MASTER'S THESIS 2025

A Performance Study of Priority Queues: Binary Heap, Fibonacci Heap, Hollow Heap

Klara Tjernström, Vera Paulsson

ISSN 1650-2884
LU-CS-EX: 2025-18

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2025-18

A Performance Study of Priority Queues: Binary Heap, Fibonacci Heap, Hollow Heap

En prestandaundersökning av prioritetsköer: Binär Heap, Fibonacci Heap, Hollow Heap

Klara Tjernström, Vera Paulsson

A Performance Study of Priority Queues: Binary Heap, Fibonacci Heap, Hollow Heap

Klara Tjernström
klara.tjernstrom@gmail.com

Vera Paulsson
vera.paulsson01@gmail.com

June 10, 2025

Master's thesis work carried out at Tactel, part of Panasonic Avionics.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Joel Engström, joel.engstrom@tactel.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

This master thesis evaluates the performance of Binary, Fibonacci, and Hollow Heaps used as priority queues in Dijkstra's algorithm, with the goal of identifying the most suitable data structure for implementation in Panasonic Avionics' Arc application. The heaps were tested across varying graph types to determine their impact on execution time, memory access patterns, and cache efficiency. Despite its higher theoretical time complexity, the Binary Heap achieved the shortest execution time for *insert* in operation benchmarking, owing to its contiguous memory layout and predictable access patterns. The Fibonacci Heap showed the fastest performance for the *extract_min* and *decrease_key* operations. Both Fibonacci and Hollow Heaps incurred overhead from pointer usage and irregular memory access.

When used as a priority queue in Dijkstra's algorithm on sparse graphs, the Binary Heap completed execution in 0.32-0.59× the runtime of the Fibonacci Heap, and 0.20-0.49× that of the Hollow Heap, depending on the graph size. On dense graphs, this performance advantage diminished, and the Fibonacci Heap achieved up to 1.24× faster execution time than the Binary Heap. This improvement is attributed to the more efficient *decrease_key* operation in the Fibonacci Heap, which, in the dense graph setting, outweighs its relatively slower *insert* and *extract_min* operations.

Implementations in C, Java, and Rust were compared for the Fibonacci Heap, with C showing the overall fastest execution times. Java and Rust offered safer abstractions but introduced runtime and complexity trade-offs. Cache behavior was analyzed using `cachegrind`, and exhibited that Binary Heap had the fewest cache misses. Prefetching was explored in all implementations; while it showed some benefit for minimizing L1-misses, overhead added complexity limited its overall execution time. The results demonstrate that the Binary Heap achieves the best execution time and cache performance across the evaluated graphs.

Keywords: Binary Heap, Fibonacci Heap, Hollow Heap, Heap Comparison, Djikstra's Algorithm, Data Prefetching, Cache Memory, C, Java, Rust

Acknowledgements

Throughout the work on this thesis, we have had the privilege of receiving support and encouragement from many people. First and foremost, we would like to thank Jonas Skeppstedt for his invaluable guidance throughout the technical aspects of our work. No matter the challenge, he was always there to offer insight and direction. On top of that, his encouraging words reminded us to believe in ourselves and capabilities, which meant more than he probably realizes.

Thank you to Flavius Gruian for his valuable feedback, which contributed to improving the quality of this report.

We would also like to thank Joel Engström for being our supervisor and primary contact at Tactel. Thank you for the many enjoyable moments of laughter.

Klara: My sincerest thanks to Vera for being a great thesis partner and friend. I would also like to thank my parents and sister for their endless love and support. I dedicate this work to my grandfather, who passed away during this school year. He was an engineer and has been a source of inspiration for me to become one as well.

Vera: I would like to express my deepest gratitude to Klara for being an unwaveringly supportive thesis partner and friend. I am also deeply thankful to my family for their constant support and belief in me. This journey has certainly been challenging, and I am incredibly grateful to all my friends for making it bearable and transforming it into an unforgettable experience that I will forever cherish.

Lastly, a big thanks to Tactel for hosting us this spring and providing the opportunity to contribute, with the hope that our work will yield valuable results for future use.

Contents

1 Introduction **7**

1.1 Contributions . . . 8
1.2 Research Questions . . . 9
1.3 Division of Work . . . 9

2 Background **11**

2.1 Heap . . . 11
2.1.1 Binary Heap . . . 12
2.1.2 Fibonacci Heap . . . 12
2.1.3 Hollow Heap . . . 13
2.2 Time Complexity . . . 14
2.3 Dijkstra's Algorithm . . . 15
2.4 Graph Characteristics . . . 17
2.5 Programming languages . . . 17
2.5.1 C . . . 18
2.5.2 Rust . . . 18
2.5.3 Java . . . 19
2.6 Cache Memory . . . 19
2.7 Prefetching . . . 20
2.8 Panasonic Avionics Arc . . . 21
2.9 Related Work . . . 22

3 Methodology **25**

4 Heap Implementation Details **27**

4.1 Binary Heap . . . 27
4.2 Fibonacci Heap . . . 28
4.3 Hollow Heap . . . 29
4.4 Prefetching in Heaps Data Structure . . . 29
4.5 Prefetching in Dijkstra's Algorithm . . . 33

5 Evaluation **35**
5.1 Test Environment . . . 35
5.2 Execution Time Test . . . 36
5.3 C, Java and Rust Test . . . 37
5.4 Dijkstra's Algorithm Test . . . 38
5.5 Memory Usage . . . 39
5.6 Cache Behavior and Prefetching . . . 39
5.7 Prefetching in Dijkstra's Algorithm . . . 40

6 Results **41**
6.1 Execution Time . . . 41
6.2 Language Evaluation . . . 45
6.3 Dijkstra's Algorithm . . . 46
6.3.1 Dense graph . . . 46
6.3.2 Sparse graph . . . 47
6.4 Memory Usage . . . 49
6.5 Cache Behavior and Prefetching . . . 50
6.6 Prefetching in Dijkstra's Algorithm . . . 52
6.7 `operf` . . . 53

7 Discussion **59**
7.1 Execution Time . . . 60
7.2 Language Evaluation . . . 61
7.3 Dijkstra's Algorithm . . . 62
7.4 Memory Usage . . . 63
7.5 Cache Behavior and Prefetching . . . 64
7.6 Prefetching in Dijkstra's Algorithm . . . 65
7.7 Sources of Error . . . 66
7.8 Future Work . . . 66

8 Conclusion **69**

Chapter 1
Introduction

Priority queues are fundamental data structures with widespread applications, ranging from process scheduling in operating systems to shortest path algorithms in graph-based applications. The choice of priority queue implementation significantly impacts the efficiency of these applications, particularly when operations such as insertions, deletions, and priority updates occur frequently. Theoretical analysis using Big O notation provides insight into the worst-case complexity of various implementations, but practical performance is often influenced by additional factors such as memory access patterns, cache efficiency, and the underlying programming language [4].

Among the many priority queue implementations, Binary Heaps are widely used due to their simplicity and balanced trade-offs between insertion and deletion efficiency [13]. However, they become less effective in scenarios with frequent *decrease_key* operations, as encountered in Dijkstra's algorithm [4]. To improve performance in such cases, more advanced heap structures have been developed [30, pp. 309–315]. Fibonacci Heaps, in particular, build on earlier designs such as binomial queues and support amortized constant time for most standard operations and *O(logn)* amortized time for deletions [6, 7]. The amortized time refers to the average performance of a sequence of operations [29, p. 451]. More recently, Hollow Heaps have emerged as an improvement over Fibonacci Heaps, reducing implementation overhead while maintaining favorable amortized performance [9].

This thesis aims to evaluate the performance of these different heap structures, Binary Heap, Fibonacci Heap, and Hollow Heap, when used in Dijkstra's shortest path algorithm. The study investigates how the choice of heap affects execution time across various types of graphs and explores whether one implementation consistently outperforms the others in practical scenarios. Furthermore, the choice of implementation language and how it will affect execution time are examined.

Dijkstra's algorithm was chosen as the context for this study because it relies on the heap operations *insert*, *extract_min*, and *decrease_key*. These operations are performed with varying frequency depending on the graph structure and expose the practical differences between heap implementations. In sparse graphs, *decrease_key* is less dominant, while in denser graphs,

it becomes the most frequent operation. This makes Dijkstra's algorithm suitable for evaluating how different heaps perform under conditions where both operation mix and memory behavior affect runtime.

Another consideration in this study is memory access patterns. Pointer-based data structures like Fibonacci and Hollow Heaps exhibit irregular memory access patterns, leading to higher cache miss rates compared to array-based structures like Binary Heaps. The inefficiencies introduced by poor spatial locality can hinder performance, particularly in graph-based applications with frequent key updates. Prefetching techniques, which aim to reduce memory latency by predicting and loading required data in advance, may offer performance improvements. However, whether these optimizations are feasible in a compiler or require explicit implementation remains an open question.

Through this research, we seek to understand the practical trade-offs between different priority queue implementations and identify conditions under which specific heaps are most suitable for Panasonic Avionics Arc. Our findings will contribute to optimizing heap selection in performance-critical applications, particularly those involving graph algorithms with dynamic priority updates.

1.1 Contributions

This study is situated in the field of priority queues and heap data structures, with a particular focus on their application within Panasonic Avionics Arc [28] which is an in-flight map. The objective is to evaluate the practical performance of three priority queue implementations, Binary Heap, Fibonacci Heap, and Hollow Heap. Through this evaluation the aim is to determine which heap structure is best suited for integration into Panasonic Arc's rendering engine, where dynamic prioritization of map elements is essential. The specific use case under consideration involves the selection and prioritization of datapoints to be rendered on the Arc map display. Since it is not feasible to test the heap implementations directly within the full Arc application, Dijkstra's algorithm is used as a representative environment in which to evaluate and compare the heap implementations. The algorithm's reliance on frequent *insert*, *extract_min*, and *decrease_key* operations makes it suitable for comparing the performance of a priority queue.

While each heap offers theoretical advantages, performance in a practical setting can deviate due to hidden constants. By applying these structures within a practical use case, this research aims to provide insights into how theoretical efficiency translates into actual performance.

Initially, all three heap structures are implemented and evaluated in C to provide a baseline for comparison. The most performant implementation in C will then be reimplemented and benchmarked in Java and Rust to assess how the programming language influences execution time and system-level behavior. In addition, this study explores whether further performance improvements can be achieved through prefetching. Specifically, we investigate whether it is feasible to integrate prefetching directly into the data structure to reduce cache latency, and whether it is possible to predict which data will be accessed during execution.

By evaluating both executime time and memory behavior, the findings of this thesis aim to support the selection of an optimal heap implementation for Arc, which could potentially offer useful insights for other performance-sensitive, graph-based applications.

1.2 Research Questions

- RQ1: Which heap produces the fastest execution time in Dijkstra's algorithm and why? Which is most appropriate for Arc to use?
- RQ2: To what extent does the choice of programming language affect heap performance? Can performance be generalized across languages, or are they implementation-specific?
- RQ3: How does the number of memory accesses differ between the different data structures and how efficient are cache memories for the Arc project?
- RQ4: Can prefetching improve performance either at the heap level, within Dijkstra's algorithm, or through compiler-level optimizations?

1.3 Division of Work

Throughout the course of this master's thesis, the majority of the work was carried out collaboratively by both authors. All major code parts were developed through pair programming to ensure shared understanding and equal knowledge of the implementations. The same approach was applied to the report writing where both authors contributed to all sections, continuously reviewing and building on each other's work to maintain a coherent style.

Regarding the division of responsibilities, certain focus areas were assigned to each author. Klara was responsible for implementing heaps in C, Dijkstra's algorithm, and integrating prefetching, along with conducting corresponding evaluations and tests for cache behavior and memory usage. Vera was responsible for implementing the Java and Rust versions of the heaps, as well as setting up the performance tests related to execution time and language evaluation.

While certain responsibilities were divided, all major decisions and analyses were discussed and agreed upon jointly to ensure alignment throughout the project.

Chapter 2
Background

This chapter provides the theoretical foundation necessary to understand the heap data structures implemented and evaluated in this thesis. It begins with an overview of heap structures and their key operations, along with how performance of their operations can be measured through time complexity. Following this, Dijkstra's algorithm is introduced as it serves as a practical use case and benchmark for evaluating the performance of different heap implementations. Finally, the chapter introduces Panasonic Avionics Arc [28], which constitutes the target environment for the performance evaluation.

2.1 Heap

Heaps are tree-based data structures in which each node contains a key that determines its position in the tree in order to satisfy the heap property. There are two kinds of heaps: min-heap and max-heap. In a min-heap each parent node has a key less than or equal to its children, whereas in a max-heap, each parent node has a key greater than or equal to its children [4, p. 152]. As a result, the root of the heap always contains the minimum or maximum element, depending on the type of heap. This thesis focuses exclusively on min-heaps, as they will be used in Dijkstra's shortest path algorithm where access to the smallest element is prioritized.

A heap supports several fundamental operations such as inserting new elements, accessing the minimum element, and removing the minimum element. Some heap variants also support additional operations such as decreasing the key of an element or merging two heaps efficiently.

Different heap implementations offer trade-offs in terms of time complexity, memory usage, and implementation complexity. This study focuses on comparing the trade-offs between the Binary Heap, the Fibonacci Heap, and the Hollow Heap. The Hollow Heap was selected because it is a recent data structure with limited practical evaluation. It was introduced as a simplification and improvement over the Fibonacci Heap, which motivates a direct comparison between the two. The Fibonacci Heap, in turn, was proposed as a general-

ization of binomial heaps with the main intent to improve Dijkstra's algorithm. The Binary Heap is included as a baseline due to its simple structure and widespread use. Each of these structures is described in more detail below.

2.1.1 Binary Heap

The Binary Heap is a data structure introduced by Williams in 1964, structured as a complete binary tree with all levels fully filled except possibly the last, filled from left to right without gaps [31].

The heap is implemented using an array where one node in the tree corresponds to an element in the array [4, p. 151]. The root is located at index 1, with the left and right children of a node at index k found at $2k$ and $2k + 1$ respectively, while its parent resides at $\lfloor k/2 \rfloor$ [31].

The fundamental operations of a Binary Heap include *insertion*, *deletion*, and *findMin* (finding the minimum or maximum key). Insertion adds a new element at the end of the heap, followed by an upward percolation until the heap property is restored. This process is commonly referred to as *up_heapify*, where the new element is compared to its parent and swapped if necessary until the correct position is found. Similarly, deletion also called the *extract_min* operation, removes the element in the heap with the smallest priority by replacing the root with the last element and restores order via downward percolation, known as *down_heapify*.

Binary heaps are widely used due to their logarithmic insertion and deletion times, as well as their compact array representation, which reduces memory usage [31]. However, operations such as *decrease_key* and heap merging are more costly in Binary Heaps. When a decrease key operation is performed on a Binary Heap, an arbitrary node in the heap lowers its priority. If the change in priority violates heap property the node is moved upwards in the tree. In worst case scenario, a node may be moved all the way up to the root creating logarithmic worst case time complexity. If no property is violated, no nodes needs to be moved and the structure is kept.

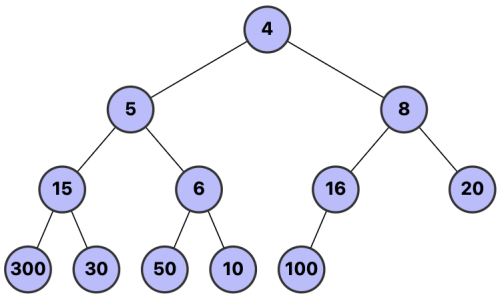


Figure 2.1: Example of a Binary Heap maintaining the min-heap order property. Visualized with [1].

2.1.2 Fibonacci Heap

The Fibonacci Heap is a heap data structure introduced by Fredman and Tarjan in 1984 [6, 7]. It consists of a collection of heap-ordered trees, where each tree satisfies the min-heap

property. The structure supports efficient operations through the use of lazy evaluation and amortized analysis.

The key operations that contribute to its efficiency are *insertion*, *decrease_key*, and *extract_min*. Insertions are done by simply adding the new node to the root list without restructuring [6, 7]. This allows insertion and merging of heaps to be performed in constant time.

The *decrease_key* operation is more efficient in the Fibonacci heap than in the Binary Heap. If the new key violates the heap order, the node is cut from its parent and moved to the root list. If its parent had previously lost a child, it is also cut and moved to the root list [6, 7]. This process continues recursively and is referred to as cascading cuts. The operation has an amortized cost of *O(1)*, which makes the Fibonacci Heap suitable for applications that require frequent updates to keys.

Deleting the minimum element involves removing the root node containing the minimum key and merging its children into the root list. After this, a consolidation step is performed to reduce the number of trees in the root list. Trees of the same degree are repeatedly linked until no two trees have the same degree. This operation has an amortized complexity of *O(logn)*.

Fibonacci Heaps are often used in graph algorithms like Dijkstra's and Prim's, where the efficiency of the *decrease_key* operation is important. However, the structure is rarely used in practice due to implementation complexity and memory overhead which makes it interesting case to compare and see if the increased effiency outweighs the memory overhead. Each node typically maintains five pointers and two additional fields, which can negatively affect cache performance [4, p. 507].

2.1.3 Hollow Heap

Hollow Heaps are a type of priority queue that introduce the concept of hollow nodes which are nodes that no longer contain a valid item but are still part of the heap structure [9]. Unlike Binary Heaps and Fibonacci Heaps, where each node directly represents a prioritized value, Hollow Heaps store items within nodes, allowing a node to become "hollow" when its item is removed or moved. When a node holds an item, it is referred to as a full node. Once the item is detached, for example during a *decrease_key* operation, the node becomes hollow and cannot hold another item. These hollow nodes are left in the heap until they are explicitly deleted. This lazy deletion strategy enables efficient restructuring and contributes to the heap's improved performance in certain operations.

Several variants of Hollow Heaps exist, including multi-root, one-root, and two-parent structures [24]. In this thesis, we focus on the two parent, single-root Hollow Heap, which forms a directed acyclic graph (DAG). This DAG structure sets it apart from the tree-based Binary and Fibonacci Heaps.

The DAG structure is achieved through two types of node links: ranked and unranked. Ranked links are used to connect two equally ranked nodes into a singular tree which affects the rank of the new parent node [24]. The unranked links are instead used to connect two nodes, either with unequal rank or where one is hollow, without affecting the rank. In the two-parent variant, a node may have one or two parents depending on whether it is full or hollow [24]. The use of both link types enables lazy deletions and ensures that structural adjustments can be postponed, making operations like *decrease_key* more efficient than in

Binary Heaps.

