HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY OF INFORMATION TECHNOLOGY

PROJECT REPORT

A BIG DATA FILE SORTER USING EXTERNAL SORTING ALGORITHMS

SUBJECT: Data Structures and Algorithms (DSA)

CLASS CODE: DASA230179E

LECTURER: Huỳnh Xuân Phụng

MEMBER:

Huỳnh Minh Tài - 22110068

Văn Phạm Thảo Nhi – 23110049

Ho Chi Minh City, January, 2026

LIST OF MEMBERS

SEMESTER I ACADEMIC CALENDAR 2025-2026

Project: A Big Data File Sorter Using External Sorting Algorithms

Student ID	**Student name**	**Assignment**	**Contribution (%)**	**Grade**
22110068	Huỳnh Minh Tài			
23110049	Văn Phạm Thảo Nhi			

<u>Lecturer’s comment</u>

—--

Ho Chi Minh City, January…….2026

Lecturer’s signature

(Sign and print name)

Acknowledgement

The authors would like to express their sincere gratitude to **Mr. Huỳnh Xuân Phụng**, lecturer of the Data Structures and Algorithms (DSA) course, for his dedicated guidance and clear theoretical foundation throughout the semester. The knowledge provided in this course, particularly on sorting algorithms, heap data structures, complexity analysis, and algorithmic trade-offs, has played a crucial role in shaping the direction and implementation of this project.

We also appreciate the constructive feedback and academic requirements set by the course, which motivated us to not only focus on algorithm correctness but also consider practical constraints such as memory usage, I/O efficiency, and system-level performance.

Finally, we would like to thank our teammates for their collaboration and effort during the development, experimentation, and documentation phases of the project.

Preface

This report presents the “Big Data File Sorter” project, developed as part of the Data Structures and Algorithms (DSA) course under the instruction of **Mr. Huỳnh Xuân Phụng.** The project aims to bridge theoretical DSA concepts with practical system-level challenges by addressing the problem of sorting datasets that exceed available main memory.

Traditional in-memory sorting algorithms, although efficient for small datasets, become impractical when data size surpasses RAM capacity. To tackle this limitation, the project adopts the External Merge Sort approach, emphasizing controlled memory usage, efficient disk I/O, and clear separation between in-memory computation and disk-based processing.

Through this project, we apply and reinforce key DSA topics taught in the course, including heap structures, priority queues, comparison-based sorting, and asymptotic complexity analysis, while also gaining insight into real-world considerations such as I/O bottlenecks and performance trade-offs. The report documents the system design, implementation, analysis, and evaluation results, serving both as a course deliverable and a practical learning experience in large-scale data processing.

Contents

PART 1: PROJECT DESCRIPTIONS.

1.1. Problem Statement.

At a large scale, even fundamental operations such as file sorting present significant technical challenges when the input size exceeds the system's available RAM capacity. Conventional in-memory (internal) sorting algorithms, such as Quick Sort and Merge Sort, are typically designed under the assumption that the entire dataset can reside in main memory for efficient processing. However, applying these methods to oversized data often leads to critical failures, most notably memory allocation errors (e.g., std::bad_alloc). Furthermore, if the operating system attempts to manage the overflow via virtual memory or swap space, performance degrades exponentially. This occurs because disk access is orders of magnitude slower than RAM, causing the system to be bottlenecked by I/O latency rather than CPU cycles.

To address these limitations, our group implemented a solution centered on the External Merge Sort algorithm. This approach strategically partitions the dataset into smaller, manageable "chunks" that fit within the allocated memory limits. Each chunk is sorted internally to generate temporary "runs" on the disk, which are subsequently consolidated through a merging phase to produce the final sorted output. By maintaining a stable memory footprint of O(M)—where M represents the defined chunk size—this methodology effectively optimizes I/O operations through sequential data access. Consequently, it provides a robust and scalable framework for handling large-scale datasets stored on local storage media.

1.2. Objectives.

The Big Data File Sorter project aims to build a complete C++ system that can sort massive datasets exceeding physical memory limits by leveraging an external sorting approach. The project focuses on the following core objectives:

+ **Correctness:** The system guarantees that output.txt is a correctly sorted (ascending) version of input.txt. To preserve data integrity, it includes a built-in verification component to validate the correctness and consistency of the final output.

+ **Large-Scale Data Handling:** A key objective is to ensure stable execution even when the dataset is far larger than available RAM. This is achieved by enforcing strict memory constraints through configurable settings such as a memory budget and adjustable chunk sizes.

+ **Performance and I/O Optimization:** Since disk I/O is costly, the system emphasizes sequential access patterns and efficient buffering. The merge strategy is designed to reduce the number of disk passes, thereby lowering overall latency and improving end-to-end performance.

1.3. Scope and Objects.

1.3.1. Scope.

The project focuses on building a large-scale file sorting system using an external sorting approach, running on a single machine with data stored on local disk. The system takes input.txt as input, where each line contains an integer, and produces output_sorted.txt (or output.txt) as the output file sorted in ascending order.

The processing pipeline follows the External Merge Sort model with two main phases. In the first phase, the program reads the input data in chunks that fit within a specified memory limit, sorts each chunk in RAM, and writes the sorted results to temporary files called runs. In the second phase, the program merges these sorted run files to generate the final sorted output.

