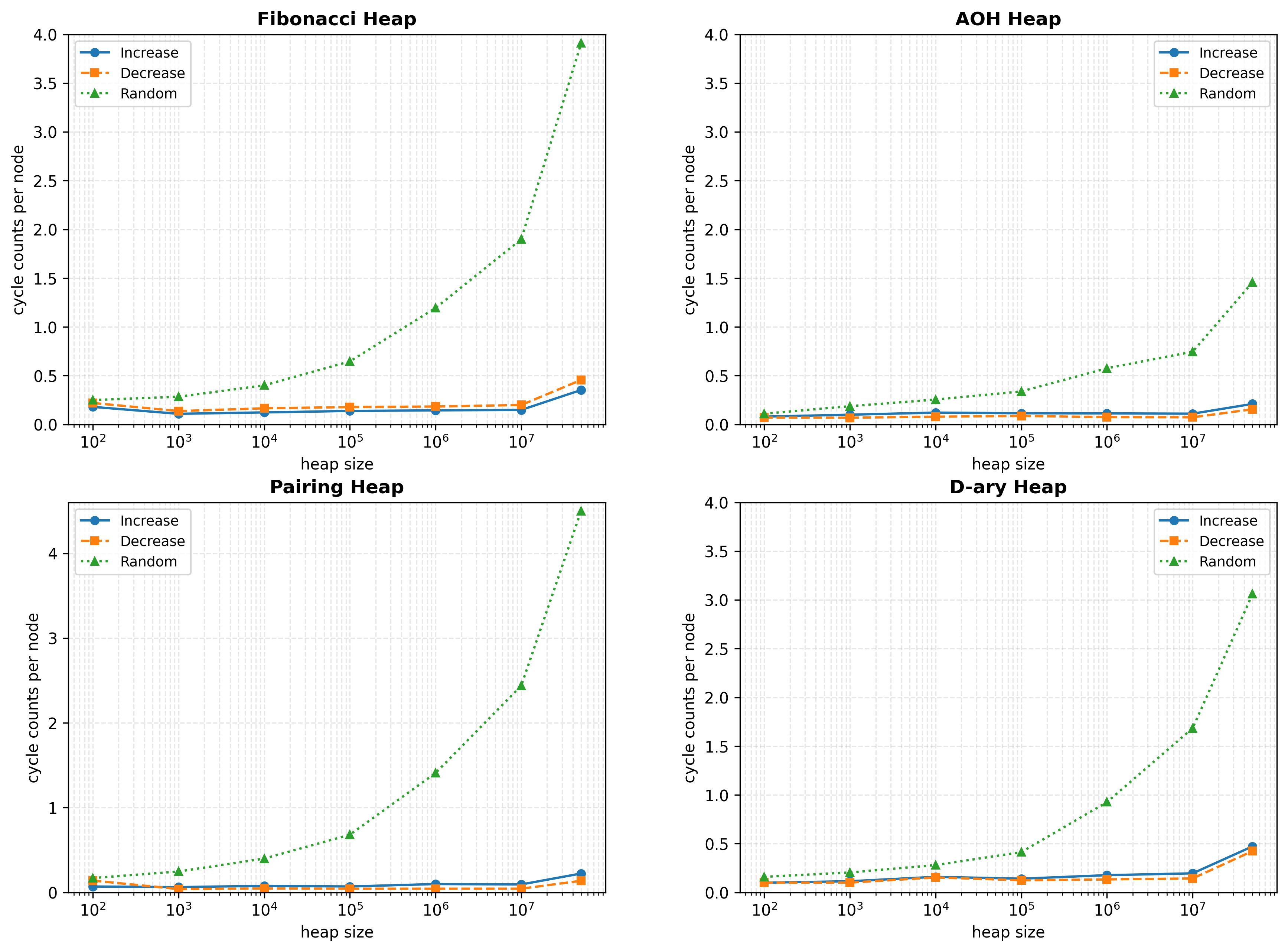


Figure 11: Delete-Min Operation Stability Across Heap Implementations

The unified pattern also poses a significant challenge to the previously claimed sub-logarithmic time complexities for the corresponding operations in pairing heaps and Fibonacci heaps. The time complexities of the four core operations in a d-ary heap depend on the input size *N*, typically when ignoring the effect of d. Furthermore, the above proof demonstrates that the amortized time complexities of the four core operations in AOH are all . The time complexity is not sub-logarithmic because is not dependent on the input size *N*, making the overall complexity proportional to as *N* scales. The above experimental evidence calls into question the widely accepted belief that some of the four core operations in pairing heaps and Fibonacci heaps have time complexities that are sub-logarithmic rather than . Could the original complexity assessments be inaccurate? This question warrants further in-depth investigation. Based on the analyses and experimental evidence presented in this paper, we argue that the time complexities of the four core operations in pairing heaps and Fibonacci heaps may in fact remain logarithmic rather than sub-logarithmic.

Additionally, investigating whether using a heap that supports key modifications instead of the binary heap employed can achieve better performance is a worthwhile research question. Furthermore, exploring the economic value of developing specialized multi-way maximum/minimum readers to replace the use of the binary heap is another pressing issue that warrants attention.

1. Conclusion

The single partial order relation between parent and child nodes in existing heaps creates a negative trade-off between maintaining structural invariants and achieving optimal performance and time complexity. The resources spent on preserving the heap structure become nearly pure overhead, offering little to no benefit to the algorithm's time complexity and performance. Moreover, from the fundamental design principles to underlying data structures, traditional heaps suffer from significant scalability and stability issues. Continuous hardware advancements—whether through increasing the computational power of a single device or horizontally scaling with multiple devices—fail to deliver corresponding performance improvements for existing heap implementations. This poses a significant challenge in the face of rapidly evolving hardware capabilities. The bottleneck is especially evident in delete-min operations, where both practical runtime efficiency and theoretical time complexity struggle to achieve breakthroughs comparable to other heap operations.

To address the limitations of traditional heaps, this paper extends the partial order relation from parent-child pairs to among child nodes themselves, and designs a simple yet effective data structure that is highly friendly to both delete-min operations and scalability. The use of an array and a binary heap for storing tree root nodes, combined with the construction of long bro-pre chains, completely breaks the traditional trade-off between performance, time complexity, and structural maintenance. From now on, maintaining structural invariants is no longer merely about correctness; rather, it exhibits a positive correlation with time complexity and performance, forming the foundation for further optimization, whether through structural innovations, algorithmic mechanisms, or hardware advancements.