The basic operations in a Hollow Heap are defined as follows. Insertion is performed by adding a new full node as a child of the root. No restructuring is required, making insertion a constant-time operation [8]. The *extract_min* operation removes the root (which holds the minimum key) and performs consolidation by applying ranked and unranked links until a single-root structure is restored. This is the only operation that triggers significant restructuring of the heap.

The *decrease_key* operation is implemented by hollowing out the node containing the old item, creating a new full node with the updated key, and inserting it into the structure. If the new node is related to the old node (e.g. if it becomes an ancestor), the old hollow node is updated to include the new node as a second parent. This strategy avoids costly structural changes and is a key reason Hollow Heaps can offer constant amortized time for *decrease_key*.

It is therefore the Hollow Heap's ability to delay restructuring through hollow nodes and flexible linking that provides the theoretical advantages.

2.2 Time Complexity

In order to evaluate and compare the performance of the heaps, we analyze the time complexity of their fundamental operations, such as *insert*, *extract_min*, and *decrease_key*. Time complexity describes how the runtime of an individual operation grows with input size, and serves as a valuable predictor for performance under varying workloads.

The table 2.1 summarizes the time complexities of key operations for the Binary, Fibonacci and Hollow Heap [31] [6, 7] [9] [24]. The time complexities are expressed in Big O notation, which abstracts the cost of operations in terms of input size n, defined here as the number of nodes stored in the heap. Where applicable, the table specifies whether the time complexity refers to worst-case or amortized performance [6, 7], [9], [31].

Table 2.1: Time Complexity for Binary Heap, Fibonacci Heap, and Hollow Heap

Operation	Binary Heap	Fibonacci Heap	Hollow Heap
Insert	$O(\log n)$	$O(1)$ (amortized)	$O(1)$ (amortized)
Extract Min	$O(\log n)$	$O(\log n)$ (amortized)	$O(\log n)$ (amortized)
Decrease Key	$O(\log n)$	$O(1)$ (amortized)	$O(1)$ (amortized)
Merge	$O(n)$	$O(1)$	$O(1)$
Build Heap	$O(n)$	$O(n)$	$O(n)$

Worst-case complexity highlights the upper bound of an operation's running time under the most demanding conditions. However, this can be overly pessimistic for many practical scenarios. In contrast, amortized time complexity provides a more nuanced measure by analyzing the average cost per operation over a sequence of operations [29]. Rather than evaluating the maximum time for a single operation, it considers the cumulative cost of a series and distributes it evenly, giving a more balanced view of performance over time [4, p. 452]. This means that even if one operation in the sequence takes a lot of time, the average time per operation can still be low. Thus, the amortized analysis provides a more balanced perspective

by accounting for the interactions between operations and avoiding the pessimism inherent in worst-case analysis [29].

In practical terms, amortized analysis helps explain why certain operations that appear expensive in isolation, such as decrease key in Fibonacci or Hollow Heaps, can still offer efficient performance when evaluated across multiple uses. For example, while a single *decrease_key* operation may involve restructuring or lazy deletions, the cost of these steps is distributed over a series of insertions and deletions, resulting in an amortized constant or logarithmic time [6, 7]. This method of analysis is particularly useful when the distribution of operations, such as insertions, deletions, and *decrease_key* updates, varies with the application, as it does in the heap-based priority queues analyzed in this thesis.

2.3 Dijkstra's Algorithm

Heaps can be used to implement a priority queue, which is a component in the efficient implementation of Dijkstra's algorithm. The algorithm computes the shortest paths in a directed graph from a given source node to all other nodes [4, p. 658]. For each node, excluding the source, it identifies all possible predecessors and evaluates their respective shortest paths from the source to the current node [24].

The core idea of the algorithm is to iteratively expand the node with the smallest known tentative distance from the source, updating the distances to its neighboring nodes. By tentative distance is meant the currently known shortest distance from the source to a given node, which may be updated as shorter paths are discovered during the algorithm's execution. The algorithm maintains two sets of nodes: one containing nodes whose shortest paths have not yet been finalized, and another containing those for which the shortest path has already been determined [24].

A priority queue is used to efficiently select the next node with the smallest tentative distance. When a shorter path to a node is discovered, the priority queue is updated using a *decrease_key* operation. The *decrease_key* operation lowers the priority of the node in the queue, reflecting that a shorter and therefore more favorable path to this node has been found. The process continues until all reachable nodes have been finalized and their shortest paths from the source are known.

A naive implementation that scans all nodes to find the minimum at each step has a time complexity of $O(n^2)$, where n is the number of nodes. However, by using a heap-based priority queue to manage node selection and updates, this can be improved to $O((n+m)\log n)$, where m is the number of edges in the graph [3]. The efficiency of the priority queue which is particularly the operations *insert*, *extract_min*, and *decrease_key* will therefore play a critical role in the overall performance of the algorithm. These operations have different levels of importance depending on the density of the graph, which will be further discussed in Section 2.4.

The process of the algorithm is visualized in Figure 2.2, which shows the progression of Dijkstra's algorithm as it builds the shortest-path tree from the source node a. Each subfigure illustrates the state of the algorithm after finalizing the shortest path to a new node, selected using the priority queue based on its current tentative distance. The bold edges indicate the finalized shortest paths and illustrate how the shortest-path tree incrementally expands to cover all reachable nodes. This visualization highlights the algorithm's greedy nature and how

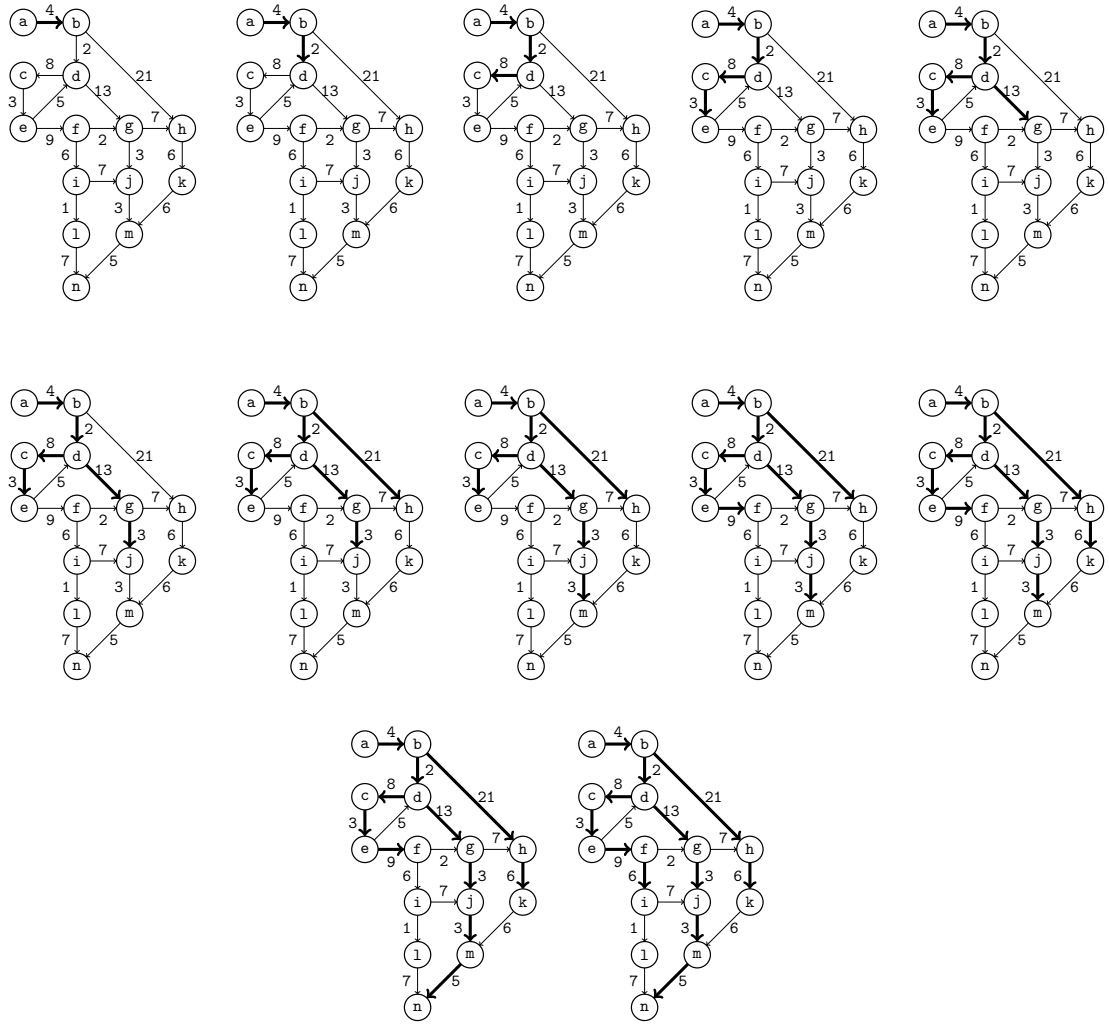


Figure 2.2: Dijkstra's algorithm finding the shortest paths from node a [24].

local decisions based on the minimum distance lead to globally optimal paths. The algorithm is greedy in the sense that, at each step, it selects the node with the current smallest tentative distance which is the locally the best choice in that moment, even though it does not yet have full knowledge of the complete graph.

For example, as seen in the figure, once node b is finalized, node d is selected next because it has the smallest tentative distance (weight 2) among the unexplored nodes. This selection is made even though more complex routes could lead to shorter paths for other nodes later. As the algorithm proceeds, nodes d, c, f, and others are added to the finalized set one by one, always based on the locally optimal choice. By repeatedly selecting the node with the smallest current distance and updating neighbors accordingly, the algorithm guarantees that once a node's shortest path is finalized, no shorter path will be found later. For instance, the path from node a to node e goes through $a \rightarrow b \rightarrow d \rightarrow c \rightarrow e$.

2.4 Graph Characteristics

In the context of shortest path algorithms, the structural properties of a graph significantly influence computational efficiency and feasibility [16]. Several key characteristics determine how algorithms like Dijkstra's perform in different graph topologies [4]. These properties include connectivity, density, and the nature of edge weights, all of which affect both the correctness and runtime of the algorithm.

A *strongly connected* graph is a directed graph in which there exists a path between every pair of nodes [24, p.94]. This property ensures that shortest path computations are meaningful for all node pairs, as every node is reachable from any other node. In contrast, a *weakly connected* graph remains connected only if edge directions are disregarded. This can lead to scenarios where no directed path exists from one node to another, meaning that there may be no paths at all between some pairs of nodes. As a result, shortest-path computations may be undefined in weakly connected graphs depending on the choice of source node.

Another important characteristic is whether the graph is *sparse* or *dense*. A sparse graph has significantly fewer edges m compared to the maximum possible number of edges $n(n-1)$ in a directed graph ($m \ll n^2$), whereas a dense graph has an edge count approaching this upper bound ($m \approx n^2$) [3, p. 36]. The efficiency of Dijkstra's algorithm varies depending on this distinction, as different priority queue implementation benefits different heap operations. This is because the number of edge relaxation steps, which dominates the runtime, is proportional to the number of edges. In sparse graphs, where each node has relatively few neighbors, the number of *decrease_key* operations is lower, making the efficiency of this operation particularly impactful. The density of a graph can be quantified as the ratio $m/n(n-1)$, which gives an indication of how close the graph is to being fully connected.

Furthermore, graphs can be categorized based on edge weights. A *weighted graph* assigns a numerical value to each edge, representing cost, distance, or another metric relevant to the problem domain. If all edge weights are non-negative, Dijkstra's algorithm remains applicable and produces correct shortest paths. However, in the presence of negative edge weights, the algorithm may produce incorrect results or enter an infinite loop. In such cases, alternative algorithms like Bellman-Ford must be used, as they are designed to handle negative weights and detect negative-weight cycles. Therefore, in this thesis, edge weights are assumed to be non-negative to ensure the correctness of Dijkstra's algorithm.

We restrict the study to Dijkstra's algorithm because of its ability to stress test the key operations of each heap relevant to this thesis. This makes it possible to evaluate how they perform in a sequence of operations rather than for a singular operation. The ability to scale the graphs, both in total amount of nodes and number of edges, and compare results creates a practical application of the heap. In addition, the operational environment created by Dijkstra's algorithm mirrors the expected usage context in the Arc project, where *insert*, *extract_min*, and *decrease_key* operations are used in combination.

2.5 Programming languages

The programming language used to implement a data structure can have a significant impact on both its runtime characteristics and its maintainability [23]. This is particularly relevant in performance-sensitive applications, such as those found in embedded or real-time systems.

Each language offers its own model for memory management, type safety and abstraction, which in turn influences how efficiently a particular data structure can be implemented and integrated. The choice of compiler, build system, libraries and environment also affects the usage which also affects the overall performance.

In this study, three languages were selected for implementing and comparing heap structures: C, Java, and Rust. These languages represent different trade-offs along the spectrum of control, safety, and abstraction. C offers direct memory access and minimal runtime overhead, making it a common choice in programming [22]. Java, with its object oriented, managed memory and garbage collector provides a higher-level alternative that simplifies development [17]. Rust introduces a modern systems programming model with memory safety guarantees through ownership and borrowing, while still aiming for performance close to that of C [12].

This section outlines the characteristics of each language relevant to the implementation of priority queues, particularly within the context of Panasonic Avionics Arc, where system performance and ease of integration are important factors.

2.5.1 C

The C programming language is an old language created in the 1970s. The idea behind the language was to create a low-level, portable, and type-safe language. The language had a wide and fast spread that led to further development of C compilers for different machines and operating systems [20].

C is widely used in embedded systems, operating system kernels, and other performance-critical domains due to its low-level capabilities and minimal runtime overhead [19]. The use of pointers allows direct memory access and manipulation, which is essential in systems where fine-grained control is required. C also supports modular development through header files and function-based organization, this in turn promotes reusable code. Its long history and broad adoption have resulted in a large ecosystem of libraries, tools, and community support that can simplify development and debugging.

Memory management in C is entirely manual and is the developer's responsibility. There are several ways of using the memory and both static and dynamic memory is to be used. Depending on the goal of the program, the memory usage can be more or less efficient as well as creating unnecessary bugs and complex errors. In C there are several ways of allocating memory, such as malloc and calloc, using dynamic memory. The positive aspects of dynamic memory is the requested memory during runtime, opening up for not knowing the size in advance. Memory can also be reallocated which allocated the new memory needed.

The principles and design choices behind C have laid the foundation for many modern programming languages. For this reason, C is often referred to as the "mother of programming languages."

2.5.2 Rust

Rust is a systems programming language first released in 2006, designed to offer memory safety without relying on a garbage collector. The language introduces a model based on ownership, borrowing, and lifetimes, which is enforced at compile time to prevent common memory errors such as dangling pointers and data races. This model allows Rust to

manage memory efficiently while supporting concurrency, making it a candidate for use in performance-critical and safety-sensitive systems [12].

Rust uses structures and encapsulation making it possible to handle certain part of development as object oriented, even though the language is not object oriented by standard. By doing this, together with its ownership rules, the language is both flexible and complex. Other aspects of rust is its way of handling errors. The compiler is very informative, giving the developer a lot of feedback regarding the current error and proposals on how to solve it. Rust also provides both low-level memory access and high-level structures adding to the flexibility of the language [12].

2.5.3 Java

Java is a high-level, widely used, object-oriented languages initially released in 1995 [17]. Due to its class-based structure and simplicity, the language is used in several domains; both in the academia and industries. Java has important features such as its security and platform independence that makes the language suitable for games, back-end development and numerical operations [17]. The language also has a large user base making it possible to use already developed structures and classes as well as finding help and feedback in the community. This together with the robustness of the language itself makes the language reliable and versatile.

Memory management in Java is handled at the level of the runtime environment rather than explicitly by the developer. The Java Virtual Machine (JVM) automatically allocates and reclaims memory through garbage collection, which tracks object reachability and reuses memory no longer in use [17]. In addition to garbage collection, the Just-In-Time (JIT) compiler optimizes runtime performance by compiling frequently executed bytecode into native machine code. Together, these features abstract away manual memory handling while still enabling competitive performance in long-running applications.

Java provides a standard library and a large set of external modules. These offer general-purpose data structures and utility functions. Developers can use these directly instead of reimplementing common functionality. The language uses a consistent syntax, which, together with widespread use and documentation, can make it easier to get started for new users [21].

2.6 Cache Memory

Cache memory is used to minimize the delay caused by memory access. Efficient data structures rely heavily on fast memory access. Modern computer architectures are built on a hierarchical memory model where different levels of memory trade off between size and access speed [27]. Cache memory resides closer to the central processing unit (CPU) than main memory and provides significantly faster access times. This difference in speed is managed by leveraging the principle of locality of reference, which describes how programs tend to access memory in predictable patterns. Locality is typically divided into two forms. Firstly, temporal locality, where recently accessed data is likely to be accessed again soon. Secondly, spatial locality, where data located near recently accessed memory addresses is likely to be accessed shortly thereafter [27].

Due to the limited size of cache memory, only a subset of all data can reside in it at any time. When the CPU requests a memory address that is already in the cache, the access is fast and results in a cache hit. If the requested address is not in the cache, a cache miss occurs [27]. In that case, the processor must fetch the data from main memory, which introduces a significant delay. The processor does not retrieve only the specific data element it requested, it instead loads an entire cache line, a block of contiguous memory (typically 64 bytes) that contains the target address. This strategy attempts to take advantage of spatial locality. However, for data structures with irregular or pointer-heavy layouts, such as those involving scattered node allocations, this behavior can lead to performance inefficiencies. The additional data brought in with the cache line may not be useful, wasting memory bandwidth and further increasing latency.

This issue is particularly relevant for heap operations in data structures such as Fibonacci Heaps and Hollow Heaps. Operations like *extract_min*, *decrease_key*, and *consolidation* often involve accessing multiple nodes linked via pointers. These nodes are typically not stored contiguously in memory, resulting in poor spatial locality and frequent cache misses. When the heap grows larger than the cache or when its nodes are poorly aligned in memory, the rate of cache misses can increase significantly. As a result, the processor spends more time waiting for data to arrive from main memory, which reduces overall performance. The impact is especially noticeable in pointer-based heaps, where irregular access patterns prevent the efficient use of the cache.

2.7 Prefetching

As modern processors continue to outpace memory speeds, memory latency has become a critical bottleneck in high-performance computing. One widely used technique to mitigate this latency is prefetching. Prefetching involves loading data into the processor's cache before it is actually needed by the program, thereby overlapping memory access latency with computation [11]. If done effectively, this can significantly reduce the time a processor spends waiting for data to arrive from main memory.