Within the scope of the current version, the system prioritizes correctness and the ability to handle datasets larger than RAM by strictly limiting how much data is loaded into memory at any time. It also organizes disk reading and writing in a largely sequential manner to reduce I/O overhead.

Features that are out of scope for this version include distributed sorting across multiple machines, full pipeline multi-threading optimizations, data compression, and handling complex multi-field records. These directions are considered future improvements and are only discussed in the “Future Work” section.

1.3.2. Objects.

The primary object of the project is an input data file stored on disk, containing a list of integers to be sorted. During execution, the system sequentially operates on data chunks loaded into memory, temporary run files that store the sorted results of each chunk, and finally the output file produced after the merge process.

In the run generation phase, each chunk is sorted in memory before being written back to disk. During the merge phase, the system maintains a heap (or priority queue) to continuously select the smallest element among the current candidates from all runs, thereby ensuring that the merge process produces the output in strictly ascending order. In addition, the program employs read/write buffering mechanisms to reduce the number of fine-grained disk I/O operations, which helps improve performance when processing large datasets.

Beyond the core sorting program, the project includes several supporting tools for testing and evaluation, such as a data generator for creating test datasets, a verification tool to check whether the output file is correctly sorted, and a benchmarking tool to measure execution time and overall performance.

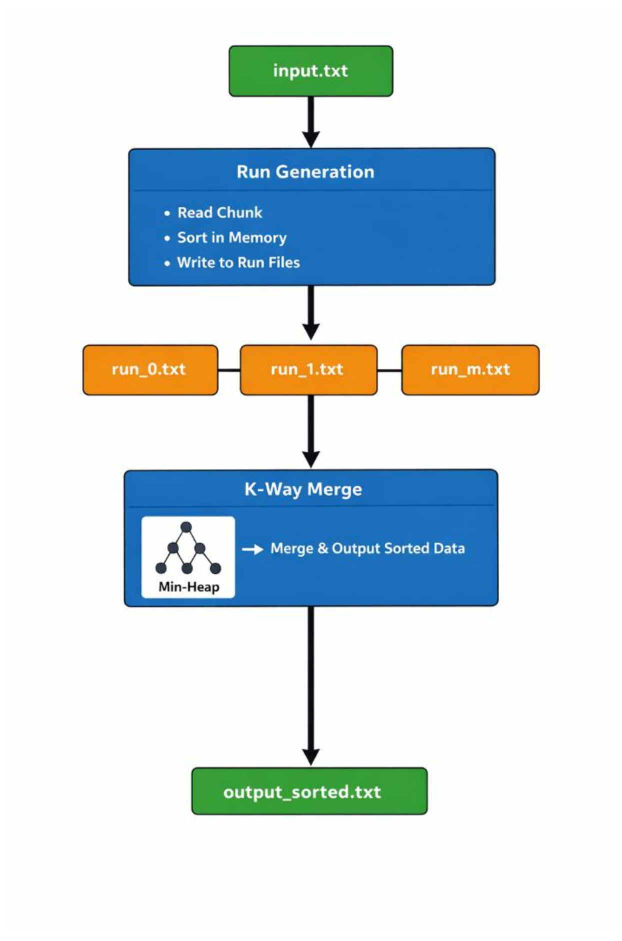
1.4 Solution.

The project’s solution is built around an external sorting approach, which is well-suited for scenarios where the input data size exceeds available RAM. Instead of loading the entire file into memory, the system processes the dataset in smaller portions that fit within a predefined memory limit, and then combines intermediate results to produce the final sorted output. The core method adopted is External Merge Sort, which consists of two phases: run generation and the merge phase.

In the first phase, the program reads data from input.txt in fixed-size blocks (chunks). Each chunk is temporarily stored in memory, sorted using an internal in-memory sorting routine, and then written to disk as a corresponding run file. This process repeats until the entire input file has been processed, resulting in multiple individually sorted runs.

In the second phase, the program merges these runs using a k-way merge strategy, where it repeatedly selects the smallest element among the current candidates from all runs and writes it to the output file. To perform this “minimum selection” efficiently, the system employs a min-heap (priority queue), ensuring fast extraction of the smallest element and stable updates as new elements are loaded from each run. As a result, the system not only guarantees the correctness of the sorted output but also maintains strong performance when handling large disk-resident datasets.

1.4.1 Solution diagram.



Overall Processing Flow:

+ **Step 1 – Chunk Reading and Partitioning (Chunking):** The system reads input.txt sequentially and divides the data into chunks whose sizes fit within the predefined memory buffer limit (chunk size).

+ **Step 2 – In-Memory Sorting (Internal Sorting):** Each chunk is sorted in RAM using Heap Sort, implemented via Heap::heapSort(...) in the heap module.

+ **Step 3 – Temporary Run File Generation (Run Files):** After sorting, each chunk is written to a temporary run file named run_0.txt, run_1.txt, and so on, which are stored in the project's temporary directory.

+ **Step 4 – K-way Merge (Merge Phase)**: The merger module opens all run files and uses a min-heap implemented with std::priority_queue (with a custom comparator) to continuously select the smallest element among the runs and write it sequentially to the output file.

+ **Step 5 – Output Generation:** The final result is produced as output_sorted.txt, which contains all input values sorted in ascending order.