The amortized time complexity of the four core operations depends on the merged-height rather than the actual tree height or the length of the next pointer chain originating from the tree root. is independent of the distribution of next–pre pointer pairs, a property that is critical for enabling further optimization through consolidation. Notably, this independence removes the necessity for tree-balancing constraints, thereby enabling continuous scalability, where increased computational resources directly lower time complexity and improve efficiency. Such a structural innovation presents considerable potential for advancing the development of algorithms.

To validate the intrinsic amortized time complexity of AOH, we conducted an analysis of the quantities *L*,, and , and further confirmed through empirical data that the amortized time complexity of the four core operations follows the formula , where is proven to be independent of *N*. This is a significant conclusion: the computational power *L*, after being transformed through structural innovation into, serves as the intrinsic factor that directly governs the algorithm’s time complexity, making the key parameter in the complexity expression. Empirical results demonstrate a statistically significant positive correlation between the array size *L* and , providing the theoretical foundation for understanding computational efficiency scaling with processing power. Compared to other heap variants, AOH demonstrates exceptional performance and stability, particularly in the delete-min operation.

The amortized time complexity of the delete-min operation surpasses the classical upper bounds of traditional heap data structures. This work demonstrates that structural innovation—rather than merely relying on increased computational power—can effectively overcome existing theoretical limitations, underscoring the value of deep insight and critical analysis, and achieving both theoretical advancements and practical engineering benefits.

AOH encounters two fundamental performance bottlenecks: (1) root node array traversal and sorting operations, and (2) inefficient batch consolidation of subtrees. While the first bottleneck can be substantially alleviated through hardware acceleration (e.g., higher-frequency CPUs), the more critical limitation stems from redundant consolidation operations during tree consolidation. To address this core challenge, we propose two complementary approaches: (1) introducing novel structural attributes and relational constraints to minimize unnecessary consolidation, and (2) exploiting domain-specific access patterns to optimize subtree management. These optimizations may effectively reduce the number of required consolidation operations while preserving the heap's theoretical guarantees.