For programs that operate on arrays or other data structures with regular memory access patterns, prefetching can be highly effective [11]. Hardware prefetchers can detect sequential or strided accesses and automatically fetch upcoming data into the cache. Similarly, software-based prefetching, where the compiler or programmer inserts explicit prefetch instructions, can also be used to hide latency in predictable memory access patterns [11].

Pointer-based data structures, such as linked lists, trees, and graphs, exhibit poor spatial locality because their elements are dynamically allocated and not necessarily contiguous in memory [15]. Additionally, their unpredictable traversal order complicates prefetching, which must be carefully scheduled to maximize latency hiding while minimizing unnecessary overhead. The primary challenge in prefetching RDSs lies in the number of pointer dereferences required to compute a future memory address, refered to by Luk and Mowry (1996) as "The Pointer Chasing Problem" [15].

To mitigate memory latency in RDSs, researchers Luk and Mowry have proposed three primary compiler-based prefetching schemes:

- Greedy Prefetching: This method leverages naturally occurring jump-pointers within an RDS. Since each node in an RDS typically contains multiple pointers, the compiler

can prefetch the ones not immediately used in the traversal. Although this approach does not ensure an optimal prefetch distance, it has minimal computational overhead and can significantly reduce cache miss penalties.

- History-Pointer Prefetching: This technique introduces artificial jump-pointers to explicitly store the addresses of nodes expected to be accessed in the near future. These pointers are updated during the first traversal of the data structure and utilized in subsequent traversals to prefetch data more accurately. While history-pointer prefetching improves accuracy, it requires additional storage and introduces overhead for maintaining the pointers.

- Data-Linearization Prefetching: In this approach, dynamically allocated nodes are arranged in contiguous memory locations, effectively transforming a pointer-based structure into an array-like layout. This transformation allows the next access address to be computed directly using simple arithmetic operations, eliminating the need for pointer dereferencing. This technique offers the most precise prefetching and enhances spatial locality but is only feasible when the structure of the RDS remains stable over time.

Software-controlled prefetching refers to the insertion of prefetch instructions by the compiler at compile time, based on an analysis of the program's control flow and memory access behavior. The goal is to hide memory latency by loading data into the cache before it is accessed. This technique is particularly relevant for pointer-based applications, where traditional hardware prefetchers struggle due to irregular access patterns and poor spatial locality [15]. Experimental results show that software-controlled prefetching can improve performance by up to 45% for certain workloads. More advanced strategies, such as history-pointer prefetching and data-linearization, aim to increase these gains by better predicting future memory accesses and reducing cache miss rates. Software-controlled prefetching has the advantage of leveraging application-specific knowledge, which allows it to outperform hardware-based methods in cases where access patterns are too complex or unpredictable for hardware to handle effectively [15]. Such complex access patterns may arise in advanced heap structures like the Fibonacci Heap or Hollow Heap, where memory layout and pointer traversals do not follow regular or easily predictable sequences.

Prefetching is relevant to this work because the aim is to optimize the data structures used in Arc. The efficiency of memory access plays an important role in overall performance, particularly for pointer-based structures. By evaluating how different heap implementations interact with prefetching mechanisms, it is possible to identify structural characteristics that either help or hinder memory locality. Although the original focus was on prefetching within a single heap, the evaluation of multiple heaps may reveal variation in behavior. If some structures perform better with respect to prefetching, this could indicate a potential advantage in memory-bound workloads. Prefetching analysis therefore complements traditional performance metrics by highlighting memory access patterns and their impact on cache efficiency.

2.8 Panasonic Avionics Arc

This thesis evaluates heap performance within the context of Panasonic Arc developed by Tactel. Tactel, a subsidiary of Panasonic Avionics, develops in-flight entertainment software

for long-haul flights. Arc is one of its product which is a high-resolution, interactive map feature designed for passenger engagement.

Arc provides a 3D globe view showing the aircraft's current position, destination, and flight path [28]. It supports standard map interactions such as zooming, panning, and exploring other aircraft routes. The map is rendered in 3D, with visual realism applied to mountains, cities, and streets. It has many features such as dynamic lighting, switching between light and dark modes to reflect the relative position of the Sun. The goal of Arc is to create a unique entertainment system that connects passengers with their surroundings through rich, location-aware content.

From discussions with Tactel, we have identified priority queues in how Arc determines which cities and landmarks are displayed on the map. The interactive globe consists of a number of layers, supported by a dataset of 35,000 data points representing cities and landmarks, including approximately 1,000 airports. To manage data, a selection algorithm prioritizes elements for display based on precomputed priority value, which considers factors such as whether a city is a capital or is associated with the airline in operation.

Currently, Arc employs a simple queue to select a subset of elements for rendering, with overlapping labels filtered based on their associated priority values. The implementation is effective in many cases, but it does have limitations, for example, when multiple elements with similar priorities compete for limited screen space. This can result in visual artifacts such as flickering, as the display rapidly alternates between competing labels.

By replacing the current queue with a more sophisticated heap which has been evaluted under similar conditions using Dijkstra's algorithm, it is possible to improve responsiveness and rendering stability. In particular, heap implementations that support efficient *decrease_key* operations, such as Fibonacci and Hollow Heaps, offer a more flexible mechanism for dynamically adjusting priorities in response to user interaction or contextual changes. This capability could mitigate flickering and enhance user experience without negatively impacting system performance.

Although Dijkstra's algorithm is not currently part of Arc's rendering pipeline, it offers a relevant benchmark due to its reliance on the same key heap operations. Moreover, should route calculations or dynamic pathfinding be incorporated into Arc in the future, instead of relying solely on precomputed data, efficient heap structures would become even more critical to maintaining interactive performance.

2.9 Related Work

Three important related works for this paper are the initial implementation and discovering of the heaps. In 1964 the Binary Heap [31] was developed as an algorithm called `HEAPSORT`. Two decades later, in 1984, the Fibonacci Heap [6, 7] was presented. Due to the complex structure of the Fibonacci Heap, the Hollow Heap [9] was introduces in 2015 as a simpler alternative, offering less intricate build up and more straightforward implementation.

Another paper which has used heaps for implemented Dijkstra's to solve the shortest path problem in weighted graphs is [14] where they compare the algorithm with a Fibonacci Heap, Binary Heap and a self-balancing binary tree. The findings were that the fastest method was not always the one with the lowest amortized complexity.

A related work investigating the time complexity of Dijkstras algorithm using Binary

Heap as priority structure is [2]. The time complexity for directed, weighted graphs were examined to evaluate the impact Binary Heap has when used in Dijkstras algorithm. The results showed a total time complexity of $O(|E| + |V| \log |V|)$ where **E** is the amount of Edges and **V** the vertices.

A related work on language evaluation, in which different programming languages were compared for use in evolutionary algorithms, is presented in [18]. The goal was to identify the most suitable language for implementing key algorithms such as mutation, crossover, and onemax, while also exploring potential alternatives to commonly used languages like C and Java. The study included a variety of language types to better understand their respective strengths and weaknesses. By incorporating less popular languages, the authors were able to highlight viable competitors within this domain. When evaluating performance and scalability, compiled languages generally outperformed others. Languages such as Go and Python also demonstrated fast execution times for many tasks. The study concluded that no single language was universally superior across all chromosome sizes and algorithms.

Chapter 3
Methodology

This chapter outlines the methodological approach followed throughout the execution of this master's thesis. It covers the initial investigation of the problem area followed by the development and testing process.

The project began with an investigation of the problem domain, carried out through discussions with our supervisor Jonas Skeppstedt and industry partner Tactel. These discussions helped define the research questions and identify the relevant problem areas for evaluation.

The focus centered on evaluating suitable priority queue implementations for the Arc application. Binary Heap, Fibonacci Heap, and Hollow Heap were selected for their contrasting internal structures and theoretical time complexities, particularly in operations such as *decrease_key*, which is central to Arc's usage patterns. Dijkstra's algorithm was chosen as the benchmark application due to its heavy reliance on priority queues and its resemblance to Arc's use case. Language evaluation candidates were also selected in collaboration with Tactel developers, to investigate alternatives to the current C++ implementation in Arc.

We studied the original publications of the selected heaps in detail to ensure our implementations were theoretically aligned. To establish a solid theoretical foundation and guide our implementation choices, we reviewed relevant literature and prior work. This review also informed the design of our evaluation framework, which we used to analyze and compare performance across implementations.

The initial development phase focused on implementing the three heap structures in C. The Binary Heap was developed first, followed by the Fibonacci and Hollow Heap. Pair programming was used throughout development to ensure code quality and understanding. Each implementation was extensively tested for correctness before proceeding to performance benchmarking.

Each heap was tested with small, medium, and large input sizes for all three core operations. Every test was repeated 10 times, with the mean value computed to account for variance and eliminate outliers. These early results provided practical insights into performance characteristics and informed our expectations for full Dijkstra runs.

For the language comparison, we selected the Fibonacci Heap due to its pointer-heavy

structure, which is well-suited to exposing differences between programming languages in memory handling. The Java implementation was written based on the published pseudocode and verified by comparing to our C implementation. For Rust, we adapted an existing open-source implementation [32] to ensure comparability with our own C and Java versions.

Once verified for correctness, the language benchmarks were conducted using the same operation tests as in the C evaluation. Each test was run 10 times, and mean values and standard deviations were calculated for each data size. The results were then analyzed to determine how language-level factors such as memory management and runtime overhead influenced performance.

To evaluate performance in an applied context, we implemented Dijkstra's algorithm for each of the three heaps. Due to structural differences between the heaps, separate versions of the algorithm were written for each. This approach ensured compatibility with each data structure and avoided performance bottlenecks due to inefficient integration.

Graphs of varying sizes and sparsity levels were used to test performance under realistic conditions. Sparse graphs were of particular interest, as they best reflect Arc's target use case of the map consisting of 35,000 elements. Although graph generation for dense graphs proved time-consuming and limited our ability to test large dense instances, the results obtained were sufficient for our goals.

To evaluate cache and memory behavior, we used the `valgrind` program's `cachegrind` tool alongside the operf profiler. We then introduced prefetching in an attempt to mitigate performance bottlenecks caused by pointer chasing, particularly in the Fibonacci and Hollow heaps. Code sections with high pointer indirection were identified as likely sources of cache inefficiencies, informed by the insights presented by Luk and Mowry [15]. As an exploratory step, we also implemented prefetching within Dijkstra's algorithm to observe its potential impact across different heap structures. However, this aspect of the study was not pursued further due to limited relevance to the industry partner's priorities and time constraints.

As an additional metric, we recorded the amount of dynamically allocated memory used by each heap structure during execution. This metric, while secondary, helped assess the potential system resource impact of each data structure in a production setting.

The final step was to aggregate, analyze and interpret the data that had been collected. Observations were drawn from both operation-level benchmarks and algorithm-level evaluations. The results were used to form conclusions and develop recommendations for Tactel regarding the most suitable heap and implementation language for Arc.

Chapter 4
Heap Implementation Details

This chapter presents the implementation details of each heap structure, detailing the data structures used and the sources that guided the implementations.

For the implementations in Java and Rust, the core design and logic are shared across implementations, minor differences arise due to the syntactic and semantic characteristics of each language. One notable distinction is memory management. In languages such as Java and Rust, memory is managed automatically through garbage collection or ownership semantics. In contrast, C requires explicit handling of dynamic memory. To address this, we adopted *malloc* as the standard allocation method throughout the development process.

For each heap variant, we provide a breakdown of the corresponding node and heap structures, including type definitions and field descriptions. These tables are intended to clarify the internal representations used in our implementations.

4.1 Binary Heap

The Binary Heap implementation follows the structure described in *Algorithms: A Concise Introduction* by Jonas Skeppstedt [24]. Binary heaps are array-based data structures that support efficient priority queue operations through the use of the *heapify* functions, which ensures that each parent node has a lower key (higher priority) than its children.

Below are tables summarizing the fields used in our implementation for both the Binary Heap nodes and the overall heap structure.

Table 4.1: Binary Heap Node Structure

Field	Type	Description
`id`	`size_t*`	Identifier of the node
`key`	`size_t`	Priority value of the node
`i`	`size_t`	Current Binary Array Index of this node

Table 4.2: Binary Heap Structure

Field	Type	Description
`nodes`	`BinaryNode*[]`	Array of pointers to nodes of type `BinaryNode`, representing the heap
`capacity`	`size_t*`	Maximum number of elements of the heap
`i`	`int`	Current number of nodes in the heap

4.2 Fibonacci Heap

The Fibonacci Heap implementation draws upon the original papers by Fredman and Tarjan [6, 7], which describe a pointer-based heap structure optimized for amortized efficiency, especially for *decrease_key*. Our design is also influenced by online teaching materials from York University [10], which provide clear procedural pseudocode and illustrations.

The node structure supports circular doubly-linked root lists, marking, and parent-child relationships. The heap keeps track of the minimum node and total size.

In the tables below, the name and type of each field is described shortly. This struct creates the Fibonacci Node which together builds up the Fibonacci Heap.

Table 4.3: Fibonacci Heap Node Structure

Field	Type	Description
`parent`	`FibonacciNode*`	Pointer to the parent node
`child`	`FibonacciNode*`	Pointer to one of the node's children (children form a circular list)
`left`	`FibonacciNode*`	Pointer to the left sibling (siblings form a circular doubly-linked list)
`right`	`FibonacciNode*`	Pointer to the right sibling (siblings form a circular doubly-linked list)
`id`	`int`	Identifier of the node
`key`	`int`	Priority value of the node
`marked`	`int`	Indicating whether the node is used in cascading cuts (1) or not (0)
`ranked`	`int`	Number of children currently attached to this node

Table 4.4: Fibonacci Heap Structure

Field	Type	Description
`min`	`FibonacciNode*`	Pointer to the root node (minimum node)
`nbrOfNodes`	`int`	Total number of nodes in the heap

4.3 Hollow Heap

The implementation of the Hollow Heap presented in this thesis is based on the description provided in the book *Algorithms: A Concise Introduction* by Jonas Skeppstedt [24], along with the original research article that introduced the data structure [9]. Hollow Heaps use a node design in which nodes can be made hollow during the *decrease_key* operation. This mechanism eliminates the need for auxiliary mechanisms such as cascading cuts in Fibonacci heaps.

Each node in a Hollow Heap is either full, meaning it contains an item, or hollow, meaning it serves as a structural placeholder where item is set to `NULL`. Hollow nodes preserve the necessary connections within the heap to enable delayed restructuring.

The following tables describe the fields and corresponding types used in the implementation, both for individual nodes and for the overall heap structure.

Table 4.5: Hollow Heap Node Structure

Field	Type	Description
`parent`	`HollowNode*`	Pointer to the parent node
`second_parent`	`HollowNode*`	Points to the second parent
`next`	`HollowNode*`	Pointer to the next sibling in the list of children
`child`	`HollowNode*`	Pointer to the first child node
`item`	`Item*`	Pointer to the item this node represents (null for hollow nodes)
`key`	`int`	Priority value of the node
`hollow`	`int`	Indicating whether the node is hollow (1) or full (0)
`rank`	`int`	Number of children attached to this node

The heap itself maintains a pointer to the root node (which must be full) and the total number of elements.

Table 4.6: Hollow Heap Structure

Field	Type	Description
`root`	`HollowNode*`	Pointer to the root node of the heap (minimum node)
`size`	`int`	Total number of nodes in the heap
`max_rank`	`int`	Maximum rank allowed in the heap

4.4 Prefetching in Heaps Data Structure

To reduce the penalties associated with cache misses, software prefetching was applied using the `__builtin_prefetch` function. This technique attempts to load data into the cache before it is accessed during program execution, thereby reducing the likelihood of pipeline stalls due to memory latency. The prefetching function takes a memory address, a read/write indicator, and a temporal locality hint as arguments. Prefetch instructions were inserted at locations in the code where repeated pointer dereferencing created predictable memory access patterns that could potentially cause cache stalls. Such patterns are common in heap

operations that involve tree or list traversal. By issuing prefetch requests ahead of time in these contexts, the goal was to ensure that the required data would already be available in the cache when accessed.

In the Binary Heap, prefetching was applied to the *up_heapify* and *down_heapify* operations. These functions were chosen because they involve repeated parent-child pointer accesses, which exhibit predictable memory access patterns due to the tree traversal structure. In *up_heapify*, prefetching is implemented to optimize parent node access. When a node needs to be moved up the heap, the parent node is prefetched before the comparison takes place, with the aim that the data will already be in cache when needed. Similarly, *down_heapify* involves traversing the tree downward by comparing and potentially swapping with child nodes, making it another suitable target for prefetching. Therefore, these two operations were selected for prefetch insertion, as they represent the core traversal mechanisms in the Binary Heap.

In the *down_heapify* function, two cases were considered. If the node has only a left child, a single prefetch is issued. If the node has both left and right children, prefetches are issued for both. The goal is to ensure that both child nodes are present in cache before selecting the minimum.

For Fibonacci Heap, prefetching was applied to three operations. We selected *extract_min*, *consolidate*, and *link* based on code inspection, focusing on operations with frequent pointer dereferencing and predictable traversal patterns. The variables described in the Algorithms 1, 2 and 3 from functions in the Fibonacci Heap have the following types:

- `fibonacci_node*`: pointers to nodes including `x`, `y`, `child`, `nextChild`, `other`, and `current`.
- `fibonacci_heap*`: the heap structure passed to functions like *link*.
- `int`: variables such as `rank` and `degree`.
- `fibonacci_node* ranks[]`: array mapping degrees to node pointers during *consolidate*.

In *extract_min*, prefetching is used during the traversal of the circular doubly linked list of children. Each iteration prefetches the `right` pointer of the next child node, so that it is available in the cache in the following iteration.

Algorithm 1 Prefetching in Fibonacci Heap *extract_min*

```
1: while child ≠ extract_node.child do
2:     nextChild ← child.right
3:     __builtin_prefetch(nextChild.right)
4:     insert_rootlist(child, heap)
5:     child.parent ← NULL
6:     child ← nextChild
7: end while
```

In the *consolidate* operation, which merges trees of equal degree, a prefetch is issued for the `right` pointer of both the current node and the merge candidate node `other`. The candidate node is fetched from the rank table, which maps degrees to nodes.

Algorithm 2 Prefetching in Fibonacci Heap *consolidate*

```
1:  while current ≠ last do
2:      next ← current.right
3:      __builtin_prefetch(next.right)
4:      degree ← current.rank
5:      while ranks[degree] ≠ NULL do
6:          other ← ranks[degree]
7:          __builtin_prefetch(other.right)
8:          /* ... merging logic ... */
9:      end while
10: end while
```

The reason for this optimization is because the operation involves multiple levels of pointer chasing and it processes the entire root list to potentially merge trees. These means that each merge operation requires accessing multiple node pointers.