1.4.2 Framework, environment and model.

+ **Algorithmic Model:** The system adopts an External Merge Sort model combined with a k-way merge strategy. During the run generation phase, data from the input file is read in chunks that conform to the memory limit. Each chunk is then sorted in RAM using a custom Heap Sort implementation (heap module) and written to disk as temporary run files (e.g., run_0.txt, run_1.txt, …). After all runs are generated, the system proceeds to the merge phase, where it opens the run files and employs a min-heap via std::priority_queue with a custom comparator in the merger module to repeatedly select the smallest element among all runs, thereby producing the final sorted output file.

+ **Programming Language and Standard:** The project is implemented in C++, targeting the C++17/C++20 standard. Leveraging the C++ standard library ensures ease of building and testing, as well as high compatibility across common development environments.

+ **Resource Management:** The system uses std::vector to store in-memory chunks during internal sorting. File input/output operations are handled with std::ifstream and std::ofstream, following RAII (Resource Acquisition Is Initialization) principles to ensure that file streams are properly closed when they go out of scope, thereby minimizing the risk of resource leaks.

+ **Modular Design:** The source code is organized in a modular manner, clearly separating algorithmic components from the control flow. The main modules include heap (internal sorting), chunker (data partitioning and run generation), and merger (k-way merging), while the overall coordination logic resides in sorter/main. This structure improves code clarity, maintainability, and facilitates future extensions of the system.

PART 2: PROJECT REQUIREMENTS.

2.1. Functional Requirements.

The Big Data File Sorter system is designed to sort data files whose sizes exceed available RAM capacity. The program takes a text-based input file and produces an output file sorted in ascending order. The sorting process follows an external sorting model, consisting of two main stages: run generation and run merging.

The system's core functionalities include:

+ The system reads data from input.txt and writes the sorted result to output_sorted.txt.

+ The system partitions the input into memory-friendly chunks, sorts each chunk in RAM, and writes the sorted chunks to temporary run files.

+ The system merges the run files using a k-way merge mechanism to produce the final sorted output.

+ The system provides supporting utilities for demonstration and testing, including tools for generating input datasets, verifying output correctness, and benchmarking runtime/performance.

2.2. Non-functional Requirements.

In addition to the core functionality, the system must satisfy several non-functional requirements to ensure reliability and usability in a big-data context:

+ **Correctness and Stability:** The system must guarantee that the output is correctly sorted for all valid inputs. It should operate reliably, handle file I/O robustly, and avoid unexpected crashes when processing large datasets.

+ **Performance**: The system is designed with I/O efficiency as a primary concern, since disk-based processing is typically dominated by read and write speeds. Therefore, it prioritizes chunk-based sequential access and appropriate buffering strategies to reduce the overhead of frequent small I/O operations. Performance is expected to scale reasonably as the dataset size increases, as demonstrated through benchmark results on datasets of varying sizes.

+ **Scalability and Maintainability:** The codebase should be clearly organized and modular to support testing and future modifications. Separating components such as chunking, internal sorting, and k-way merging makes it easier to upgrade algorithms or incorporate additional optimizations later.

+ **Presentation Transparency:** The processing pipeline should be easy to explain step by step—chunk reading and partitioning, chunk sorting, run generation, run merging, and final output production. This clarity supports effective demo videos and helps deliver a convincing, easy-to-follow defense presentation.

2.3. Constraints.

The project is developed under practical hardware and operating system constraints, which directly influence the design of the External Merge Sort architecture:

+ **Memory (RAM) Constraints:** The system operates under the fundamental premise that the dataset may significantly exceed available physical memory. To maintain stability, the implementation utilizes an incremental processing strategy, where only a single data chunk is loaded into RAM at any given time. In the current version, the chunk size is defined as a tunable parameter within main.cpp. This configuration ensures robust execution and prevents memory-related failures (such as std::bad_alloc) even when handling massive datasets—often ranging in the tens of gigabytes—that far surpass the system's physical memory capacity.

+ **I/O Bottleneck:** Because the dataset resides on disk, read/write operations can become the primary performance bottleneck. If the design is inefficient, an increased number of I/O operations can significantly prolong the runtime. Therefore, the system is designed to favor sequential streaming I/O and to minimize frequent small read/write requests.

+ **Limit on Concurrent Open Files:** Merging multiple run files requires opening many temporary files at the same time. However, operating systems impose limits on the number of file handles available to a process. This constraint must be considered when scaling the system (e.g., increasing the number of runs or increasing *k* in k-way merge).

+ **Input Data Format:** The input is stored in plain text, where each line contains a single integer. To ensure stable execution, the input must follow the expected format and avoid malformed lines or invalid characters.

2.4. Environment.

+ **Programming Language and Standard:** The project is implemented in C++, targeting the C++17 or C++20 standard. Adhering to modern C++ standards improves code clarity, enables effective use of STL containers, and enhances safety in resource management.

+ **Operating System and Compilers:** The program can be deployed on both Windows and Linux platforms. The source code is compatible with common compilers such as: MSVC (Visual Studio), g++ (MinGW/Linux), and Clang, making it convenient to build and run demos across different environments.

+ **Data Structures and Resource Management:** During the run generation phase, the system uses std::vector to store data chunks in RAM before internal sorting. In the merge phase, it employs std::priority_queue with an appropriate comparator to simulate a min-heap, enabling efficient k-way merging and continuous selection of the smallest element among runs.