The implementation code can be found at <https://github.com/idler66/AOHeap>.

ACKNOWLEDGMENTS

I would like to thank my wife, Ms. Geng, for taking care of our family during the years I was unemployed, and for her selfless understanding and unwavering support. I am also deeply grateful to Professor Chen for his expert advice and dedicated supervision, which greatly contributed to the completion of this study.

REFERENCES

1. Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. 2025. Strict Fibonacci Heaps. ACM Trans. Alogor. 21, 2, Article 15 (January 2025), 18 pages. <https://doi.org/10.1145/3707692>
2. Michael L. Fredmanand Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM 34, 3 (1987), 596–615. DOI: https://doi.org/10.1145/28869.28874
3. Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. The pairing heap: A new form of self-adjusting heap. Algorithmica 1, 1 (1986), 111–129. DOI: https://doi.org/10.1007/BF01840439
4. John William Joseph Williams. 1964. Algorithm 232: Heapsort. Communications of the ACM 7, 6 (1964), 347–348. DOI: https://doi.org/10.1145/512274.512284
5. Gerth Stølting Brodal. 2013. A survey on priority queues. In Proceedings of the International Conference on Space- Efficient Data Structures, Streams, and Alogorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday. Lecture Notes in Computer Science, Vol. 8066, Springer, 150–163. DOI: <https://doi.org/10.1007/978-3-642-40273-9_11>
6. Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. Journal of Functional Programming, 6(6):839–857, 1996.
7. Seth Pettie. 2015. Towards a Final Analysis of Pairing Heaps. In Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’15), January 4–6, 2015, San Diego, CA, USA. SIAM, 254–269. DOI:https://doi.org/10.1137/1.9781611973739.19
8. Tarjan, R. E. 1985. Amortized computational complexity. SIAM J. Algebraic Discrete Methods 6, 2 (Apr. 1985), 306–318. DOI:https://doi.org/10.1137/0606021
9. Dani Dorfman, Haim Kaplan, László Kozma, and Uri Zwick. 2018. Pairing heaps: The Forward variant. In Proceedings of the 43rd International Symposium on Mathematical Foundations of Computer Science. Leibniz International Proceedings in Informatics (LIPIcs), Vol. 117, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 13:1–13:14. DOI: https://doi.org/ 10.4230/LIPIcs.MFCS.2018.13
10. Daniel Dominic Sleator and Robert Endre Tarjan. 1986. Self-adjusting heaps. SIAM Journal of Computing 15, 1 (1986), 52–69. DOI: <https://doi.org/10.1137/0215004>
11. Maria Hartmann, László Kozma, Corwin Sinnamon, and RobertE. Tarjan. 2021. Analysis of smooth heaps and slim heaps. In Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP ’21). Leibniz International Proceedings in Informatics (LIPIcs), Vol. 198, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 79:1–79:20. DOI: <https://doi.org/10.4230/LIPIcs.ICALP.2021.79>
12. László Kozma and Thatchaphol Saranurak. 2020. Smooth heaps and a dual view of self-adjusting data structures. SIAM Journal of Computing 49, 5 (2020), STOC18–45–STOC 18–93. DOI: <https://doi.org/10.1137/18M1195188>
13. Haim Kaplanand Robert Endre Tarjan. 2008. Thin heaps, thick heaps. ACM Transactions on Alogorithms4, 1 (2008), 3:1–3:14. DOI: <https://doi.org/10.1145/1328911.1328914>
14. Clark Allan Crane. 1972. Linear Lists and Priority Queues as Balanced Binary Trees. Ph.D. Dissertation. Stanford University, Stanford, CA.
15. Amr Elmasry, Claus Jensen, and Jyrki Katajainen. 2008. Two-tier relaxed heaps. Acta Informatica 45, 3 (2008), 193–210. DOI: <https://doi.org/10.1007/s00236-008-0070-7>
16. Jean Vuillemin. 1978. A data structure for manipulating priority queues. Communications of the ACM 21, 4 (1978), 309–315. DOI: <https://doi.org/10.1145/359460.359478>
17. Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. 2011. Rank-pairing heaps. SIAM Journal of Computing 40, 6 (2011), 1463–1485. DOI: <https://doi.org/10.1137/100785351>
18. Corwin Sinnamon and RobertE. Tarjan. 2025. Efficiency of self-adjusting heaps. ACM Transactions on Algorithms (2025).
19. Dani Dorfman, Haim Kaplan, László Kozma, Seth Pettie, and Uri Zwick. 2018. Improved bounds for multipass pairing heaps and path-balanced binary search trees. In Proceedings of the 26th Annual European Symposium on Algorithms (ESA ’18), LIPIcs, Vol. 112, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:13. DOI: https: //doi.org/10.4230/LIPIcs.ESA.2018.24
20. MichaelL. Fredman. 1999. On the efficiency of pairing heaps and related data structures. J. ACM 46, 4 (1999), 473–501. DOI: <https://doi.org/10.1145/320211.320214>
21. Amr Elmasry. 2009. Pairing heaps with 𝑜(loglog𝑛) decrease cost. In Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 471–476. Retrieved from http://doi.acm.org/10.1145/1496770.1496822
22. Amr Elmasry. 2017. Toward optimal Self-adjusting heaps. ACM Transactions on Algorithms 13, 4 (2017), Article 55, 14 pages. DOI: <https://doi.org/10.1145/3147138>
23. John T . Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis . Communications of the ACM, 30(3) :234-249, March 1987.