In *link*, prefetches are issued for the `left` and `right` pointers of the target node, and for the child list if it exists. The goal is to reduce latency when manipulating the child pointers of a node during the link process.

Algorithm 3 Prefetching in Fibonacci Heap *link*

```
1:  procedure LINK(heap, x, y)
2:      __builtin_prefetch(y.right)
3:      __builtin_prefetch(y.left)
4:      /* ... existing code ... */
5:      if x.child ≠ NULL then
6:          __builtin_prefetch(x.child.left)
7:          * ... child list manipulation ... */
8:      end if
9:  end procedure
```

Moving from the Fibonacci Heap to the Hollow Heap, we turn our attention to the *delete* function, which is invoked during *extract_min*. In the *delete* function, the sibling list of a node's children is traversed. Prefetching is applied to the next sibling node to reduce latency in the next iteration.

In the Hollow Heap code, the following types apply:

- `hollow_node_t*`: pointers to nodes such as `child`, `next`, and `current`.
- `int`: used for `rank` in operations such as linking.
- `hollow_node_t* A[]`: array indexed by rank, holding node pointers during delete and linking operations.

Algorithm 4 Prefetching in Hollow Heap *delete* (traversal part)

```
while current ≠ NULL do
    next ← current.next
    if next ≠ NULL then
        __builtin_prefetch(next, 0, 1)
    end if
    /* ... existing code ... */
    current ← next
end while
```

The delete function also accesses the rank array when performing repeated links. Prefetching is done for the entry corresponding to the current node's rank, before the link is performed.

Algorithm 5 Prefetching in Hollow Heap *delete* (linking part)

```
1: while A[current.rank] ≠ NULL do
2:     __builtin_prefetch(A[current.rank], 0, 1)
3:     current ← link(current, A[current.rank])
4:     A[current.rank++] ← NULL
5: end while
```

The *free_node* function is invoked by *free_heap* , which is called at the end of Dijkstra's algorithm to deallocate the heap structure. In the *free_node* function of the Hollow Heap, the child list is traversed to free all descendants of a node. A prefetch is issued for the next child node before the pointer is updated.

In the Algorithm 6, both `child` and `next` are pointers of type `hollow_node_t*`, which represent individual nodes in the Hollow Heap structure.

Algorithm 6 Prefetching in Hollow Heap *free_node*

```
1: while child ≠ NULL do
2:     next ← child.next
3:     if next ≠ NULL then
4:         __builtin_prefetch(next, 0, 1)
5:     end if
6:     /* ... existing code ... */
7:     child ← next
8: end while
```

These specific prefetch points were selected through code inspection and light profiling test with `operf`, which is part of OProfile for Linux. The locations are at points in the code where the memory access patterns are predictable and where pointer chasing causes delays. These additions allow us to measure whether software prefetching can improve memory performance for pointer-heavy data structures.

As we will see in Section 6.5, `cachegrind` does not register the software prefetch, as no noticeable effects were observed. To validate whether manual prefetching resulted in

measurable improvements in cache usage, additional profiling was carried out using operf with the following command:

```
operf -e PM_LD_MISS_L1:10000 ./a.out
opannotate -s
```

This configuration tracks L1 data cache load misses, with one sample taken every 10 000 misses. The opannotate tool was then used to map these misses back to source code lines. Note that on the POWER8 architecture used for this test, cache blocks are 128 bytes. While the exact CPU architecture is not the focus of this study, the profiling confirmed that software prefetching did reduce L1 miss rates in several of the annotated regions. These results are discussed in more detail in Section 6.5.

4.5 Prefetching in Dijkstra's Algorithm

To evaluate the potential benefits of software prefetching in Dijkstra's algorithm, a smaller-scale experiment was conducted by adding prefetch instructions to the main loop. Specifically, the prefetch was applied to the data structure representing the outgoing edges of the current node before accessing them. The implementation of prefetching varied slightly depending on the heap structure used in the priority queue.

In the Binary Heap version, a prefetch instruction was added before accessing each edge of the current node:

```
for (size_t i = 0; i < g->vertex[u].d; i++) {
        __builtin_prefetch(&g->vertex[u].edge[i+1], 1, 3);
      size_t v = g->vertex[u].edge[i].v;
```

For the Fibonacci Heap implementation, a similar approach was used. Note the offset in the prefetch index:

```
      for (size_t i = 0; i < u_vertex->d; i++) {
        __builtin_prefetch(&u_vertex->edge[i+1], 1, 3);
            edge_t* edge = &u_vertex->edge[i];
```

In the Hollow Heap version, prefetching was also applied with an offset:

```
 vertex_t *vtx = &g->vertex[u];
    for (size_t i = 0; i < vtx->d; i++) {
        __builtin_prefetch(&vtx->edge[i+1], 1, 3);
      edge_t *e = &vtx->edge[i]
```

The offset of `i+1` was used to prefetch the next edge in advance of its use, allowing the processor time to load it into cache while the current edge is being processed. This leverages both spatial and temporal locality as edge data is stored contiguously in memory (spatial locality), and consecutive edges are accessed in rapid succession within a tight loop (temporal locality).

Chapter 5
Evaluation

This chapter outlines the methods used to evaluate the performance and behavior of the three heap data structures. The evaluation focuses on ensuring fair and consistent testing across the different implementations. In addition we present how the test environments were set up, how prefetching, memory usage and cache behavior were evaluated. We also present how the heaps were tested as priority queues in Dijkstra's algorithm on graphs of varying sizes and densities.

5.1 Test Environment

The computer setup was provided by the Industry Partner Tactel. The computer was allocated to us for the duration of the thesis work in order to replicate a test environment similar to that used by the developers in the Arc Project. The second set-up, provided by Jonas Skeppstedt, allowed for profiling with operf.

All benchmarks and evaluations tests were executed locally on an Apple MacBook Pro equipped with an M1 Max chip, 32 GB of unified memory, and running macOS Ventura version 15.4.1 (build 24E263). The following compiler version was used:

```
Apple clang version 17.0.0 (clang-1700.0.13.3)
Target: arm64-apple-darwin 24.4.0
```

To be able to perform tests in a Linux environment with `valgrind` and `cachegrind`, all memory profiling and cache behavior benchmarks were conducted inside a Docker container. The Docker container used for these tests was based on a Linux environment with GCC version:

```
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
```

The prefetching measurements using operf were conducted on a 20-core IBM S822L machine running Debian Linux. All tests were compiled with the system's default GNU Compiler Collection, specifically `GCC version 13.2.0`.

5.2 Execution Time Test

A series of benchmarks tests were conducted measuring the average execution time of key functions to evaluate the performance of the heap operations in a practical setting. These tests aim to capture the practical latency of operations such as insertion, *extract_min* and *decrease_key* across the three different heap implementations.

When conducting tests on the extract min and decrease key operation, a double amount of insertions are performed beforehand. Meaning that if 500 extract min operations are to be tested, 1,000 nodes has been inserted. For Fibonacci and Hollow heap, the nodes and items that are inserted are saved to be able to free the objects after the conducted test. These insertions and deallocations are not included in the recorded execution time and will therefore not affect the measured performance.

The following tests were conducted to evaluate the actual run-time, executed values, for each operation in Binary, Fibonacci and Hollow Heap:

1.1 100 Insertions

1.2 1000 Insertions

1.3 100,000 Insertions

1.4 500,000 Insertions

1.5 1,000,000 Insertions

1.6 100 Extract Min

1.7 1000 Extract Min

1.8 100,000 Insertions

1.9 500,000 Extract Min

1.10 1,000,000 Extract Min

1.11 100 Decrease Key

1.12 1000 Decrease Key

1.13 100,000 Insertions

1.14 500,000 Decrease Key

1.15 1,000,000 Decrease Key

The tests were conducted 10 times each and a mean value were then calculated. This value was further divided by the number of operations to create a mean value per operation for each function. The value are presented in nanoseconds to find small changes in runtime values.

5.3 C, Java and Rust Test

We evaluated the environment and usage of different programming languages on the performance and implementation complexity of heap data structures. The Fibonacci Heap was selected for this study due to its pointer intensive structure and dynamic memory usage. This made it a suitable candidate for observing differences across languages with varying memory management models.

The Fibonacci Heap was chosen over the Hollow Heap because it had faster execution time when implemented in C, which we will see in 6.1.

The Fibonacci Heap was implemented in C, Java and Rust. The use of a consistent algorithmic structure across all implementations isolates the effect of each programming language. The Rust implementation is based on [32], with modifications made to ensure comparability with the Java and C implementations. Since C requires manual memory management, this test highlights how pointer-heavy structures behave in environments with explicit allocation and deallocation. In Rust, the ownership and borrowing system was used to manage memory without garbage collection. In Java, the implementation relied on automatic memory management via garbage collection.

Heaps of varying sizes were used to capture a broader range of performance characteristics. Input sizes were categorized as small for 500 operations, medium for 35,000; and large for both 500,000 and 1,000,000 to help identify potential performance shifts or constant overheads that may not be apparent when testing with a single dataset size.

The test case of 35,000 is based on the Arc map dataset, which includes 35,000 locations, making it a relevant reference point for evaluating heap performance at this scale.

2.1 500 Insertions

2.2 35,000 Insertions

2.3 500,000 Insertions

2.4 1,000,000 Insertions

2.5 500 Extract Min

2.6 35,000 Extract Min

2.7 500,000 Extract Min

2.8 1,000,000 Extract Min

2.9 500 Decrease Key

2.10 35,000 Decrease Key

2.11 500,000 Decrease Key

2.12 1,000,000 Decrease Key

Each test was executed 10 independent times to be able to calculate a mean for the execution time per operation. A standard deviation was also calculated to measure the variability. The resulting mean was divided by the number of operations performed, yielding a per-operation time for each test. All results are presented in nanoseconds.

5.4 Dijkstra's Algorithm Test

Dijkstra's algorithm is used in this study as a practical test case to evaluate the performance of the heap structures when used as priority queues. The role of the heap in the algorithm is to manage the selection of the next node with the smallest tentative distance during each iteration.

Separate implementations of the algorithm were developed for each heap type in C. This was necessary due to differences in the internal behavior and memory handling across the heap structures. For example, the Hollow Heap performs the *decrease_key* operation by creating a new node and marking the original as hollow, which introduces the need for additional logic to maintain consistent node references. In contrast, Binary and Fibonacci Heaps follow a more direct approach, modifying nodes in place. As a result, requiring heap-specific implementation for the algorithm was needed to avoid memory leaks or undefined behavior.

To evaluate the heaps under Dijkstra's algorithm, a range of synthetic graphs were generated. The graphs were created to allow for controlled variation in both size and density, in order to ensure consistency and reproducibility across all tests.

Performance is measured as the total execution time of the algorithm from a single source node, covering the complete pathfinding process across the input graph. This approach allows for direct comparisons between heap types under consistent algorithmic conditions.

For sparse graphs, the number of nodes ranged from 1,000 to 70,000. For dense graphs, the range was limited to 1,000 to 25,000 nodes. The upper limit for dense graphs was reduced due to computational constraints related to the graph generation process. To be more precise, it was related to the overhead of checking for duplicate edges using hash tables and the increased number of edges required for strong connectivity in dense topologies. As graph density increases, the number of edges approaches n^2, making both memory usage and preprocessing time (such as computing strongly connected components) significantly more expensive.

The graph generation code was provided by our supervisor, Jonas Skeppstedt. It uses a randomized approach in which vertices are connected by weighted edges. Each edge is assigned a randomly generated positive weight, ranging from 1 to a maximum equal to the number of nodes in the graph. The graph construction occurs in two passes, first an initial pass randomly assigns edges, and a second pass ensures full connectivity by adding any missing links necessary for the graph to be strongly connected. This step is critical, as Dijkstra's algorithm requires the ability to reach all nodes from the source.

To maintain deterministic behavior across multiple runs and heap implementations, a fixed seed was used to initialize the random number generator during graph generation. There is a seed used to initialize the random number generated through srand(seed) in the *make_rand_graph* function that generates a random graph. The fixed seed means that the function produces the same random graph every time, allowing for reproducibility in the testing.

For each test, a new graph was generated using the parameters for number of nodes, number of edges, and maximum edge weight, where the maximum weight was set equal to the number of nodes. These parameters were supplied as input to the graph generation function. To minimize the influence of runtime variability, each configuration of Dijkstra's algorithm was executed three times on each graph, and the average execution time was recorded as the final result.

5.5 Memory Usage

Since memory usage can influence execution time, it is an important factor to consider when evaluating the performance of different heap implementations in Dijkstra's algorithm. Understanding how efficiently each heap handles memory can help explain differences in runtime behavior. Therefore, we have chosen to observe the memory consumption of each heap implementation during a test that reflects the scale of the Arc dataset.

To analyze the memory consumption characteristics of each heap implementation, measurements were performed using the `massif` tool in `valgrind` and runtime allocation diagnostics. The goal was to quantify total memory allocations, frees, and overall heap usage during a representative execution scenario.

For this purpose, each heap was evaluated using a graph with 35,000 nodes and 105,000 edges. This graph size was selected to reflect the scale of the Arc dataset, serving as a representative benchmark for the application context under consideration. Memory statistics were collected at runtime for each C-based implementation of Dijkstra's algorithm using the respective heap.

5.6 Cache Behavior and Prefetching

To evaluate how different heap implementations interact with the memory hierarchy, we used `valgrind` program's `cachegrind` tool. The `cachegrind` tool simulates a machine's cache and branch prediction behavior, enabling detailed analysis of instruction and data cache usage. It reports metrics such as instruction references, data reads and writes, and cache misses at multiple levels, including L1 and last-level cache (LL).

The test program was executed with the `--tool=cachegrind` option, which generated a detailed profile of memory behavior during heap operations. The output included counts of L1 instruction cache (I1), L1 data cache (D1), and LL cache references and misses.

The selection of metrics such as D1 (L1 data cache) misses, LL (last-level cache) misses, and instruction references (I1) was motivated by their direct correlation with memory access efficiency and overall execution performance. L1 data cache misses help identify how well each heap structure utilizes fast, frequently accessed memory during operations like insertions, deletions, and key updates. Last-level cache (LL) misses are especially critical because they typically indicate access to main memory (RAM), which incurs significantly higher latency. A high LL miss rate can severely degrade performance, particularly for pointer-heavy data structures like Fibonacci and Hollow heaps. Instruction cache misses (I1) and instruction reference counts were also included to account for the execution footprint of each implementation.

As a means to assess scalability, the experiments were conducted on both small and large graphs. The smaller dataset consisted of 35,000 nodes and 105,000 edges, while the larger graph included 1,000,000 nodes and 2,000,000 edges. This allowed us to observe how cache behavior evolves with input size and to detect any implementation-specific scaling bottlenecks.

Once baseline cache behavior was established, we introduced manual software prefetching in the heap implementations and evaluated its effectiveness using the same `cachegrind` metrics. The goal was to determine whether prefetching could reduce D1 and LL misses by

improving spatial and temporal locality during key operations.

In order to further validate the impact of prefetching on actual hardware, we employed operf, a performance profiling tool based on hardware performance counters. This allowed us to compare `cachegrind` tool's simulated results with the actual behavior and verify whether the observed reductions in cache misses translated into measurable improvements in L1 data cache performance.

5.7 Prefetching in Dijkstra's Algorithm

We measured the execution time of Dijkstra's algorithm with and without prefetching applied to the edge iteration loop. Heap implementations were left unchanged to ensure that only the impact of prefetching edge access was evaluated. Each configuration was executed three times, and the mean runtime was recorded.

The benchmark graph contained 35,000 nodes and 105,000 edges. Prefetching targeted the next edge in the loop using an `i+1` offset to improve temporal locality. This ensured the data would be available in cache when accessed in the next iteration.

The motivation behind this experiment was to investigate whether software prefetching outside the heap operations could offer measurable performance improvements. The goal was to assess the potential of applying prefetching in the Arc application, where graph data may be accessed before calling heap operations. By evaluating this behavior in the context of Dijkstra's algorithm, we aim to determine whether this form of optimization warrants further exploration as a potential performance improvement strategy.

Chapter 6

Results

In this chapter we present results comparing the heaps in execution time, language evaluation, as a priority queue in Dijkstra's algorithm , their memory usage and cache behavior.

6.1 Execution Time

In this section, we present results comparing the execution time of heap operations across the three implementations in C of the Binary Heap, Fibonacci Heap, and Hollow Heap.

The experiments were conducted by executing each operation repeatedly and recording the average time per operation. For *extract_min* and *decrease_key* tests, an initial phase of double insertions was performed to populate the heap. The time spent on these insertions and any subsequent deallocations (applicable to Fibonacci and Hollow Heaps) were excluded from the measurements.

Table 6.1 presents the execution times for various heap operations implemented in C, measured in nanoseconds.

Table 6.1: Execution time tests for operations in C listed in 5.2. Values in nanoseconds (ns)

Test	Binary Heap	Fibonacci Heap	Hollow Heap
1.1	**96.67**	166.67	120
1.2	**68.68**	127	93
1.3	**73.19**	79.99	80.86
1.4	58.59	**54.38**	62.91
1.5	**47.19**	50.21	48.25
1.6	**546.67**	833.34	560
1.7	635.67	824.33	**624.67**
1.8	1089.39	**915.98**	1254.97
1.9	1826.94	**1132.11**	1463.56
1.10	1756.17	**1192.62**	2123.88
1.11	126	**23**	86.7
1.12	63.67	**6.33**	56.33
1.13	52.07	**5.32**	41.61
1.14	31.38	**4.62**	32.84
1.15	30.84	**4.4**	32.2

As shown in Table 6.1, the Binary Heap demonstrates the lowest execution time for the majority of tests covering insertions (Tests 1.1 to 1.5)This is expected due to its simple array-based structure and minimal overhead.

To better understand the consistency of these results, Table 6.2 reports the corresponding standard deviations for each test. These values highlight how predictable each heap's performance is across repeated runs of the same operation.

Table 6.2: Standard deviations in execution time tests for operations in C listed in 5.2. Values in nanoseconds (ns)

Test	Binary Heap	Fibonacci Heap	Hollow Heap
1.1	38.586	26.247	35.590
1.2	21.746	23.338	26.882
1.3	20.908	23.334	16.056
1.4	9.696	8.149	3.017
1.5	1.144	2.315	0.220
1.6	151.070	266.124	240.559
1.7	183.256	216.268	252.468
1.8	338.562	33.282	443.745
1.9	399.390	135.172	305.720
1.10	52.356	41.433	714.587
1.11	33.993	12.472	51.854
1.12	18.373	2.625	16.131
1.13	6.678	1.308	6.064
1.14	1.245	0.103	0.556
1.15	0.518	0.021	0.143

For *insert* operations (tests 1.1–1.5), variability decreases as the number of operations increases. The Hollow Heap shows the most stable performance at scale, with particularly low deviation in test 1.5.