For resource management, file I/O operations are handled using std::stream and std::ofstream, following the RAII (Resource Acquisition Is Initialization) principle. This ensures that file streams are properly closed when they go out of scope, reducing the risk of resource leaks and improving overall system stability when processing large datasets.

2.5. Acceptance Criteria.

The project is considered complete when the following criteria are satisfied:

+ The program executes successfully with input.txt and produces the output file output_sorted.txt.

+ The file output_sorted.txt is verified to be sorted in ascending order using the provided verification tool.

+ The system includes a data generation utility to support demonstrations and testing across multiple dataset sizes.

+ Benchmark results are provided on at least several datasets to demonstrate stable system behavior and measurable performance improvements through I/O optimization.

+The report and demo video clearly explain the processing workflow and accurately reflect the algorithms implemented in the repository (Heap Sort for chunk sorting and a priority queue–based merge phase).

PART 3: THEORETICAL BACKGROUND.

3.1. Overview of the Problem Context.

In modern data systems, sorting is a fundamental operation that supports many tasks such as normalizing time-ordered logs, deduplication, grouping, and query optimization. When datasets are small, in-memory sorting is a straightforward choice because the entire dataset can be loaded into RAM for processing. However, when the dataset becomes large enough to exceed available RAM, this approach is no longer practical. If the program attempts to load all data into memory, it may fail due to insufficient memory allocation or be forced into heavy swapping by the operating system, leading to severe performance degradation.

The problem addressed in this project falls into the “data > RAM” category, where data resides on disk and must be processed in a way that matches the memory hierarchy. Therefore, the appropriate approach is external sorting: partition the dataset into memory-sized chunks, sort each chunk independently, and then merge the intermediate results to produce the final fully sorted output file.

3.2. Complexity Analysis and Big-O Notation.

Big-O notation is used to describe the asymptotic upper bound of time or space complexity with respect to the input size n. In the context of external sorting, analyzing CPU time alone is insufficient; instead, Big-O must be considered as a combination of computation cost and data access cost:

+ **Computational cost (CPU complexity):** The time required to sort individual data chunks and to maintain the heap structure during the merge phase.

+ **Data access cost (I/O complexity):** The number of data blocks read from and written to disk.

The system targets an overall time complexity of O(n log n) and an auxiliary memory complexity of O (M), where MMM denotes the size of the chunk loaded into RAM at any given time. This design enables stable operation under constrained memory resources.

3.3. External Sorting and the External-Memory Model.

External sorting is a technique used when a dataset cannot fit entirely in RAM. The main idea is to process the data in smaller parts, write intermediate sorted results to disk, and then merge these sorted parts to produce the final output. This technique aligns well with the external-memory model, because disks are typically much more efficient when performing block-based and sequential read/write operations.

In practice, operating systems and hardware are not optimized for fine-grained access at the level of individual elements. If a program reads or writes data line by line, the number of I/O operations can grow dramatically, reducing overall throughput.

In contrast, using sufficiently large buffers helps reduce system call overhead and takes advantage of OS-level caching. For this reason, in external sorting, choices such as buffer design, chunk size, and the number of passes over the data are just as important as the internal sorting algorithm itself.

3.4. External Merge Sort.

External Merge Sort is the most widely used external sorting method due to its clear structure and strong I/O efficiency. The algorithm consists of two main phases:

+ **Phase 1 – Run Generation**: The input data is read in chunks that fit into RAM. Each chunk is sorted using an internal sorting algorithm and then written to disk as a temporary file called a run. At the end of this phase, multiple run files are produced, each already sorted in ascending order.

+ **Phase 2 – Merge Phase**: The sorted runs are merged to generate the final output file. Since each run is already ordered, the merge process can be performed by repeatedly selecting the smallest element among the current leading elements of all active runs. When multiple runs are merged simultaneously using a k-way merge strategy, the number of merge passes is reduced, which in turn decreases the total amount of disk I/O.

External Merge Sort is particularly well suited for disk-resident data because it emphasizes sequential read/write access, allows precise memory control through chunk size selection, and clearly separates in-memory processing from disk-based processing.

3.5. Internal Sorting Used in Run Generation.

In the project repository, each chunk is sorted using a custom Heap Sort implementation developed by the team. Heap Sort is based on the heap data structure and consists of two main steps: building a heap from the input array and repeatedly moving the extreme element (maximum or minimum) to the end of the array to produce the sorted order.

In terms of complexity, Heap Sort runs in O(n log n) time in all cases. This is an important property because it avoids the worst-case(n^2) behavior that can occur in some quicksort variants when the input data has unfavorable patterns. In addition, Heap Sort requires very little auxiliary memory, as it operates in place, making it suitable for scenarios where memory usage per chunk must be tightly controlled.

However, in practice, Heap Sort may perform worse than std::sort in many cases because heap operations tend to access memory in a non-contiguous, tree-like pattern, resulting in poorer cache locality. Nevertheless, within the scope of a DSA course, using Heap Sort clearly demonstrates understanding of heap-based algorithms while providing strong theoretical worst-case guarantees.

3.6. Heap / Priority Queue and K-way Merge.

After the sorted runs are generated, the problem reduces to merging multiple increasing sequences into a single final sorted sequence. If only two runs are merged at a time, the number of merge passes can become large, leading to repeated full scans of the data and increased disk I/O. A k-way merge allows k runs to be merged simultaneously, thereby reducing the number of merge passes.