A  RAndom key

Table 1: Clock cycles consumed by insert operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Fibonacci | 30 | 140 | 1453 | 10055 | 104908 | 929109 | 15699570 |
| AOH | 22 | 175 | 1336 | 11938 | 137204 | 1175545 | 11973090 |
| Pairing | 14 | 102 | 902 | 7846 | 96393 | 1376387 | 16174580 |
| D-ary | 23 | 126 | 1024 | 9465 | 116459 | 1514442 | 20345040 |

Table 2: Clock cycles consumed by decrease-key operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Fibonacci | 9 | 45 | 666 | 13415 | 378518 | 4849939 | 30097491 |
| AOH | 7 | 136 | 1762 | 24640 | 652925 | 7786720 | 44071274 |
| Pairing | 11 | 90 | 1349 | 18783 | 467526 | 7716629 | 42046954 |
| D-ary | 8 | 69 | 1108 | 18465 | 499292 | 9920904 | 43050789 |

Table 3: Clock cycles consumed by increase-key operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Fibonacci | 16 | 137 | 1770 | 26397 | 490371 | 5943868 | 54746129 |
| AOH | 4 | 97 | 1118 | 14722 | 414216 | 5962854 | 51275255 |
| Pairing | 7 | 43 | 781 | 12781 | 348830 | 6661346 | 43699159 |
| D-ary | 6 | 45 | 655 | 10973 | 360679 | 5404676 | 47640716 |

Table 4: Clock cycles consumed by delete-min operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Fibonacci | 25 | 284 | 4008 | 64636 | 1194201 | 18984676 | 195494281 |
| AOH | 11 | 187 | 2550 | 33863 | 575802 | 7459118 | 72892979 |
| Pairing | 17 | 247 | 3993 | 68077 | 1408615 | 24374698 | 224989810 |
| D-ary | 16 | 205 | 2806 | 41307 | 926364 | 16844187 | 153014655 |

B  Average merged-height

In this appendix, Tables 5-7 present the experimental results for *L*=16, 64, and 256 respectively, where the independent variable *x* represents (taking values 3, 4, 5, 6, 7, and 7 \* log 5), and the dependent variables are *H̅* for four core operations: insert, key-decrease, key-increase, and delete-min. For each fixed L on each type of operation, we conducted a linear regression (*y* = *k* \* *x* + *b*, where *y* represents *H̅* ) using ordinary least squares (OLS).

The linear fitting results from the experimental data show that all models have high R² values (close to 1), and the p-values are all less than 0.0001. Therefore, these linear fits can be considered reliable. It can be concluded that the slope *k* is remains invariant with respect to x, being solely determined by the parameter *L* for each operation type. This robust linear relationship conclusively demonstrates that is independent of problem size *N*.