In *extract_min* operations (tests 1.6–1.10), standard deviations are higher across all heaps due to the complexity of the operation. The Fibonacci Heap shows relatively stable results in some cases such as test 1.8, while the Hollow Heap exhibits large fluctuations, especially in test 1.10.

For *decrease_key* operations (tests 1.11–1.15), the Fibonacci Heap stands out for its consistently low standard deviations, indicating stable performance even with increasing workload sizes. Binary and Hollow Heaps exhibit more variability, particularly in smaller test cases.

The total execution times for each test case have been visualized in Plot 6.1, 6.2 and 6.3 respectively. These values represent the cumulative execution time for the respective number of operations performed in each test, providing a clearer picture of how each heap scales with increased workload.

Figure 6.1 shows that the *insert* execution time generally favors the Binary Heap across increasing input sizes with the exception of 500,000 insertions (Test 1.4) where the Fibonacci Heap performs better.

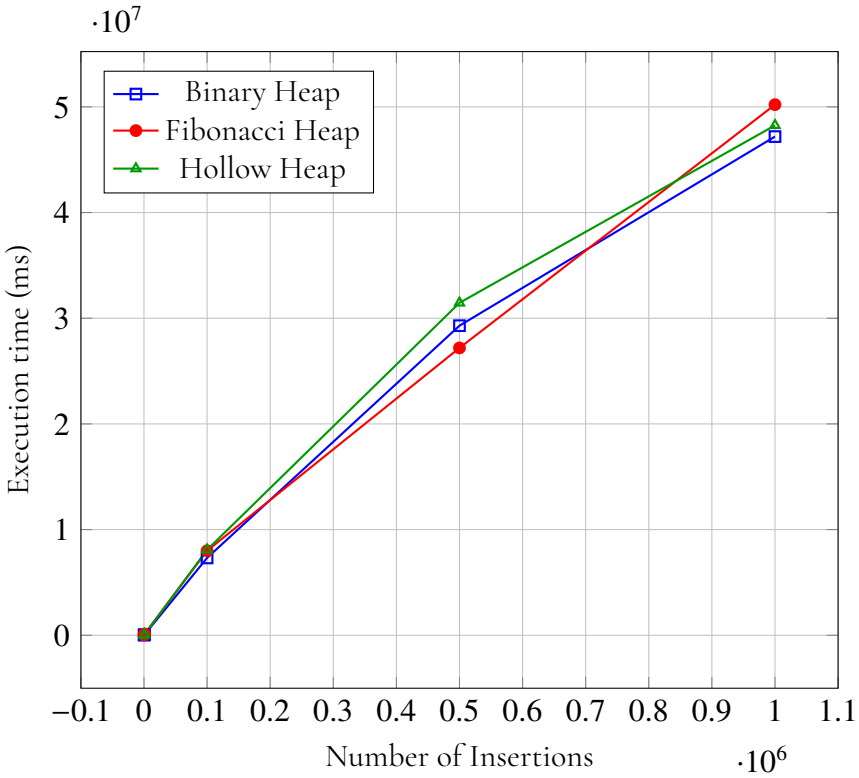


Figure 6.1: Insert Operation in C: Total Execution Time (ms)

In the *extract_min* tests (Tests 1.6 to 1.10), the Fibonacci Heap achieved the best performance in three out of five cases. The Binary Heap was the fastest in the smallest test 1.6, which involved 100 operations. The Hollow Heap performed best in test 1.7 with 1,000 operations. The Fibonacci Heap demonstrated the most favorable results for the *extract_min* operation.

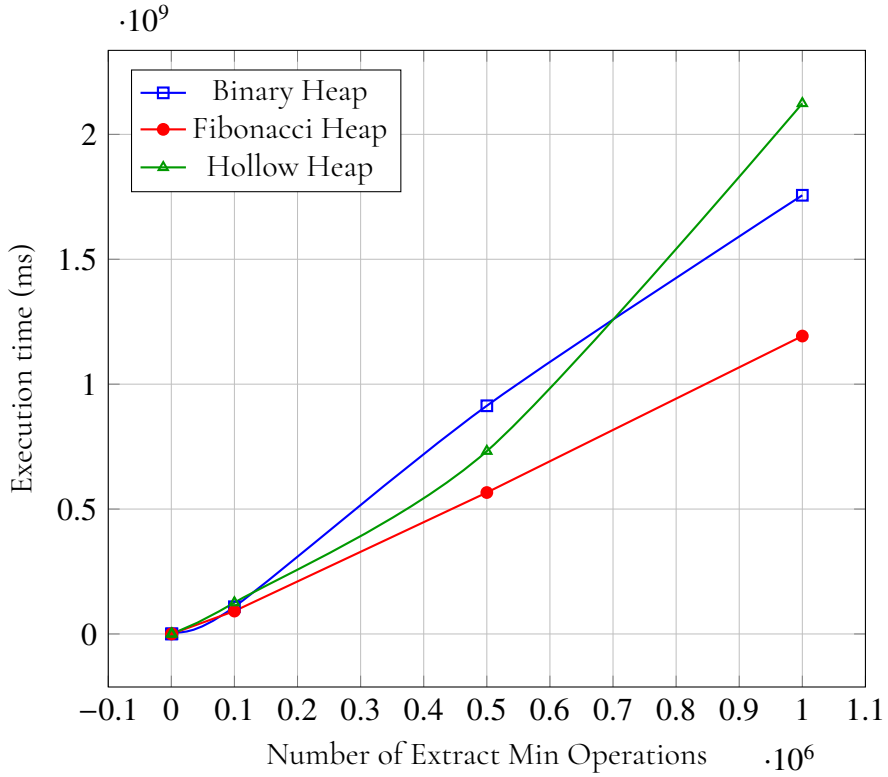


Figure 6.2: Extract Min Operation in C: Total Execution Time (ms)

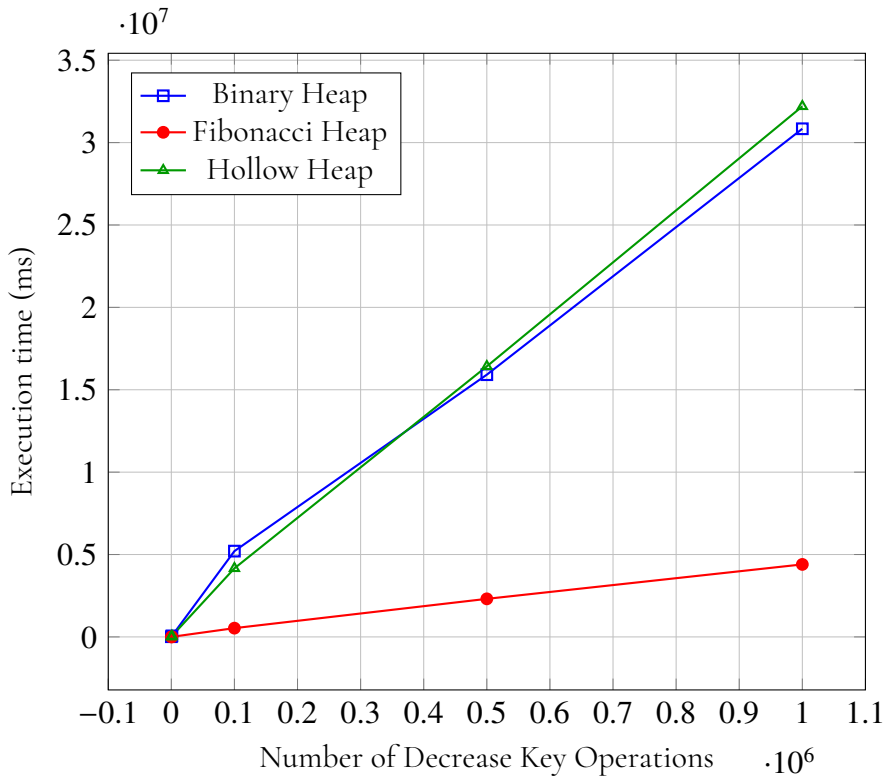


Figure 6.3: Decrease Key Operation in C: Total Execution Time (ms)

The *decrease_key* operations (Tests 1.8 to 1.11) show an advantage for the Fibonacci Heap, especially in larger tests, such as 1.15, where it significantly outperforms the other two heaps. This is in line with the theoretical *O(1)* amortized time for *decrease_key* in Fibonacci Heap. Hollow Heap, while also designed for efficient *decrease_key*, lags behind in practice, possibly due to additional pointer management overhead and more complex internal state transitions.

Overall, Fibonacci Heap shows the most scalable performance across different operation types, while Binary Heap is more appropriate for the *insert* operation. Hollow Heap shows slowed execution time in comparison to the other heaps which makes the practical values less promising than the theoretical amortized time complexity.

6.2 Language Evaluation

To evaluate the impact of language-specific memory management and runtime behavior, we measured the execution time per operation for various Fibonacci Heap operations implemented in C, Rust, and Java. The results of these tests are presented in Table 6.3.

Each test described in the methodology 5.3 was executed 10 times, and the mean execution time was calculated. To enable comparison across tests with different operation counts, the mean was divided by the number of operations in each test. This yields the average execution time per operation, reported in nanoseconds. Standard deviation is also included to indicate variability and aid in interpreting the reliability of the mean values.

The table reflects three heap operations: *insert*, *extract_min*, and *decrease_key*, across different input sizes, from as few as 500 operations to up to 1,000,000. This variation captures both startup overhead and large-scale performance behavior. Importantly, all tests used the same static input sequence for all languages. No randomized numbers were used, mirroring the deterministic behavior of the Arc application, which serves as a representative real-world dataset and motivated the inclusion of the 35,000-element test.

Table 6.3: Execution time per operation for tests on the Fibonacci Heap in C, Rust and Java

Test	C	Rust	Java
2.1	**32**	40,516	567
2.2	**28.3**	610	28.6
2.3	58.33	71	**18.94**
2.4	27	55.6	**9.51**
2.5	**278.6**	752.33	5403
2.6	**325.33**	1492.33	640.7
2.7	**338**	2239	354.33
2.8	**272**	2153.6	378.67
2.9	12	**3**	267.56
2.10	5.94	**4**	37.85
2.11	9.3	**4.66**	9.56
2.12	8.2	**4.7**	7.01

For the insert operation, C shows the fastest execution times in the smaller tests (2.1 and 2.2), while Java performs best with larger input sizes. In the *extract_min* operation, C

Table 6.4: Standard Deviation for tests on the Fibonacci Heap in C, Rust and Java

Test	C	Rust	Java
2.1	6.92	1,863.21	13.67
2.2	4.84	26.449	0.43
2.3	9.33	1.41	0.29
2.4	2.6	6.59	0.32
2.5	11.81	5.19	197.43
2.6	38.98	111.25	4.36
2.7	8.4	20.74	11.16
2.8	0.77	18.86	4.57
2.9	21.83	0.47	3.56
2.10	0.72	0	2.24
2.11	0.61	0.47	0.08
2.12	0.66	0.48	0.09

consistently outperforms both Java and Rust across all tests. For the *decrease_key* operation in tests 2.11 and 2.12, all languages demonstrate low execution times, with Rust performing best, followed closely by C, and Java showing the highest average execution time per operation.

The standard deviation (SD) for each test is presented in Table 6.4. It reflects the variance of the values used to calculate the mean execution time per operation. The SD provides insight into how the measured values deviate from the mean. The largest standard deviation is observed in test 2.1 for Rust, indicating significant variation in execution time for this particular case. Conversely, the lowest standard deviation is also recorded by Rust in test 2.10, where all measured values are identical, resulting in zero variation.

C exhibits relatively stable standard deviation across most tests, suggesting consistent performance and similar results over repeated runs. An exception is test 2.9, where the standard deviation is notably high relative to the mean.

Java also demonstrates stable performance in terms of standard deviation across most tests, with the exception of test 2.5, where a higher variation is observed. Although, considering the mean value of 5403, a deviation of 197 remains relatively small.

6.3 Dijkstra's Algorithm

In this section, we focus on the behavior of Binary, Fibonacci and Hollow Heaps in C when used as the priority queue in Dijkstra's algorithm, applied to both a dense graph (with 20% edge density) and sparse graphs (with approximately 0.006% edge density). The results below compare execution times across increasing graph sizes.

6.3.1 Dense graph

The Table 6.5 and its corresponding plot 6.4 quantify how Dijkstra's algorithm scales with increasing graph size. Figure 6.4 visualizes the execution times reported in Table 6.5, illustrating the relative performance of the three heap implementations. For small graphs, the

Binary Heap exhibits slightly lower execution times. However, as the input size grows, the Fibonacci Heap demonstrates superior scalability due to its asymptotically faster *decrease_key* operation. Despite its favorable theoretical bounds, the Hollow Heap performs comparably to the Fibonacci Heap in larger graphs but consistently lags slightly behind, likely due to increased pointer indirection and memory overhead.

Table 6.5: Measured execution times (in milliseconds) of Dijkstra's algorithm using Binary, Fibonacci, and Hollow Heaps as priority queues on dense graphs (20% edge density), implemented in C.

Nodes	Edges	Binary Heap (ms)	Fibonacci Heap (ms)	Hollow Heap (ms)
1,000	99,900	1.116	1.164	1.675
5,000	2,499,500	18.806	17.250	20.381
10,000	9,999,000	72.351	61.724	67.909
15,000	22,497,500	167.530	139.631	142.621
20,000	39,998,000	302.893	245.158	253.730
25,000	62,497,500	451.565	362.880	382.993

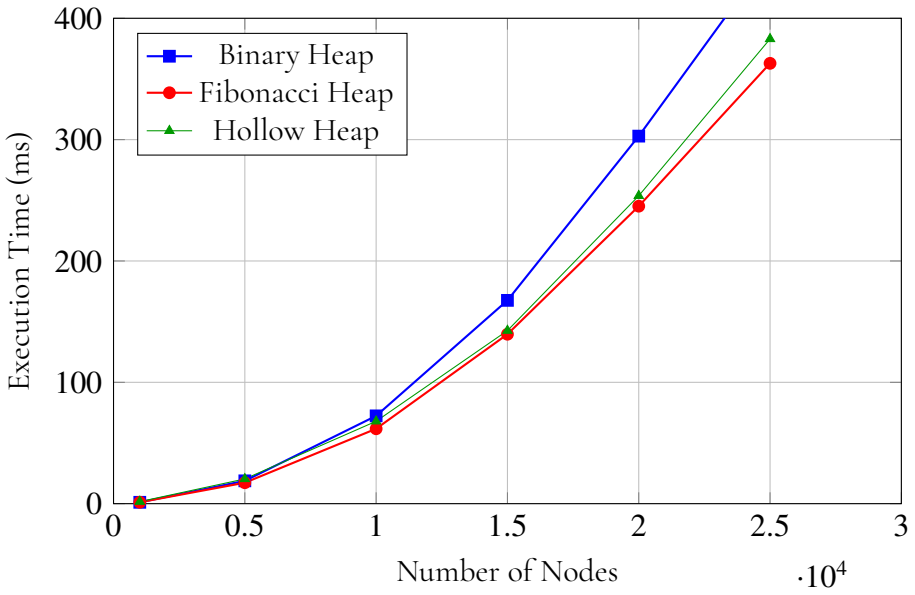


Figure 6.4: Execution time comparison of Binary, Fibonacci, and Hollow Heaps used as priority queues in Dijkstra's algorithm on dense graphs (20% edge density), implemented in C.

6.3.2 Sparse graph

The Table 6.6 presents execution times for Dijkstra's algorithm across varying graph sizes, increasing both nodes and edges. As seen in Table 6.6, the Binary Heap consistently achieves the lowest execution times throughout all tested input sizes.

In contrast, the Fibonacci Heap performs significantly slower, and the performance gap widens as the graph grows. Despite its theoretical benefit for *decrease_key* operations, the

added pointer complexity and overhead result in higher execution times in practice. The Hollow Heap exhibits the slowest performance overall, with execution times increasing steeply with graph size. This trend can likely be attributed to its heavy reliance on dynamic memory allocation and more complex node management.

Table 6.6: Execution time comparison of Binary, Fibonacci, and Hollow Heaps used as priority queues in Dijkstra's algorithm on sparse graphs (approximately 0.006% edge density), implemented in C.

Edges	Nodes	Binary Heap (ms)	Fibonacci Heap (ms)	Hollow Heap (ms)
3,000	1,000	0.254	0.562	1.286
15,000	5,000	1.442	4.553	7.013
30,000	10,000	3.686	8.116	10.070
45,000	15,000	5.482	11.437	14.338
60,000	20,000	8.781	15.017	20.072
75,000	25,000	11.087	18.683	24.790
90,000	30,000	14.044	24.528	30.914
105,000	35,000	16.358	28.945	36.178
120,000	40,000	19.185	35.765	43.652
135,000	45,000	23.554	44.119	48.022
150,000	50,000	25.401	45.953	54.278
165,000	55,000	27.525	50.442	61.307
180,000	60,000	30.986	60.299	64.151
195,000	65,000	31.643	57.009	71.443
210,000	70,000	37.372	64.657	77.945

Figure 6.5 visualizes the execution times listed in Table 6.6 for increasingly large but sparse graphs. The plot illustrates how the Binary Heap outperforms both the Fibonacci and Hollow Heaps across all input sizes.

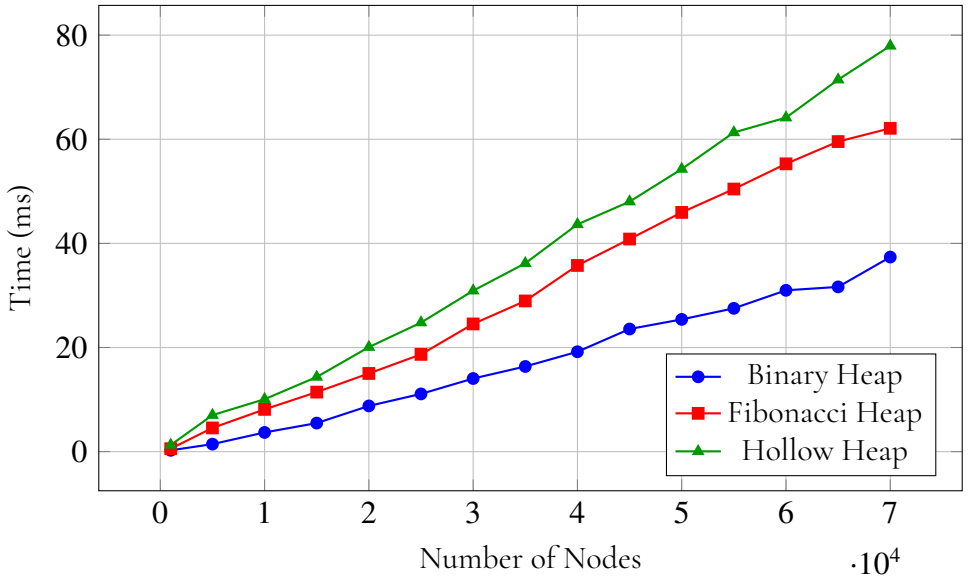


Figure 6.5: Execution time comparison of Binary, Fibonacci, and Hollow Heaps used as priority queues in Dijkstra's algorithm on sparse graphs (0.006% edge density), implemented in C.