In this project, k-way merge is implemented using std::priority_queue with a custom comparator to simulate a min-heap. At any moment, the heap stores the current element from each run (typically the *next unconsumed element* of that run). The standard procedure is as follows:

+ Insert the first element of each run into the heap.

+ Repeatedly extract the smallest element from the heap and write it to the output; then read the next element from the same run and insert it into the heap.

+ When a run is exhausted, it no longer contributes elements to the heap. The process terminates when the heap becomes empty.

The complexity of the merge phase can be estimated as follows: with a total of nnn elements, each element participates in exactly one pop operation and, except for the last element of each run, one push operation. Each heap operation costs O(log k). Therefore, the CPU complexity of the merge phase is approximately O(n lok). This efficiency explains why heap-based structures are particularly well suited for multi-way merging problems.

3.7. I/O Bottleneck, Buffering, and Cache Locality.

In external sorting, I/O operations often dominate the total runtime because data must be repeatedly transferred between RAM and disk. Three key factors have a strong impact on practical performance:

+ **I/O Bottleneck:** Even when the CPU-side algorithm has a time complexity of O(n log n), the program can still be slow if it performs excessive or fine-grained disk I/O. With large datasets, the total volume of data read and written may be several times larger than the input size due to the creation of run files and the final output.

+ **Buffering:** Instead of reading or writing individual numbers, the system should perform I/O using buffers (e.g., groups of lines or data blocks). This approach reduces the number of I/O system calls, increases effective throughput, and better utilizes operating system caching mechanisms.

+ **Cache Locality:** Within RAM, accessing contiguous memory locations is generally faster than accessing scattered ones. This affects both the choice of internal sorting algorithm and the organization of buffers. Although the current project uses Heap Sort to demonstrate DSA concepts, the performance evaluation should acknowledge that cache locality is an important empirical factor influencing real-world performance.

3.8. Trade-offs and Key Parameters.

External sorting inherently involves trade-offs between memory usage, I/O cost, and CPU computation. Several parameters strongly influence system behavior:

+ Chunk size: Larger chunks reduce the number of runs, thereby lowering the load on the merge phase and reducing the number of temporary files. However, large chunks require more RAM and may compromise stability if they exceed the memory budget. Smaller chunks provide tighter memory control but increase the number of runs, leading to higher merge cost and greater overall I/O overhead.

+ Value of k in k-way merge: A larger k reduces the number of merge passes, but each heap operation becomes more expensive due to the log(k) factor, and the number of concurrently open files increases (subject to operating system limits). In addition, if buffering is applied per run, a larger k consumes more RAM for buffers.

+ Data format (text): Text-based data is easy to inspect and convenient for demonstrations, but it introduces parsing overhead and typically occupies more storage than binary formats. This is a known limitation and a potential direction for future improvement (e.g., switching to binary runs to improve performance).

3.8.1. Runs and Merge Passes Estimation.

+ To better understand the trade-offs between chunk size, number of runs, and I/O cost, we can estimate key quantities in External Merge Sort.

Assume:

+ N = size of the input data (measured in number of elements or bytes)

+ M = maximum amount of data that can be processed in RAM per chunk

The number of runs generated during the Run Generation phase is approximately:

R ≈ ceil(N / M)

+ This means that a smaller chunk size (smaller M) leads to a larger number of runs R, increasing the workload of the merge phase. Conversely, a larger chunk size reduces R but requires more RAM and increases the risk of exceeding memory limits.

+ During the Merge Phase, if the system can merge up to k runs at a time (k-way merge), the number of merge passes can be estimated as:

P ≈ ceil(log_base_k (R))

where log_base_k(R) denotes the logarithm of R with base k.

3.9. Mapping DSA Topics to This Project.

The Big Data File Sorter project directly integrates core Data Structures and Algorithms (DSA) concepts through the design and implementation of an external sorting system. At the highest level, External Merge Sort serves as the main algorithmic framework, enabling the system to handle datasets larger than available RAM through two well-defined phases: generating sorted runs from individual chunks and merging these runs to produce the final output file.

During the run generation phase, the project applies a custom Heap Sort implementation to sort each chunk in memory. This choice demonstrates understanding of the heap data structure and provides a stable time complexity of O(n log n) for internal sorting. In the merge phase, the system employs a min-heap implemented via std::priority_queue with a custom comparator to perform k-way merging, ensuring that the smallest element among all active runs is selected at each step, with a heap operation cost of approximately O(log k).

In addition, the project applies Big-O complexity analysis from a practical big-data perspective, emphasizing the impact of I/O bottlenecks and the trade-offs among chunk size, number of runs, and number of merge passes. As a result, the system is not only algorithmically correct but also aligned with the practical requirements of large-scale data processing in an external-memory environment.