Table 5: of Four Core Operations at L = 16

|  | 3 | 4 | 5 | 6 | 7 | 7\*log5 |
| --- | --- | --- | --- | --- | --- | --- |
| Insert | 2.644000 | 3.902550 | 5.121235 | 6.318553 | 7.489616 | 8.347980 |
| Decrease-key | 2.747667 | 3.742633 | 4.633140 | 5.520898 | 6.403667 | 6.991390 |
| Increase-key | 2.901667 | 3.824633 | 4.731340 | 5.620312 | 6.506419 | 7.133426 |
| Delete-min | 2.648000 | 3.933050 | 5.141655 | 6.308402 | 7.488569 | 8.321159 |

Table 6: of Four Core Operations at L = 64

|  | 3 | 4 | 5 | 6 | 7 | 7\*log5 |
| --- | --- | --- | --- | --- | --- | --- |
| Insert | 1.057000 | 1.976300 | 2.748860 | 3.479367 | 4.200098 | 4.703200 |
| Decrease-key | 1.631667 | 2.290100 | 2.831783 | 3.366850 | 3.873060 | 4.216692 |
| Increase-key | 1.682000 | 2.391100 | 2.920663 | 3.431946 | 3.936116 | 4.293537 |
| Delete-min | 1.053000 | 1.987050 | 2.753330 | 3.475734 | 4.199172 | 4.696862 |

Table 7: of Four Core Operations at L = 256

|  | 3 | 4 | 5 | 6 | 7 | 7\*log5 |
| --- | --- | --- | --- | --- | --- | --- |
| Insert | 0.769500 | 1.009350 | 1.664095 | 2.202320 | 2.773728 | 3.129983 |
| Decrease-key | 1.251000 | 1.639800 | 2.084117 | 2.451888 | 2.843409 | 3.095897 |
| Increase-key | 1.254333 | 1.729200 | 2.173857 | 2.538564 | 2.909921 | 3.165956 |
| Delete-min | 0.769500 | 1.009150 | 1.662330 | 2.202167 | 2.773428 | 3.127870 |

C  decrease key

Following the same methodology, we construct test datasets with strictly monotonically increasing keys to evaluate the algorithm's robustness against this specific input pattern. The heap is initialized with elements containing strictly monotonically decreasing keys (from *N* down to 1), where *N* represents the dataset cardinality. All subsequent key modifications (including both decrease-key and increase-key operations) are generated at random within the range [1, *N*]. The below experimental results exhibit nearly consistent performance trends with those from randomly generated datasets, confirming the robustness of AOH across varying operational conditions.

Table 8: Clock cycles consumed by insert operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 34 | 89 | 1000 | 8706 | 92189 | 824560 | 8106871 |
| Fibonacci | 17 | 82 | 815 | 8268 | 88294 | 826621 | 9447808 |
| Pairing | 9 | 63 | 705 | 6151 | 80268 | 747598 | 7585986 |
| D-ary | 14 | 82 | 840 | 9800 | 106862 | 1033190 | 11762207 |

Table 9: Clock cycles consumed by decrease-key operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 4 | 55 | 786 | 12401 | 403273 | 4886410 | 30921693 |
| Fibonacci | 3 | 20 | 318 | 5669 | 268856 | 3615134 | 24195462 |
| Pairing | 5 | 37 | 536 | 8390 | 373287 | 4754652 | 32121210 |
| D-ary | 5 | 32 | 466 | 7660 | 433571 | 6477727 | 52783998 |

Table 10: Clock cycles consumed by increase-key operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 5 | 55 | 1053 | 12052 | 380431 | 4844813 | 48130705 |
| Fibonacci | 17 | 89 | 1339 | 16856 | 494201 | 9509021 | 62888665 |
| Pairing | 5 | 36 | 705 | 9252 | 468899 | 6344454 | 38951054 |
| D-ary | 3 | 31 | 476 | 6484 | 361695 | 5185923 | 39561341 |

Table 11: Clock cycles consumed by delete-min operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 7 | 69 | 794 | 8734 | 74799 | 731517 | 7777591 |
| Fibonacci | 22 | 137 | 1654 | 17817 | 183928 | 1990669 | 22809357 |
| Pairing | 14 | 40 | 469 | 4414 | 44148 | 443435 | 6878130 |
| D-ary | 10 | 101 | 1538 | 12508 | 133628 | 1433588 | 21230417 |

D  Increase key

Following the same methodology, we construct test datasets with strictly monotonically increasing keys to evaluate the algorithm's robustness against this specific input pattern. The heap is initialized with elements containing strictly monotonically increasing keys (from 1 down to *N*), where *N* represents the dataset cardinality. All subsequent key modifications (including both decrease-key and increase-key operations) are generated at random within the range [1, *N*]. These experimental results exhibit consistent performance trends with those from randomly generated datasets, confirming the robustness of AOH across varying operational conditions.