6.4 Memory Usage

The memory consumption of each heap implementation in C was measured using the `valgrind` program's `massif` tool and runtime diagnostics. For a graph with 35,000 nodes and 105,000 edges, the following memory usage statistics were recorded:

Table 6.7: Memory allocation and usage for each heap type in C, measured on a graph with 35,000 nodes and 105,000 edges.

Heap Type	Allocations	Frees	Bytes Allocated
Binary Heap	175,118	175,118	29,939,776
Fibonacci Heap	490,118	490,118	66,279,056
Hollow Heap	551,477	551,477	256,255,568

These results highlight the significantly higher memory demands of the Fibonacci and Hollow heaps compared to the Binary heap. The Hollow heap, in particular, uses almost four times more memory than the Fibonacci Heap and nearly ten times more than the Binary heap. This increase is largely due to its use of multiple auxiliary node structures for managing hollow and full nodes. The memory efficiency of the Binary heap reflects its simpler structure and lower pointer overhead.

6.5 Cache Behavior and Prefetching

To assess the impact of memory access patterns and prefetching, we measured cache behavior during the execution of Dijkstra's algorithm using each of the three heap types. The tests were conducted on two sparse graph, a small graph with 35,000 nodes and 105,000 edges, and a large graph with 1,000,000 nodes and 2,000,000 edges.

Table 6.8 shows the instruction cache (I1), data cache (D1), and last-level cache (LL) misses for each heap type on the small graph. The same test were performed with prefetching enabled, but the impact on the results were minimal and therefore we chose to exclude them from the Table 6.8.

Table 6.8: Cache behavior without prefetching for Binary, Fibonacci and Hollow Heap on a small sparse graph, implemented in C.

Heap Type	I1 Misses	D1 Misses	LL Misses	D1 Miss Rate	LL Miss Rate
Binary Heap	1,671	4,448,893	742,230	2.7%	0.2%
Fibonacci Heap	1,762	4,710,266	802,629	2.0%	0.1%
Hollow Heap	1,701	4,434,970	820,635	2.2%	0.1%

For the small graph, the Binary Heap records the highest L1 data cache (D1) miss rate at 2.7%, while Fibonacci and Hollow Heaps have slightly lower D1 miss rates of 2.0% and 2.2% respectively. However, Binary has the fewest last-level (LL) cache misses at 742,230, compared to 802,629 for Fibonacci and 820,635 for Hollow. This is because the total data accesses were lower for Binary, with around 165 million loads compared to over 235 million for Fibonacci. Despite the higher miss rate, the Binary Heap benefits from more compact memory layout and better spatial locality, resulting in fewer costly LL cache accesses. The pointer-based structures of Fibonacci and Hollow Heaps lead to less predictable memory access and increased pressure on LL cache.

We now turn to the results for the larger sparse graph, shown in Table 6.9.

Table 6.9: Cache behavior without prefetching for Binary, Fibonacci and Hollow Heap on a large sparse graph, implemented in C.

Heap Type	I1 Misses	D1 Misses	LL Misses	D1 Miss Rate	LL Miss Rate
Binary Heap	1,678	123,305,520	90,408,731	3%	0.8%
Fibonacci Heap	1,768	114,789,542	93,193,364	1.7%	0.5%
Hollow Heap	1,704	113,035,629	90,051,815	2.2%	0.6%

For the large graph, D1 miss rates rise slightly across all heaps. Binary Heap shows a D1 miss rate of 3%, while Fibonacci and Hollow show 1.7% and 2.2% respectively. The number of LL cache misses remains high for all three, but Binary again maintains a slight edge with 90.4 million LL misses versus 93.2 million for Fibonacci and 90.0 million for Hollow. This suggests that while D1 misses are more frequent in Binary, a larger proportion are successfully handled by L2, reducing main memory accesses.

In case the negligible impact of the prefetching was due to `cachegrind` being able to identify the prefetched data, further analysis of prefetching was done using operf. The analysis revealed a reduction in L1 data cache misses for certain key operations. For instance, in the

extract_min function of the Fibonacci Heap, the number of sampled L1 load misses decreased from 3589 to 2404 after manual prefetch instructions were added. This result indicates that the inserted prefetches were effective and placed at performance-critical points in the code. The corresponding source lines aligned with pointer traversal operations, supporting the hypothesis that these memory accesses benefit from early loading into cache.

These findings suggest that, although `cachegrind` did not fully reflect the impact of prefetching, hardware-level sampling confirmed its positive effect. The improvement is especially relevant in pointer-heavy structures where irregular memory access contribute to cache inefficiency but may still benefit from well-targeted prefetching.

Next, Table 6.10 shows the average execution times and standard deviations for each heap on the same small graph.

Table 6.10: Average execution time and standard deviation (in milliseconds) for three runs of Binary, Fibonacci and Hollow Heap, with and without prefetching on a small sparse graph, implemented in C.

Heap Type	Prefetching	Avg Time (ms)	Std Dev (ms)
Binary Heap	No	15.633	0.665
Binary Heap	Yes	14.970	0.546
Fibonacci Heap	No	30.906	3.644
Fibonacci Heap	Yes	30.689	4.914
Hollow Heap	No	27.349	2.836
Hollow Heap	Yes	37.867	5.740

On the small graph, Binary Heap runs in 15.6 ms without prefetching and 15.0 ms with prefetching. The difference is small but consistent. Fibonacci Heap shows no meaningful change in runtime with prefetching, while Hollow Heap performance degrades significantly, from 27.3 ms to 37.9 ms, likely due to cache pollution or prefetches evicting useful data.

Table 6.11 presents the corresponding execution times and coresponding standard deviations for three runs for the large sparse graph.

Table 6.11: Average execution time and standard deviation (in milliseconds) for three runs of Binary, Fibonacci and Hollow Heap, with and without prefetching on large sparse graphs, implemented in C.

Heap Type	Prefetching	Avg Time (ms)	Std Dev (ms)
Binary Heap	No	932.953	2.768
Binary Heap	Yes	950.094	3.389
Fibonacci Heap	No	1656.727	20.416
Fibonacci Heap	Yes	1711.305	86.012
Hollow Heap	No	1700.462	65.943
Hollow Heap	Yes	2054.398	51.016

On the large graph, Binary Heap again performs best, completing in 933 ms without prefetching and 950 ms with prefetching. Fibonacci and Hollow Heaps are significantly slower overall and both show worse performance when prefetching is enabled. Fibonacci

increases from 1657 ms to 1711 ms and Hollow increases from 1700 ms to 2054 ms. Standard deviations also increase, suggesting greater runtime instability under prefetching.

Two primary factors may explain why the prefetching did not improve the execution times. First, manually inserted prefetches may evict useful data from cache, especially in L1 or L2, leading to increased miss penalties. Second, the cost of executing additional prefetch instructions may outweigh the benefit if the prefetched data is not accessed soon enough or frequently enough to justify it. These results suggest that prefetching must be applied selectively, with careful consideration of both temporal locality and cache pressure introduced by extra instructions.

Since the manual prefetches were inserted with knowledge of when and where data would be accessed, replicating this logic is challenging for a compiler. Preliminary tests using compiler optimization flags to enable automatic prefetching showed no measurable benefit and were therefore not pursued further. The presence of pointer-based data structures and non-trivial control flow, particularly in the Fibonacci and Hollow Heaps, limits the compiler's ability to statically analyze access patterns and determine effective prefetch targets and timing. A more comprehensive evaluation of compiler-inserted prefetching remains an area for future investigation.

Across both graph sizes, the Binary Heap exhibits the lowest average execution time and smallest standard deviation, indicating more consistent performance compared to the other heap types. We evaluated the impact of prefetching inserted in both the compiler and manually inserted, but did not observe a consistent performance improvement from the manual approach and in some cases, it led to increased run time.

6.6 Prefetching in Dijkstra's Algorithm

To evaluate the impact of software prefetching within Dijkstra's algorithm itself (outside of the heap implementation), a smaller, separate test was conducted where execution times were measured for all three heap variants both with and without manual prefetching. Prefetching was applied in the loop that iterates over neighboring edges, as described in Section 5.7.

This test focuses solely on prefetching within the Dijkstra loop, i.e. no prefetching was applied within the heap operations themselves. The goal is to isolate the effects of prefetching on graph traversal and assess whether improving data locality before heap interactions can influence overall performance.

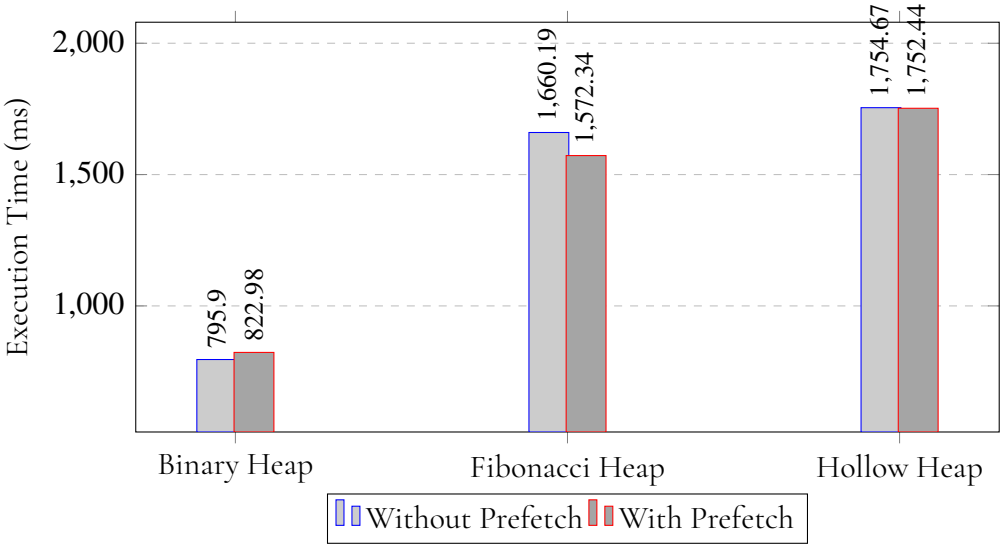


Figure 6.6: Execution time (ms) with and without prefetching in Dijkstra's Algorithm

As shown in the Figure 6.6, prefetching led to a modest performance improvement in the Fibonacci Heap variant, while the Hollow Heaps showed no clear benefit and the Binary Heap showed even a slight slowdown. The corresponding numerical values are presented in Table 6.12.

Table 6.12: Execution time (ms) with and without prefetching in Dijkstra's Algorithm

Heap Type	Without Prefetch (ms)	With Prefetch (ms)
Binary Heap	795.90	822.98
Fibonacci Heap	1,660.19	1,572.34
Hollow Heap	1,754.67	1,752.44

6.7 operf

We use operf to profile the performance of the different heap implementations when running Dijkstra's algorithm on small and large input graphs. The purpose is to identify where execution time is spent, and how the data structure design influences cycle distribution among core functions. Figures 6.7 to 6.12 show the ten most cycle-intensive functions for each configuration.

In the case of Binary Heap (Figures 6.7 and 6.8), execution time is relatively well distributed among several key operations. For small inputs, no single function dominates, functions such as `make_connected` and `insert_linear` appear prominently in the cycle breakdown, as they are used to generate the graph for the experiment. In the current test setup, the graphs were generated for each run to allow flexibility in the number of nodes and edges. However, to eliminate the overhead introduced by graph generation, future work could instead use a static dataset.

```
Counted CYCLES events (Cycles) with a unit mask of 0x00 (No unit mask)
count 100000

Samples  %         image name           Symbol name
752     16.2244     a.out                make_connected
429      9.2557     a.out                down_heapify
413      8.9205     lib-c-2.31.so        random
311      6.7098     a.out                swap
257      5.5448     a.out                insert_linear
253      5.4585     a.out                make_rand_graph
243      5.2427     a.out                dijkstra
201      4.3366     a.out                xedge_hash
166      3.5814     a.out                reallocate
142      3.0636     a.out                find_pair
126      2.7184     a.out                up_heapify
125      2.6996     a.out                compare_xedge
116      2.5027     a.out                too_many
105      2.2654     a.out                first
91       1.9633     lib-c-2.31.so        _int_malloc
78       1.6828     a.out                top
61       1.3161     a.out                min
52       1.1219     kallsyns             prep_new_page
51       1.1003     a.out                pop
49       1.0572     a.out                free_all
```

Figure 6.7: Binary Heap (Small input)

```
Counted CYCLES events (Cycles) with a unit mask of 0x00 (No unit mask)
count 100000

Samples   %        image name         Symbol name
39772    20.2355    a.out              make_connected
35245    17.9322    a.out              down_heapify
18118     9.2182    a.out              dijkstra
16326     8.3065    a.out              make_rand_graph
9828      5.0004    a.out              swap
9607      4.8879    a.out              xedge_hash
8395      4.2713    a.out              up_heapify
8331      4.2387    libc-2.31.so       random
6813      3.4664    a.out              find_pair
3752      1.9090    a.out              insert_linear
3636      1.8499    a.out              compare_xedge
3095      1.5747    a.out              free_all
2875      1.4628    a.out              reallocate
2692      1.3697    libc-2.31.so       _int_malloc
2583      1.3142    a.out              pop
2172      1.1051    libc-2.31.so       malloc_consolidate
2167      1.1025    a.out              too_many
2144      1.0908    a.out              decrease_key
2083      1.0598    a.out              top
```

Figure 6.8: Binary Heap (Large input)

For the Fibonacci Heap (Figures 6.9 and 6.10), a different pattern is observed. In both input sizes, a large proportion of the cycles is spent in the consolidate function. This is expected, as Fibonacci Heaps require frequent restructuring after deletions. The functions extract_min, link, and decrease_key also contribute significantly. These results illustrate how the theoretical amortized benefits of Fibonacci Heaps can be offset by the overhead of dynamic memory operations and pointer-heavy access patterns.

```
Counted CYCLES events (Cycles) with a unit mask of 0x00 (No unit mask)
count 100000

Samples   %        image name         Symbol name
4047    35.3481    a.out              consolidate
901      7.8697    a.out              dijkstra
746      6.5159    a.out              extract_min
731      6.3848    a.out              link
710      6.2014    a.out              make_connected
549      4.7952    a.out              insert_rootlist
460      4.0178    libc-2.31.so       random
264      2.3059    a.out              make_rand_graph
246      2.1487    a.out              insert_linear
224      1.9565    libc-2.31.so       _int_malloc
197      1.7207    a.out              xedge_hash
191      1.6683    a.out              decrease_key
163      1.4237    a.out              reallocate
159      1.3888    libc-2.31.so       _int_free
140      1.2228    a.out              find_pair
126      1.1005    a.out              compare_xedge
```

Figure 6.9: Fibonacci Heap (Small input)

```
Counted CYCLES events (Cycles) with a unit mask of 0x00 (No unit mask)
count 100000

Samples   %        image name         Symbol name
155320   26.5849    a.out              consolidate
78833    13.4932    a.out              dijkstra
63203    10.8179    a.out              extract_min
52370     8.9638    a.out              make_connected
51042     8.7364    a.out              link
25504     4.3653    a.out              make_rand_graph
22466     3.8453    a.out              decrease_key
17934     3.0696    a.out              insert_rootlist
12751     2.1825    libc-2.31.so       random
11173     1.9124    a.out              find_pair
10014     1.7140    a.out              xedge_hash
8124      1.3905    libc-2.31.so       _int_malloc
7043      1.2055    libc-2.31.so       unlink_chunk.isra.0
6750      1.1553    libc-2.31.so       malloc_consolidate
```

Figure 6.10: Fibonacci Heap (Large input)

The Hollow Heap (Figures 6.11 and 6.12) shows a dominance of the delete function, especially for the large input, where it accounts for over half of the total cycles. This suggests that the Hollow heap's structure introduces considerable runtime overhead, particularly during cleanup and child relocation. Other frequently occurring functions include add_child and malloc_consolidate, which further reflect the cost of managing multiple levels of indirection and dynamically allocated nodes.

```
Counted CYCLES events (Cycles) with a unit mask of 0x00 (No unit mask)
count 100000

Samples   %        image name          Symbol name
10573    59.2060    a.out               delete
1022      5.7229    a.out               dijkstra
775       4.3398    a.out               link
764       4.2782    a.out               make_connected
571       3.1974    a.out               add_child
492       2.7551    libc-2.31.so        malloc_consolidate
435       2.4359    libc-2.31.so        random
333       1.8647    libc-2.31.so        _int_malloc
279       1.5623    a.out               insert_linear
275       1.5399    a.out               make_rand_graph
212       1.1871    a.out               xedge_hash
197       1.1031    libc-2.31.so        _int_free
```

Figure 6.11: Hollow Heap (Small input)

```
Counted CYCLES events (Cycles) with a unit mask of 0x00 (No unit mask)
count 100000

Samples   %        image name          Symbol name
380901   52.0592    a.out               delete
95155    13.0052    a.out               dijkstra
51636     7.0573    a.out               make_connected
29628     4.0494    a.out               link
24874     3.3996    a.out               make_rand_graph
20130     2.7512    a.out               add_child
15691     2.1445    libc-2.31.so        malloc_consolidate
12558     1.7163    libc-2.31.so        random
11045     1.5096    a.out               find_pair
10229     1.3980    a.out               xedge_hash
 9823     1.3425    libc-2.31.so        _int_malloc
```

Figure 6.12: Hollow Heap (Large input)

Together, the operf results confirm the earlier observation that while pointer-based heaps like Fibonacci and Hollow Heaps offer better theoretical amortized complexity, they suffer in practice due to costly memory management routines. A significant share of execution time is spent in functions such as *delete* and *consolidate*, but also in system-level memory allocation functions like `malloc`, `_int_malloc`, and `malloc_consolidate`. This overhead stems from frequent dynamic allocations and deallocations, which also degrade cache behavior due to fragmented and unpredictable memory access patterns. In contrast, the Binary Heap exhibits a more balanced distribution of cycles across core operations and avoids most dynamic memory costs, resulting in better alignment with hardware characteristics and more predictable performance.