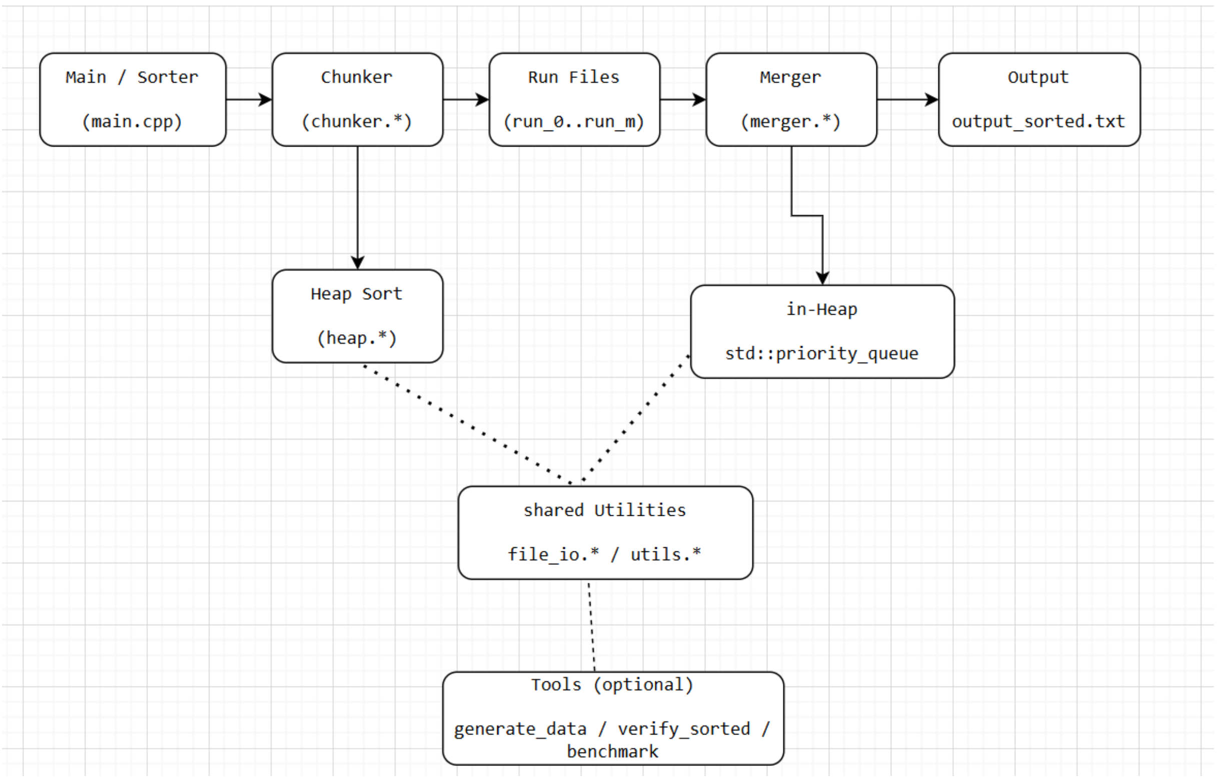
PART 4: SYSTEM DESIGN & ARCHITECTURE.

4.1. Overall Architecture.

The Big Data File Sorter system is designed as a pipeline architecture built around the External Merge Sort algorithm. This design allows data to flow through sequential processing stages, ensuring a clear separation between the orchestration logic and the core algorithmic components.

The system consists of two primary processing blocks: Run Generation (creating temporary sorted segments) and the Merge Phase (merging sorted runs). These stages are coordinated by a unified main program. The codebase follows a modular architecture, which improves maintainability, enables independent testing of components, and supports future scalability as dataset sizes grow.

Within the repository, the orchestration flow is centralized in the Sorter layer (invoked from main.cpp). This component is responsible for configuring input/output file paths, setting key parameters such as chunkSize, and managing the lifecycle of the three major stages: run generation, run merging, and cleanup of intermediate artifacts.



Legend:

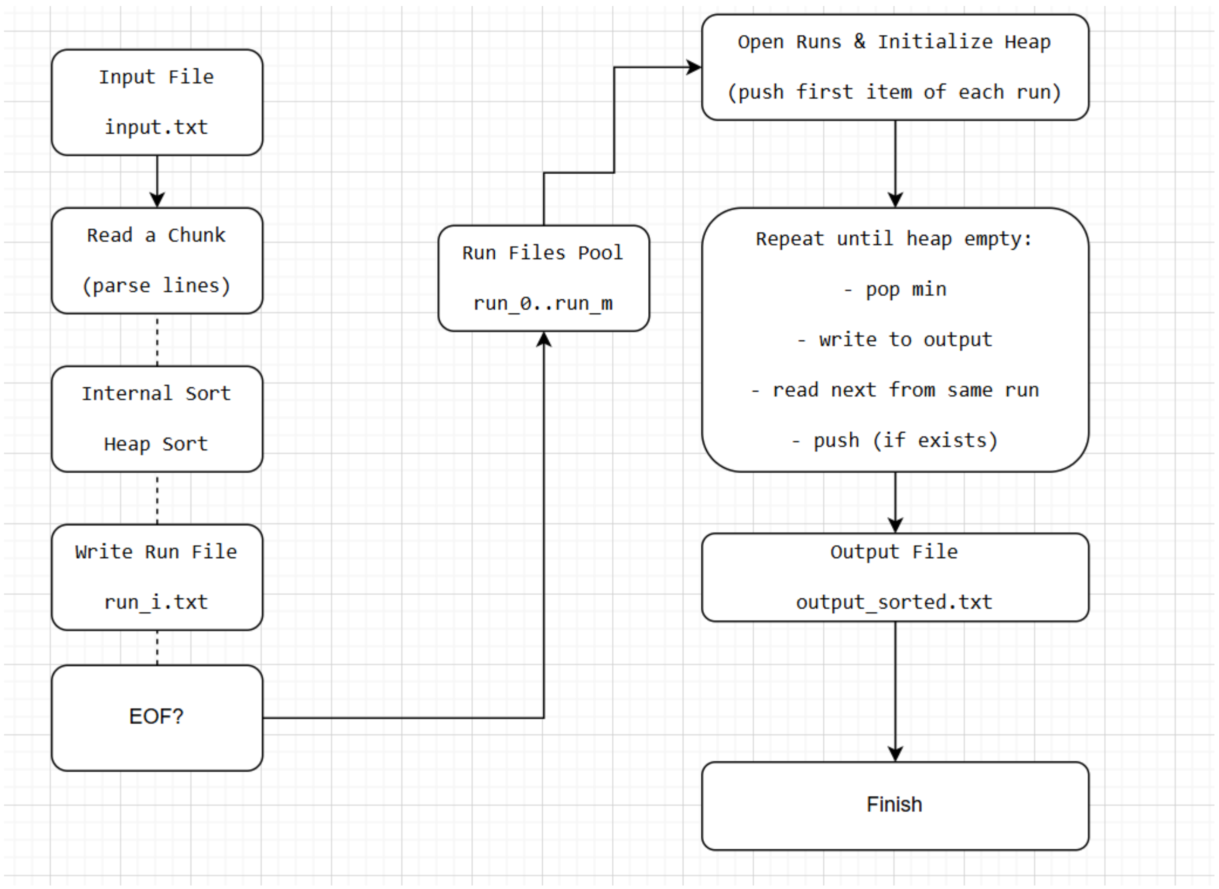
+ Main data/control flow

+ Support / helper usage

4.2. Data Flow & Processing Pipeline.

The processing flow begins with the input.txt file. Instead of loading the entire dataset into RAM, the system reads the data in chunks whose sizes are configured in the orchestration program. Each chunk, once loaded into memory, is sorted in ascending order and written to disk as a temporary run file. This process repeats until the entire input file has been consumed, resulting in a collection of partially sorted run files.

After the run generation phase, the system transitions to the merge phase. In this stage, the run files are opened for sequential reading and merged using a k-way merge strategy. The system maintains a min-heap that stores the current element from each run. Each time the smallest element is extracted from the heap and written to output_sorted.txt, the system reads the next element from the same run and inserts it back into the heap. Through this mechanism, the output is produced in strictly ascending order without ever requiring the entire dataset to be loaded into memory.



legend:

+ Processing flow

+ Loop (continue reading)

4.3. Module Decomposition.

Modular decomposition clarifies responsibilities and makes the system easier to test. The chunker module focuses on reading the input file in chunks and generating sorted run files. It acts as a transformation stage that converts “raw disk data” into “partially sorted segments,” preparing the data for efficient merging.

The heap module provides a custom Heap Sort implementation for in-memory sorting. Separating the internal sorting algorithm into its own module helps highlight core DSA knowledge and prevents the chunker module from becoming bloated with algorithm-specific logic.

The merger module is responsible for merging the sorted run files. Its central data structure is a min-heap (implemented using std::priority_queue with a custom comparator), which enables fast selection of the smallest element among the current candidates from all runs and guarantees that the final output is produced in ascending order.

Module	File (Reference)	Role	Input	Output
Main/Sorter	main.cpp / sorter.*	Orchestrates execution flow, configures parameters, and invokes modules	Paths + parameters	Calls chunker/merger
Chunker	chunker.*	Reads input in chunks, sorts each chunk, and generates run files	input.txt	run_0..run_m
Heap	heap.*	Custom Heap Sort implementation for in-memory chunk sorting	std::vector (chunk)	Sorted chunk
Merger	merger.*	Performs k-way merge using a min-heap (priority queue)	Run files	output_sorted.txt
File I/O + Utils	file_io.* / utils.*	Common file I/O routines and shared utilities for stable processing	Data / paths	Stable I/O utilities
Tools	generate_data.cpp / verify_sorted.cpp / benchmark.cpp	Utilities for data generation, output verification, and performance benchmarking	Test parameters	Test files + results

4.4. Chunking Strategy / Run Generation.

In the “data > RAM” setting, the goal of the run generation phase is to control memory usage and prevent out-of-memory failures. The system reads input.txt in fixed-size chunks. Each chunk is stored in a std::vector, sorted using Heap Sort, and then written to disk as a corresponding run file.

This strategy offers two key benefits. First, memory usage remains bounded because the program holds at most one chunk in RAM at any time. Second, the dataset becomes “pre-sorted” into multiple ascending runs, which simplifies the workload of the subsequent merge phase.

4.5. Merge Strategy / K-way Merge Design.

The goal of the merge phase is to combine all sorted runs into a single output file while still keeping memory usage under control. The system performs a k-way merge based on a min-heap: initially, each run contributes its first element to the heap. The system then repeatedly extracts the smallest element from the heap, writes it to the output, and loads the next element from the same source run.

This design guarantees correctness because the heap always contains the smallest remaining candidate from each run at every step. At the same time, memory usage is well controlled: the heap only needs to store a number of elements proportional to the number of runs, not to the total dataset size.