Table 12: Clock cycles consumed by insert operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 43 | 134 | 1209 | 8702 | 97144 | 936636 | 7565633 |
| Fibonacci | 19 | 100 | 872 | 7930 | 89998 | 872060 | 8251206 |
| Pairing | 12 | 73 | 702 | 6223 | 86090 | 828394 | 9654326 |
| D-ary | 25 | 92 | 808 | 7276 | 87297 | 927349 | 9756167 |

Table 13: Clock cycles consumed by decrease-key operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 5 | 56 | 794 | 12851 | 412610 | 7898339 | 44664393 |
| Fibonacci | 3 | 20 | 333 | 5789 | 269178 | 3737330 | 24258984 |
| Pairing | 4 | 42 | 524 | 10080 | 328359 | 5269256 | 33100466 |
| D-ary | 4 | 32 | 487 | 8533 | 421370 | 9260746 | 42771177 |

Table 14: Clock cycles consumed by increase-key operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 4 | 56 | 1031 | 11713 | 380283 | 4656597 | 42818795 |
| Fibonacci | 10 | 86 | 1164 | 17368 | 502879 | 5879412 | 54514251 |
| Pairing | 2 | 20 | 724 | 5480 | 420116 | 6061247 | 51554546 |
| D-ary | 3 | 29 | 682 | 8394 | 350930 | 4529885 | 42803933 |

Table 15: Clock cycles consumed by delete-min operations

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| AOH | 8 | 100 | 1212 | 11477 | 113228 | 1104188 | 10522959 |
| Fibonacci | 18 | 109 | 1233 | 13832 | 144424 | 1486327 | 17653399 |
| Pairing | 7 | 63 | 778 | 7048 | 99126 | 943913 | 11167265 |
| D-ary | 10 | 115 | 1602 | 14110 | 177270 | 1959439 | 23630560 |

E  Reorganized Experimental Results

We organize the experimental results for four heap implementations (AOH, Pairing, Fibonacci, and d-ary) according to three fundamental input patterns: (1) increasing-type, (2) decreasing-type, and (3) random-type data. This classification enables comparative evaluation of operational stability across varying data distributions when applying identical methods.

E.1 Insert Operations

Table 16: Fibonacci Heap Insert Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 19 | 100 | 872 | 7930 | 89998 | 872060 | 8251206 |
| decreasing-type | 17 | 82 | 815 | 8268 | 88294 | 826621 | 9447808 |
| random-type | 30 | 140 | 1453 | 10055 | 104908 | 929109 | 15699570 |

Table 17: AOH Heap Insert Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 43 | 134 | 1209 | 8702 | 97144 | 936636 | 7565633 |
| decreasing-type | 34 | 89 | 1000 | 8706 | 92189 | 824560 | 8106871 |
| random-type | 22 | 175 | 1336 | 11938 | 137204 | 1175545 | 11973090 |

Table 18: Pairing Heap Insert Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 12 | 73 | 702 | 6223 | 86090 | 828394 | 9654326 |
| decreasing-type | 9 | 63 | 705 | 6151 | 80268 | 747598 | 7585986 |
| random-type | 14 | 102 | 902 | 7846 | 96393 | 1376387 | 16174580 |

Table 19: D-ary Heap Insert Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 25 | 92 | 808 | 7276 | 87297 | 927349 | 9756167 |
| decreasing-type | 14 | 82 | 840 | 9800 | 106862 | 1033190 | 11762207 |
| random-type | 23 | 126 | 1024 | 9465 | 116459 | 1514442 | 20345040 |