Chapter 7
Discussion

In this chapter, we discuss the results obtained from evaluating the heap implementations and their performance across various dimensions. The focus is on interpreting the trade-offs observed in execution time, memory usage and cache behavior. We also examine the impact of the language choice, not only in terms of performance regarding execution time but also from a maintainability and integration standpoint relevant to a business-critical application.

The heaps were implemented and tested with the aim of understanding how theoretical advantages translate into practical performance. As mentioned in the background chapter in 2.2 on amortized time, even if amortized constant time operations may appear beneficial in theory there can be real world constraints such as memory access patterns and pointer overhead.

We begin by analyzing execution time, where we observed that the Binary Heap consistently delivered the fastest performance for insertion operations, owing to its simple array-based structure and low overhead. The Fibonacci Heap outperformed the others in both *extract_min* and *decrease_key* operations, aligning well with its theoretical advantages. Hollow heap, though theoretically efficient, underperformed in practice, likely due to its complex structure and excessive dynamic memory use.

The section on memory usage discusses how the Binary heap outperformed the other implementations in terms of allocation efficiency. Its pre-allocated array minimized dynamic memory operations. Conversely, the Fibonacci and Hollow heaps made frequent allocations for each node and supporting arrays, especially during consolidate or delete operations, significantly increasing both memory usage and runtime overhead.

We then delve into cache behavior, which proved to be a critical factor in performance. The Binary Heap benefited from predictable access patterns and spatial locality due to its array-based structure, leading to relatively low miss rates, particularly in the instruction and last-level caches. Although the Fibonacci and Hollow Heaps rely more heavily on pointer-based indirection, they did not consistently exhibit worse cache statistics; in fact, they sometimes achieved lower last-level cache miss rates.

We also examined the role of prefetching and whether it offers measurable improvements.

Our results showed that prefetching had negligible impact on cache miss rates and often introduced performance overhead, especially on larger graphs. In some cases, such as with the Hollow Heap, prefetching substantially increased execution time, suggesting that poorly aligned prefetch instructions can interfere with natural cache behavior rather than enhance it.

Lastly, the programming language evaluation compares the performance and maintainability of implementations in C, Java, and Rust. C provided the fastest execution overall due to low-level memory control and minimal abstraction. Java offered strong performance on larger datasets thanks to its JIT optimizations and managed memory, while Rust struck a balance between performance but introduced implementation complexity due to its strict ownership model.

Each section builds on the quantitative findings to provide a more nuanced understanding of how heap choice and implementation strategy affect performance in practical scenarios.

7.1 Execution Time

When total execution times were measured, each heap exhibited different performance characteristics across its operations. Depending on the chosen operation the different heaps presented various values which are further discussed in this section.

The Binary heap presented the fastest times for insertions. It exhibited stable results between the tests involving *insert* operations. The average execution time per *insert* operation remains relatively stable regardless of the number of operations, showing consistent performance across both small and large test cases. This aligns with its theoretical constant-time complexity, as presented in Section 2.2. The insert operation ranged from 47.19 to 96.67 ns, making the Binary Heap the most efficient among the three, with consistently lower execution times. For the *extract_min* operation, the Binary Heap showed greater variation in execution time per operation as the number of operations increased. The execution time per operation ranged from 546.67 to 1756.17 ns, where smaller data sets yielded faster results.

For the Fibonacci Heap, insert times ranged from 166.67 ns in the smallest test to 50.21 ns in the largest. Despite its theoretical constant amortized time, the Fibonacci Heap showed higher overhead in smaller tests, with performance improving as the number of operations increased. As the input size increased, the performance improved, possibly due to reduced relative impact of fixed overhead. When evaluating *extract_min*, the Fibonacci Heap the results ranged from 833.34 to 1192.62 ns, depending on the number of operations in the test cases. The results remained consistent for larger tests, indicating reliable performance. The Fibonacci Heap performed exceptionally well in *decrease_key* operations, making it the best choice in systems where such operations dominate. The Fibonacci Heap's faster execution times in *decrease_key* operations can be attributed to its lazy reconstruction strategy.

The Hollow Heap is, in theory, the most efficient structure presented in this study due to its constant amortized time complexity for all operations. However, in practice, its execution times were slower than those of the other heaps in most tests. This highlights the importance of evaluating practical performance rather than relying solely on theoretical complexity.

The Hollow Heap's poor performance may result from a lack of rebuilding of the structure. Hollow nodes remain in the structure, affecting the overall structure of the DAG. This

accumulation increases complexity, resulting in higher overhead and longer execution times when reconstructions are needed. A dense structure with many nodes and pointers can significantly degrade performance.

Analyzing the internal structure reveals important differences. The Fibonacci and Hollow Heaps are pointer-based structures that use linked lists or DAGs, which adds complexity when modifying nodes. In contrast, the Binary Heap is array-based and uses index calculations to manage parent-child relationships, offering a more efficient and predictable memory layout. This structure benefits from better cache locality and lower memory overhead, which can significantly reduce execution time, this is further discussed in Section 7.5.

From a theoretical perspective, insertion operations in Fibonacci and Hollow Heaps should be more efficient than in Binary Heaps. In a Fibonacci Heap, inserting a node is done simply by adding a singleton tree at the end of the root list making a sequence of insertions easy to perform. In the Hollow Heap and its DAG structure there are the slight difference that an added node is immediately compared to the root. If the new node should be the root, the old structure is added as children to the new root. This approach simplifies the Hollow Heap's structure by allowing insertions to extend the tree downward without requiring immediate reorganization. For the Binary Heap, the node is added at the end of the array which is done in constant time. If any heap property is violated, the node is moved up in the tree with worst case being the new root resulting in logarithmical time complexity.

Although the Binary Heap has a theoretically slower time complexity for *insert* and *decrease_key* operations with *O(log n)* versus the *O(1)* amortized time of Fibonacci and Hollow Heaps, it outperformed both in practice. This discrepancy suggests that the more complex pointer-based heaps may encounter frequent worst-case behavior due to overhead from dynamic memory allocation and poor cache utilization. In contrast, the Binary Heap's array-based design delivers faster execution times.

For further evaluation, the Fibonacci Heap was selected for implementation in Java and Rust to facilitate cross-language performance comparisons. This choice was motivated in part by its relatively shorter execution times for the *decrease_key* and *extract_min* operations.

7.2 Language Evaluation

Across all operations tested, C consistently exhibited the best performance, especially in scenarios involving frequent *extract_min* and *insert* operations. It also performed well on *decrease_key*, although Rust was slightly faster in some instances.

C's performance advantages may stem from its fine-grained memory management. This makes it possible to have full control over the used memory which contributes to efficient systems. The data structure used in these tests, the Fibonacci Heap, is itself pointer-based, making it particularly well-suited for implementation in C. Because C is a pointer-oriented language, the environment naturally complements the requirements of such structures. Additionally, the original pseudocode used for all heap implementations was written in a C-like syntax, which may have contributed to better alignment with the language's features and performance model.

If the application relies heavily on *decrease_key* operations, Rust presents a viable alternative. However, since *decrease_key* is typically accompanied by a large number of *insert* operations, Rust may be most appropriate in workloads that involve high operation volumes, such

as those in Test 2.3 and 2.4.

Java performed best only for *insert* operations in the large-scale tests (2.3 and 2.4), possibly due to internal optimizations by the Just-In-Time (JIT) compiler. Outside of this scenario, Java generally exhibited higher execution times, making it the least suitable of the three languages in the context of this thesis.

The language evaluation results show that no single language is universally superior. The most suitable choice depends on the characteristics of the target application, particularly the mix and frequency of heap operations.

Beyond raw performance, other factors also influence language selection from a company perspective. These include time to market, development cost, and the learning curve associated with the language. Such practical concerns can have a significant impact on the efficiency and quality of the software development process.

Java is widely used in both industry and academia. Its readable syntax and managed memory model reduce the barrier to entry, making it ideal for teams with mixed experience levels. Moreover, its mature ecosystem, extensive libraries, and robust tooling contribute to faster development cycles and easier maintenance.

Rust is a powerful but complex language, characterized by strict semantic and syntactic rules. Its ownership and borrowing model ensures memory safety without a garbage collector which offers strong performance and safety guarantees. This technical capacities and stability of Rust makes it possible to have low level control [12].

C, often regarded as the foundational language of modern systems programming, provides developers with a high degree of control and flexibility. With a large user base and well-established standard libraries, C can be relatively accessible for experienced developers. However, its manual memory management and reliance on pointers can lead to subtle bugs and increased maintenance effort. Still, for those seeking performance and control, C remains a strong choice.

In the context of Panasonic Avionics Arc, developed by Tactel, the use of multiple programming languages reflects the varied requirements across system components. The core engine is written in a C++ environment, making C the most natural and efficient option given the team's existing expertise. Furthermore, the benchmarks from this study show that C consistently delivered the best performance across heap implementations, particularly in insert-heavy and *extract_min* intensive scenarios.

Given that the Arc application's queue is likely to perform many insertions initially, followed by a combination of numerous *extract_min* and *decrease_key* operations, the findings of this evaluation provide a strong foundation for recommending C as the most efficient language for future development of priority queues within the Arc environment.

7.3 Dijkstra's Algorithm

Dijkstra's algorithm was tested using all three heap implementations to assess how the combination of operations impacts execution time when computing shortest paths. For sparse graphs, the Binary Heap showed the best performance, consistently achieving lower execution times compared to both the Fibonacci Heap and the Hollow Heap. In this test, the nodes in the heaps were initialized with infinite distances. These nodes were then updated using *decrease_key* operations to reflect actual shortest path distances.

Based on prior results in 6.1, the Fibonacci Heap was expected to perform best. From a theoretical standpoint, the Hollow Heap should also be a strong contender due to its constant amortized time for *decrease_key* operations. Nevertheless, the actual results contradicted these expectations. The Binary Heap consistently outperformed the other heap implementations across all tests covering sparse graphs, suggesting that the theoretical advantages of alternative heaps do not necessarily translate into practical performance gains. Possible reasons for this could be the memory management which is further discussed in 7.4 but also other factors like a lot of pointer overhead and dynamic allocation cost in both Fibonacci and Hollow Heap.

One contributing factor appears to be the expensive *consolidate* step within the Fibonacci Heap's *extract_min* operation. As observed in the profiling results presented in Section **??**, this operation introduces significant overhead, particularly in sparse graphs where fewer *decrease_key* operations are performed relative to *extract_min*. In such cases, the cost of consolidation is not sufficiently amortized over multiple lightweight operations, leading to degraded performance.

For denser graphs, the Binary Heap performed slightly slower than the other two alternatives. The reasons for this result could be that the number of neighbors increases the use of *decrease_key*. When looking at the *decrease_key* operation, the Fibonacci Heap presented the fastest values making it appropriate for this heap to present the fastest execution times for large dense graphs.

7.4 Memory Usage

When analyzing memory management, the amounts of allocations, frees and bytes allocated are presented for each heap structure. While all heaps showed equal counts of allocations and deallocations, this alone does not guarantee the absence of memory leaks. However, in this case, the consistent match between allocs and frees suggests that memory was correctly deallocated and that no leaks were detected in the tested scenarios.

In the comparison of allocations the Binary heap has noticeable lower numbers in comparison to the Fibonacci Heap as can be seen in Table 6.8. The Hollow heap has the largest number of allocations with 3 times the amount of the Binary heap. This supports the conclusion that the Binary Heap has best memory management during Dijkstra's algorithm for graphs with 35,000 nodes and 105,000 edges.

One reason for the Binary Heap's efficient memory usage is its use of a contiguous array of double pointers, which is allocated once at heap creation with a capacity matching the expected maximum heap size. Since this array does not require frequent reallocation during execution, memory remains stable and predictable across all heap operations, isolating memory management to a single preallocated block.

In contrast, the Fibonacci and Hollow Heaps rely on pointer-based structures, which inherently requires allocation and deallocation at runtime. This leads to a higher total number of allocations and increased overall memory usage.

Each node in the Fibonacci Heap has in total four pointers marking siblings, children and parent of the node. The sibling pointers, right and left, are set to the node itself to create a circular doubly linked list, the children and parent pointers are set to `NULL` until these are found by the structure. When a node is full, four pointers are in use. In consolidate there

are two double pointer arrays keeping track of the roots and ranks of the heap. These arrays are allocated, and deallocated in the consolidate function making it time consuming for the overall run of Dijkstra's algorithm 6.10. Reusing these arrays could reduce runtime, but they must be sized for worst-case scenarios, potentially making them excessively large.

The hollow nodes holds four pointers to other nodes as well as a pointer to an item. This item is necessary to be able to make the node hollow during decrease key. The node pointers makes it possible to create linked lists in the DAG structure. This memory needs to be allocated for every node in the structure making the hollow node the most memory demanding node in this study. This memory is allocated during runtime on the dynamic memory. In the delete function there are a double pointer array handling the rank of the subtrees to be able to rebuild during extract min operations. This array is allocated and freed during each run of delete, like consolidate in the Fibonacci Heap, which affects the total execution time 6.12. Like in the Fibonacci Heap, reuse of this array could potentially decrease the runtime resulting in better performance both in time and memory.

Furthermore, in the Hollow Heap, each *decrease_key* operation creates a new node while marking the original one as hollow. These hollow nodes remain in the structure until the heap is rebuilt, which only happens during specific operations like *extract_min*. This results in a growing number of allocated nodes over time, contributing further to the total number of bytes allocated and the overall memory consumption. When and how often the heap should be rebuilt to efficiently manage hollow nodes is left for future research.

7.5 Cache Behavior and Prefetching

In Table 6.8, we observed that the Binary Heap had the fewest I1 and LL cache misses on the small sparse graph. Its D1 miss rate was 2.7% and the LL miss rate 0.2%, which is considered low and aligns with findings that array-based structures tend to exhibit regular memory access patterns and lower cache miss rates due to spatial locality [5]. Although the Binary Heap had slightly higher D1 misses than the Hollow Heap, this can likely be attributed to the frequent jumps in its contiguous array representation.

Fibonacci and Hollow Heaps exhibited slightly higher LL cache misses than Binary Heap, despite their lower D1 miss rates (2.0% and 2.2% respectively). These results suggest that although their pointer-heavy structures reduce the number of unique memory locations accessed per operation (lowering D1 pressure), they result in more scattered memory accesses that lead to elevated LL cache pressure.

When scaling up to the large sparse graph (Table 6.9), we observed similar trends. The Binary Heap again showed the highest cache miss counts, with a D1 miss rate of 3%. Both Fibonacci and Hollow Heaps saw D1 miss rates around 1.7 to 2.2%. The LL misses increases for the Binary Heap presenting a value larger than for the Hollow Heap. One reason for the increased LL cache misses for the Binary Heap could be that the extended node array becomes large enough to exceed the size of the LL cache leading to higher LL misses.

Table 6.11 shows the runtime impact of including prefetching in the data structures. The Binary Heap was the fastest, taking around 933 ms without prefetching and slightly more with it. Fibonacci Heap showed a modest increase in runtime with prefetching. In contrast, the Hollow Heap saw a larger increase in runtime, from 1700 ms to over 2050 ms. This reinforces the interpretation that the prefetching overhead can dominate if the targeted memory

accesses are not well-aligned with actual use patterns.

We have evaluated the effect of compiler-inserted prefetching by comparing it to manually inserted prefetch instructions added directly into the heap implementations. This comparison was based on the execution times presented in Table 6.10 and Table 6.11, where manually inserted prefetching showed no clear performance benefit and in some cases increased runtime. These results suggest that prefetching is only effective when memory access patterns are predictable and prefetched data is reused shortly after being fetched.

In the evaluated heap structures, such predictability is difficult to achieve. Due to the extensive use of pointer indirection and complex control flow, especially in the Fibonacci and Hollow Heaps, it is unlikely that a compiler can reliably identify effective prefetch targets or determine the correct timing for their insertion. Unlike a programmer who has knowledge of the heap's operational semantics and traversal patterns, the compiler lacks context about the structure's behavior during execution.

These findings indicate that, at least in this study, manually inserted prefetching functions are more likely to yield meaningful results than relying on compiler-inserted prefetching. However, this conclusion is based on a limited test setup. In future work, compiler-inserted prefetching could be explored further by evaluating different optimization levels and compiler flags that influence prefetch behavior, to more rigorously assess its potential in this context.

Overall, we found that prefetching offered only marginal performance gains on small graphs and consistently increasing runtime on large graphs. This suggests that the potential benefits from prefetching are outweighed by the added overhead and disruption it causes to natural cache behavior in these heap implementations.

7.6 Prefetching in Dijkstra's Algorithm

The results from the smaller test indicate differing effects of manual prefetching within Dijkstra's main loop depending on the heap implementation. For the Binary Heap, prefetching led to a slight performance degradation of approximately 1–2%, suggesting that the additional instructions introduced overhead without improving cache behavior. In contrast, the Fibonacci Heap saw a notable performance improvement of around 5-6%, indicating that prefetching helped mitigate the costs of its more complex memory access patterns. The Hollow Heap also benefitted, though to a lesser extent, with a modest improvement of about 0.5-1%.

No additional experiments were conducted on software prefetching within Dijkstra's algorithm beyond the initial exploratory tests. This decision was based on the scope of the project, which focused on evaluating heap data structures rather than optimizing graph traversal algorithms. Dijkstra's algorithm was used primarily as a representative workload to stress and compare heap implementations under realistic usage. As such, further investigation into prefetching at the algorithmic level was not pursued. Nonetheless, the performance improvement observed in the Fibonacci Heap indicates that there may be potential for future work in this area.

7.7 Sources of Error

There are several potential sources of error that may have influenced the results of this study. One possibility is that certain edge cases were not encountered during testing, which could have exposed incorrect behavior or implementation flaws in one or more of the heap structures. To minimize this risk, we performed rigorous functional testing across a wide range of input and operation sequences. These tests included stress tests with extreme values, mixed-operation sequences and preformed random sample inspections to check correct behavior of the heaps.

Another potential source of error lies in the underlying hardware used for benchmarking. All tests were executed on a single machine to ensure consistency, but results may vary on different computers, particularly in aspects such as cache size and CPU architecture. These hardware dependent variables could have affected execution time and memory access patterns, especially in pointer-heavy data structures as Fibonacci and Hollow Heap. In cases where the performance differences between heaps were marginal, it is possible that a alternative hardware configurations could have produced different relative results.