E.2 Decrease-Key Operations

Table 20: Fibonacci Decrease-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 3 | 20 | 333 | 5789 | 269178 | 3737330 | 24258984 |
| decreasing-type | 3 | 20 | 318 | 5669 | 268856 | 3615134 | 24195462 |
| random-type | 9 | 45 | 666 | 13415 | 378518 | 4849939 | 30097491 |

Table 21: AOH Decrease-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 5 | 56 | 794 | 12851 | 412610 | 7898339 | 44664393 |
| decreasing-type | 4 | 55 | 786 | 12401 | 403273 | 4886410 | 30921693 |
| random-type | 7 | 136 | 1762 | 24640 | 652925 | 7786720 | 44071274 |

Table 22: Pairing Decrease-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 4 | 42 | 524 | 10080 | 328359 | 5269256 | 33100466 |
| decreasing-type | 5 | 37 | 536 | 8390 | 373287 | 4754652 | 32121210 |
| random-type | 11 | 90 | 1349 | 18783 | 467526 | 7716629 | 42046954 |

Table 23: D-ary Decrease-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 4 | 32 | 487 | 8533 | 421370 | 9260746 | 42771177 |
| decreasing-type | 5 | 32 | 466 | 7660 | 433571 | 6477727 | 52783998 |
| random-type | 8 | 69 | 1108 | 18465 | 499292 | 9920904 | 43050789 |

E.3  Increase-Key Operations

Table 24: Fibonacci Increase-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 10 | 86 | 1164 | 17368 | 502879 | 5879412 | 54514251 |
| decreasing-type | 17 | 89 | 1339 | 16856 | 494201 | 9509021 | 62888665 |
| random-type | 16 | 137 | 1770 | 26397 | 490371 | 5943868 | 54746129 |

Table 25: AOH Increase-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 4 | 56 | 1031 | 11713 | 380283 | 4656597 | 42818795 |
| decreasing-type | 5 | 55 | 1053 | 12052 | 380431 | 4844813 | 48130705 |
| random-type | 4 | 97 | 1118 | 14722 | 414216 | 5962854 | 51275255 |

Table 26: Pairing Increase-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 2 | 20 | 724 | 5480 | 420116 | 6061247 | 51554546 |
| decreasing-type | 5 | 36 | 705 | 9252 | 468899 | 6344454 | 38951054 |
| random-type | 7 | 43 | 781 | 12781 | 348830 | 6661346 | 43699159 |

Table 27: D-ary Increase-Key Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 3 | 29 | 682 | 8394 | 350930 | 4529885 | 42803933 |
| decreasing-type | 3 | 31 | 476 | 6484 | 361695 | 5185923 | 39561341 |
| random-type | 6 | 45 | 655 | 10973 | 360679 | 5404676 | 47640716 |

E.4 Delete-Min Operations

Table 28: Fibonacci Delete-Min Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 18 | 109 | 1233 | 13832 | 144424 | 1486327 | 17653399 |
| decreasing-type | 22 | 137 | 1654 | 17817 | 183928 | 1990669 | 22809357 |
| random-type | 25 | 284 | 4008 | 64636 | 1194201 | 18984676 | 195494281 |

Table 29: AOH Delete-Min Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 8 | 100 | 1212 | 11477 | 113228 | 1104188 | 10522959 |
| decreasing-type | 7 | 69 | 794 | 8734 | 74799 | 731517 | 7777591 |
| random-type | 11 | 187 | 2550 | 33863 | 575802 | 7459118 | 72892979 |

Table 30: Pairing Delete-Min Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 7 | 63 | 778 | 7048 | 99126 | 943913 | 11167265 |
| decreasing-type | 14 | 40 | 469 | 4414 | 44148 | 443435 | 6878130 |
| random-type | 17 | 247 | 3993 | 68077 | 1408615 | 24374698 | 224989810 |

Table 31: D-ary Delete-Min Performance (Clock Cycles) by Input Pattern

|  | 10² | 103 | 104 | 105 | 106 | 107 | 5\*107 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| increasing-type | 10 | 115 | 1602 | 14110 | 177270 | 1959439 | 23630560 |
| decreasing-type | 10 | 101 | 1538 | 12508 | 133628 | 1433588 | 21230417 |
| random-type | 16 | 205 | 2806 | 41307 | 926364 | 16844187 | 153014655 |

1. [↑](#footnote-ref-1)