Compiler optimizations and background system activity during test execution may also have influenced the results. Although we made efforts to isolate the benchmarking environment and control for external factors, such as background processes, complete isolation is difficult to guarantee without dedicated hardware. As a result of this, minor fluctuations in execution time should therefore be interpreted with caution, especially when comparing results within narrow performance margin.

Time measurement precision is another limiting factor. Despite that high resolution timers were used in the testing, small scale variations in timing can still occur, in particular for the very fast operations or short running test cases. Repeated measurements and averaging were used to mitigate this issue but some inherent variability may remain.

Lastly, potential errors may exist in the Fibonacci Heap implementations in Java and Rust due to the absence of thorough stress testing. While smaller functionality tests were performed to ensure correctness, these implementations were not subjected to the same level of rigorous, large-scale testing as the C version. As a result, the reliability of the Java and Rust results may be lower. It is possible that, under larger inputs or extended runtimes, bugs or performance issues could emerge that were not detected in the limited testing performed.

Taken these factors under consideration, the overall trends of the study are likely robust but individual data points with minor differences should be interpreted with precaution and with these limitations in mind.

7.8 Future Work

While this thesis provides a practical evaluation of Hollow Heaps in a specific application context, there remains considerable scope for future research. Our study contributes by being one of few practical assessments of Hollow Heaps, which have primarily been discussed from its theoretical advantages previously. Although, the limited and domain-specific nature of our test environment, particularly regarding the constrained size of the graphs, means that the results cannot be generalized to broader or more demanding scenarios without additional testing.

To build on this work, a natural next step would be to evaluate the performance of Hollow Heaps in substantially larger dense graphs, where their asymptotic advantages might become more apparent. Furthermore, integrating Hollow Heaps into other graph algorithms such as Prim's algorithm for minimum spanning trees or A* for pathfinding could help assess their versatility and performance across a broader range of operations, beyond the ones emphasized in Dijkstra's algorithm.

While this thesis focuses primarily on execution time, further studies could explore other dimensions of performance in more detail. This includes delving deeper into hardware-level behaviors such as prefetching effectiveness and branch prediction accuracy. In particular, evaluating the impact of manual versus compiler-inserted prefetching would be valuable, especially approaches where prefetching is triggered only upon second-level cache misses. Techniques such as those proposed in [26] use trap handlers to activate a compiler-controlled prefetch engine, which operates independently once started and introduces minimal instruction overhead.

An expanded test suite covering all heaps implemented in all three languages would also allow for a more comprehensive evaluation of language suitability for various problem domains. Further tests focusing on amortized time behaviors could help clarify how theoretical advantages translate into practical performance under varying workloads.

Additionally, memory management strategies could be optimized, particularly in C where allocation and deallocation introduce potential overhead. Exploring alternative approaches such as memory reuse could yield improved implementations with better runtime efficiency.

A specific area of interest for future investigation is the handling of hollow nodes in the Hollow Heap. During *decrease_key* operations, new nodes are created while the old ones are marked as hollow, which can lead to significant memory overhead if not managed carefully. The heap must eventually be rebuilt to discard these hollow nodes and maintain performance, but determining the optimal timing or conditions for triggering such a rebuild remains an open question. Future work should investigate strategies for efficiently managing hollow node accumulation, potentially improving both time and space performance.

Finally, future work could include evaluating heap performance using a detailed simulator of the CPU pipeline and memory hierarchy. Ongoing work on modeling the IBM S822L with POWER8 processors in the asim simulator can possibly be used to learn more about prefetching in the evaluated heaps [25].

Chapter 8
Conclusion

In this master thesis we have answered the following four research questions.

- **RQ1: Which heap produces the fastest execution time in Dijkstra's algorithm and why? Which is most appropriate for Arc to use?**

  The results demonstrate performance differences among the heap structures, influenced by the graph characteristics and the frequency of specific operations. The Binary Heap achieved fastest execution times for the *insert* operation overall. While the Fibonacci Heap performed best in both *extract_min* and *decrease_key* operations when evaluated in isolation, these were not fully realized in sparse graph scenarios.

  In particular, the *extract_min* operation incurred significant overhead due to the complex consolidation process, as shown in the profiling results in Section 6.7. Consequently, when used as the priority queue in Dijkstra's algorithm, the Fibonacci Heap did not outperform the Binary Heap on sparse graphs. For denser graphs, however, the Fibonacci Heap exhibited superior execution times, making it more suitable in cases where *decrease_key* operations are more frequent.

  Thus, these results suggest that the heap selection in Dijkstra's algorithm should be guided by the graph's sparsity and the relative frequency of heap operations. The Binary Heap is most suitable for sparse graphs, where *insert* and *extract_min* dominate, while the Fibonacci Heap may be preferable for dense graphs, where the *decrease_key* operation is performed more frequently. Given that *decrease_key* operations are the most frequent in the Arc application, the Fibonacci Heap emerges as the most appropriate data structure.

- **RQ2: To what extent does the choice of programming language affect heap performance? Can performance be generalized across languages, or are they implementation-specific?**

  The choice of programming language had a significant impact on heap performance in the conducted tests. C consistently delivered the fastest execution times in most

scenarios, particularly for the *extract_min* operation, establishing it as the most performant option for the Arc application.

Rust demonstrated strong performance in the *decrease_key* operation across all input sizes, achieving the lowest execution times in this category. However, its performance in insertion-heavy scenarios was less competitive, especially in smaller tests, where it incurred significant overhead.

Java, while generally slower overall, showed competitive performance in large-scale insertion tests, outperforming both C and Rust in those cases. Despite this, Java remained the slowest language in most other operations, particularly in *extract_min* and *decrease_key*, making it the least suitable option for implementing the Fibonacci Heap in the Arc context.

Altogether, no single language emerged as optimal for all use cases. C demonstrated the most consistent and robust performance across the tested operations, followed by Rust, with Java lagging behind in most categories.

However, generalizing these results to other applications should be done with caution. Performance outcomes are highly sensitive not only to the implementation details of the data structures, and operation frequency, but also to broader factors such as compiler optimizations, runtime environments, standard libraries, and just-in-time (JIT) compilation strategies. As such, performance is heavily influenced by the surrounding ecosystem and implementation choices. While one language may offer advantages in one context, a different combination of language, compiler, and runtime may be more suitable in another. Therefore, language choice should be informed by a combination of performance requirements, tooling maturity, and developer environment constraints.

- **RQ3: How does the number of memory accesses differ between the different data structures and how efficient are cache memories for the Arc project?**

  The results indicate that the Binary Heap makes the most efficient use of memory, followed by the Fibonacci Heap and then the Hollow Heap. This likely stems from the structural differences between the data structures. The Binary Heap relies on an array-based structure, which minimizes pointer usage and thereby reducing memory fragmentation. In contrast, both the Fibonacci and Hollow Heaps employ more complex node structures involving multiple pointers. This leads to additional memory allocations during runtime and greater memory fragmentation which increases pressure on the cache memory.

  Cache behavior was found to influence runtime performance outcomes. While the Binary Heap occasionally exhibited higher L1 data cache miss rates, it maintained lower last-level (LL) cache miss counts, especially under larger workloads. This suggests that its memory access patterns are overall more cache-friendly. The frequent but predictable accesses allowed higher cache-levels, such as L2, to handle misses effectively, thereby reducing expensive accesses to main memory.

  Surprisingly, the Fibonacci and Hollow Heaps did not consistently exhibit worse cache performance. In certain benchmarks, they recorded lower last-level cache miss rates

compared to the Binary Heap. This implies that while their memory usage is more irregular, it may also result in better cache dispersion or reduced conflict on certain cache lines.

For the Arc project, the Binary Heap appears to be the most suitable option from a cache-efficiency perspective. Although individual cache metrics occasionally favored the Fibonacci or Hollow Heaps, these differences were not substantial enough to offset the Binary Heap's advantage, as it resulted in the fewest total cache misses in the majority of cases.

- **RQ4: Can prefetching improve performance either at the heap level, within Dijkstra's algorithm, or through compiler-level optimizations?**

  The execution times for smaller sparse graphs (Table 6.10) showed a modest improvement in the Binary Heap's runtime when prefetching was enabled. In contrast, the Fibonacci Heap exhibited negligible performance change, while the Hollow Heap experienced a noticeable slowdown. This suggests that prefetching may interfere with the Hollow Heap's memory access patterns rather than optimize them.

  In tests with larger sparse graphs (Table 6.11), prefetching consistently resulted in slower execution times across all heap implementations. This degradation is likely due to the increased overhead from speculative memory accesses that are not well aligned with actual usage in large pointer-heavy structures. The added prefetch instructions may have fetched unnecessary or untimely data, which could be replacing relevant data.

  While cache-level analysis using `cachegrind` showed minimal impact from prefetching, deeper sampling with hardware-level profiling with *operf* revealed reductions in L1 data cache misses for specific operations. For instance, in the Fibonacci Heap's *extract_min* function, where the number of sampled L1 data cache load misses decreased from 3589 to 2404 at a critical source code line. This demonstrates that selectively placed prefetching functions can be beneficial in isolated, pointer-intensive operations. Still, the improvements were not reflected in overall runtime. This is likely because the gains from reduced cache misses were offset by the cost of additional prefetch instructions and the risk of cache pollution when data is fetched unnecessarily or prematurely.

  Compiler-inserted prefetching was briefly considered but not explored further, as it was unlikely to outperform the manually placed prefetch instructions already tested. Despite being inserted with detailed knowledge of each data structure's memory behavior, the selected prefetch locations did yield consistent performance improvements and, in some cases, even degraded runtime. Given this, compiler-generated prefetching, lacking contextual awareness, was expected to perform even worse, particularly for pointer-heavy structures like Fibonacci and Hollow Heaps. Limited tests confirmed no measurable benefit, and thus, compiler prefetching was excluded from further evaluation. Future work could revisit this topic by systematically experimenting with different optimization levels and compiler flags to more rigorously evaluate the potential of compiler-based prefetching

  Prefetching within Dijkstra's main loop was evaluated in a smaller-scale test. The results showed a minor improvement for the Fibonacci Heap, a slight regression for the

Binary Heap, and a small gain for the Hollow Heap. However, as the primary focus of this thesis was on heap behavior rather than graph algorithm optimization, this line of investigation was not extended.

Overall, the findings indicate that prefetching can offer performance improvements in specific contexts, particularly for the Binary Heap on smaller workloads or in targeted pointer traversals in the Fibonacci Heap. Nevertheless, it is not a universally effective optimization strategy and may even degrade performance when applied indiscriminately, especially on large workloads.

References

[1] Visualization of heap operations, 2024. Accessed: 2025-05-28. URL: https://gallery.selfboot.cn/en/algorithms/heap.

[2] Michael Barbehenn. A Note on the Complexity of Dijkstra's Algorithm for Graphs with Weighted Vertices. *IEEE Transactions on Computers*, 47(6):749–751, 1998. doi:10.1109/12.689660.

[3] Guy E. Blelloch and Jonathan Harper. *Algorithms: Parallel and Sequential*. Carnegie Mellon University, 2022.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, us, 3 edition, 7 2009.

[5] Ulrich Drepper. What Every Programmer Should Know About Memory. *Red Hat, Inc*, 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[6] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.

[7] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. In *25th Annual Symposium on Foundations of Computer Science, 1984.*, pages 338–346, 1984. doi:10.1109/SFCS.1984.715934.

[8] Thomas D. Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow heaps. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015), Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 689–700. Springer-Verlag, 2015. doi:10.1007/978-3-662-47672-7_90.

[9] Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow Heaps. *ACM Trans. Algorithms*, 13(3), jul 2017. doi:10.1145/3093240.

[10] Jason Huang Hu and Wei Wang. Algorithm for Fibonacci Heap Operations, 2003. Accessed: 2025-05-28. URL: https://www.eecs.yorku.ca/~aaw/legacy/Jason/FibonacciHeapAlgorithm.html.

[11] Jim Jeffers and James Reinders. Chapter 5 - Lots of Data (Vectors). In Jim Jeffers and James Reinders, editors, *Intel Xeon Phi Coprocessor High Performance Programming*, pages 107–164. Morgan Kaufmann, Boston, 2013. doi:10.1016/B978-0-12-410414-3.00005-0.

[12] Steve Klabnik and Carol Nichols. Understanding ownership, 2025. Accessed: 2025-04-08. URL: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[13] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. In *Proceedings of the Meeting on Algorithm Engineering & Experimiments*, page 61–72, USA, 2014. Society for Industrial and Applied Mathematics.

[14] Rhyd Lewis. A Comparison of Dijkstra's Algorithm Using Fibonacci Heaps, Binary Heaps, and Self-Balancing Binary Trees, 2023. arXiv:2303.10034.

[15] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. *Association for Computing Machinery*, pages 222–233, 9 1996. doi:10.1145/237090.237190.

[16] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. A survey of shortest-path algorithms, 2017. arXiv:1705.02044.

[17] Desiree Martinez, Axl Remegio, and Darllaine Lincopinis. A Review on Java Programming Language, 05 2023.

[18] Juan-Julian Merelo-Guervós, Israel Blancas-Álvarez, Pedro A. Castillo, Gustavo Romero, Victor M. Rivas, Mario García-Valdez, Amaury Hernández-Águila, and Mario Román. A comparison of implementations of basic evolutionary algorithm operations in different languages. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 1602–1609, 2016. doi:10.1109/CEC.2016.7743980.

[19] SC Peta. Programming Language–Still Ruling the World. *Global Journal of Computer Science and Technology*, 22(1):9–13, 2022.

[20] Dennis M Ritchie. The development of the C language. *ACM Sigplan Notices*, 28(3):201–208, 1993.

[21] C.L. Sabharwal. Java, java, java. *IEEE Potentials*, 17(3):33–37, 1998. doi:10.1109/45.714612.

[22] Robert Seacord. Effective c. *Computer*, 53:79–82, 11 2020. doi:10.1109/MC.2020.3016369.

[23] Robert W Sebesta, Soumen Mukherjee, and Arup Kumar Bhattacharjee. *Concepts of programming languages*, volume 7. Addison-Wesley Reading, Massachusetts, 1999.

[24] Jonas Skeppstedt. *Algorithms: A Concise Introduction*. Independently published, 2023.

[25] Jonas Skeppstedt. The ASIM Power multiprocessor simulator v. 1.9.1 . Technical report, 2025.

[26] Jonas Skeppstedt and Michel Dubois. Compiler Controlled Prefetching for Multiprocessors Using Low-Overhead Traps and Prefetch Engines. *J. Parallel Distrib. Comput.*, 60(5):585–615, 2000.

[27] Jonas Skeppstedt and Christian Söderberg. Writing efficient C code : a thorough introduction. *CreateSpace*, 1 2020.

[28] Tactel. Panasonic Avionics Arc, 2025. Accessed: 2025-05-13. URL: https://tactel.se/en/cases/exploring-the-world-below-from-the-sky-above/.

[29] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.

[30] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4), April 1978. doi:10.1145/359460.359478.

[31] J. W. J. Williams. Algorithm 232 : HEAPSORT. *Communications of the ACM*, 7:347–348, 1 1964.

[32] xvi-xv-xii-ix-xxii-ix xiv. Fibonacci heap in rust. Accessed: 2025-03-10. URL: https://github.com/xvi-xv-xii-ix-xxii-ix-xiv/fibonacci_heap.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2025-05-23

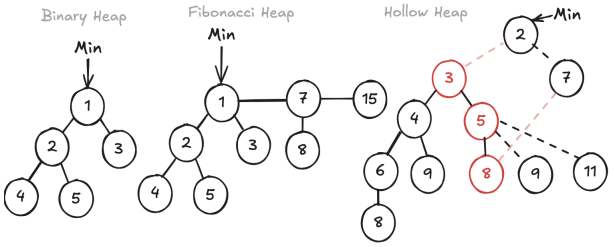
EXAMENSARBETE A Performance Study of Priority Queues: Fibonacci Heap, Binary Heap, Hollow Heap
STUDENTER Vera Paulsson, Klara Tjernström
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Flavius Gruian (LTH)

Heaps Don't Lie: En jämförelse av heapar för Panasonic Avionics Arc

POPULÄRVETENSKAPLIG SAMMANFATTNING **Vera Paulsson, Klara Tjernström**

En heap är en form av prioritetskö som används för att lagra och sortera data i olika typer av applikationer såsom vid schemaläggning och vägplanering. I detta examensarbete undersöks olika heapstrukturer med målet att utvärdera dess effektivitet i Panasonic Avionics Arc och Dijkstras algoritm.

Detta arbete presenterar resultat och jämförelsedata för tre olika heapar för att hitta det bästa alternativet för Panasonic Avionics Arc. I nuläget använder systemet en lista som sorteras vilket anses ineffektivt, och som förslag presenteras en blandning av heapar för att öka prestandan i programmet. Följande heapar undersöks: Binary Heap, Fibonacci Heap och Hollow Heap.



För att förenkla konceptet dras en jämföras med ett besök på akuten. När du anländer registrerar du dig och får en plats i kön, detta översätts till en *insert* i heapen. Om ditt tillstånd plötsligt försämras får du högre prioritet, motsvarande en *decrease_key*. När det är dags att bli undersökt av en läkare kallas den med högst prioritet in först, detta representerar *extract_min* operationen. Operationerna ovan testas enskilt i examensarbetet för att ge uppfattning om deras individuella prestation. De testas även i Dijkstras algoritm, som hittar kortaste vägen från en plats till alla andra platser. På så sätt kan man iaktta förändringar i isolerade operationer samt i en kombination av dem.

Genom att jämföra heapstrukturer kan vi föreslå en alternativ lösning till Arc, med syfte att eliminera problem som flimmer på kartan.

När de teoretiska bidragen inom området granskas kan man anta att Fibonacci Heap och Hollow Heap bör prestera bäst. Detta eftersom deras genomsnittliga tidskomplexitet för en sekvens av operationer, också kallad *amortized time*, förväntas operationer snabbare i förhållande till den för Binary Heap. Däremot finns det dolda tidskrävande arbeten i datorn som orsakas av strukturernas komplexitet och minneshantering vilket i sin tur påverkar deras prestanda i praktiken. Sålunda måste praktiska resultat utvärderas då den kan skilja sig från de teoretiska. Resultaten påvisade att Binary Heap var mest effektiv för *insert* och använde minst minne, medan Fibonacci Heap var snabbast för *decrease_key* och *extract_min*. Detta visar att teori och praktik inte alltid överensstämmer. I fallet för Arc är Fibonacci Heap den mest lämpliga lösningen då den har kortast körtid för den mest avgörande operationen *decrease_key*.