4.6. Parameter Selection and Trade-offs.

System performance is highly sensitive to the chunk size and the resulting number of runs. A larger chunk size reduces the number of runs, which in turn lowers merge pressure and decreases extra read/write overhead. However, an overly large chunk increases RAM requirements and may reduce stability on low-spec machines. Conversely, a smaller chunk size provides tighter memory control but produces more run files, making the merge phase heavier and increasing total I/O.

In addition, when the number of runs becomes very large, opening too many files simultaneously may hit the operating system’s file handle limits. This is why large-scale systems often require multi-pass merging strategies or enforce an upper bound on how many runs can be merged at once. In the current project, k-way merge is chosen based on available machine resources, and this is also a potential optimization direction to discuss in the Future Work section.

Parameter	Main Impact	When Increased	When Decreased
Chunk size (M)	Determines number of runs and RAM usage	Fewer runs, lighter merge phase, but higher RAM usage	More runs, heavier merge phase, higher I/O overhead
Number of runs (R)	Affects heap size, merge cost, and number of temp files	Larger heap, more temporary files	Lighter merge, fewer temporary files
k (k-way merge)	Number of runs merged simultaneously	Fewer merge passes, higher log(k), more open	More merge passes, higher total I/O

		files, higher buffer usage	
Buffer size	Reduces fine-grained I/O, improves throughput	Fewer syscalls, more efficient I/O, but higher RAM usage	More small I/O operations, slower performance
File format	Text is easy to demo but costly to parse	Binary:faster, smaller size, harder to debug	Text: easier to inspect, but slower

4.7. Error Handling & Edge Cases.

To ensure stable behavior during demonstrations, the system explicitly considers common edge cases and defines appropriate handling strategies. If input.txt does not exist or cannot be opened, the program terminates gracefully and reports a clear error message. If the input file is empty, the system still generates a valid output file (which may also be empty).

During the merge phase, when a run reaches end-of-file (EOF), the system stops loading further elements from that run to avoid out-of-bounds reads and to ensure proper termination of the merge loop.

Because the input data is stored in text format, the project assumes that each line contains a valid integer for demonstration purposes. This assumption is explicitly stated in the Requirements section to prevent malformed input data from causing unexpected runtime errors or interrupting the execution flow.

Scenario	Description	Expected Handling
Cannot open input file	Invalid file path or insufficient access permissions	Report a clear error message and terminate the program
Empty input file	The input file contains no data	Generate a valid output file (possibly empty)
Run exhausted	End-of-file (EOF) reached during merge phase	Stop pushing elements from this run and continue with remaining runs
Invalid data format	A line in the input is not a valid integer	State the assumption of valid input for demo or add exception handling
Too many run files	Exceeds OS file handle limits when opening runs simultaneously	Limit k or perform multi-pass merging (future work)
Scenario	Description	Expected Handling

PART 5: ALGORITHM ANALYSIS.

This chapter analyzes the performance of the Big Data File Sorter system from both an algorithmic and a system-level perspective. Since the dataset size may exceed available RAM, real-world runtime is influenced not only by CPU complexity (Big-O) but also heavily by disk I/O costs. Therefore, the evaluation is conducted based on four main criteria:

+ Time complexity of each processing phase

+ Memory usage

+ I/O cost and bottlenecks

+ Trade-offs when varying parameters such as chunk size and the number of runs merged concurrently

5.1. Complexity Analysis.

5.1.1. Time Complexity.

The total execution time of External Merge Sort is the sum of the costs of its two sequential phases:

+ **Phase 1 – Run Generation:** Assume the input file contains *n* elements, and each chunk loaded into RAM has size *k*. The number of temporary run files generated is *m = n / k*.

+ Each chunk is sorted using Heap Sort with time complexity *O(k log k)*.

+ The total cost for all chunks is:
T1 = m × O(k log k) = (n / k) × O(k log k) = O(n log k).

+ **Phase 2 – K-way Merge:** The system uses a min-heap to manage the leading elements of the *m* run files.

+ Each of the *n* elements is extracted from and (except the last element of each run) inserted into the heap exactly once.

+ Each heap operation costs *O(log m)*.

+ The total merge cost is:
T2 = O(n log m).

Overall Time Complexity:
T = T1 + T2 = O(n log k) + O(n log m)
Since *m = n / k*, this simplifies to:
T = O(n log n).

This is the optimal asymptotic complexity for comparison-based sorting algorithms, ensuring that the system remains efficient even as the dataset size grows exponentially.

5.1.2. Space Complexity.

The system demonstrates strong efficiency in memory resource management:

+ **RAM usage:** At any time, the system maintains only one data chunk of size *O(k)* and a heap of size *O(m)*. This allows sorting datasets of tens of gigabytes using only a few hundred megabytes of RAM.

+ **Disk usage:** The system requires up to 2 × O(n) disk space to store temporary run files and the final sorted output.

5.2. I/O Bottleneck.

In external sorting, disk speed is typically several orders of magnitude slower than RAM, making I/O the primary system bottleneck.

+ **I/O Optimization:** The system is designed to read and write data in large blocks, prioritizing sequential access over random access. This approach maximizes the effective bandwidth of HDDs/SSDs and leverages operating system buffering mechanisms.

+ **I/O Complexity:** Under the external-memory model, the I/O complexity can be expressed as:
O((n / B) × log_(M / B)(n / B)),
where *B* is the disk block size and *M* is the available RAM.
By employing k-way merge, the system minimizes the number of full data passes over the disk—typically requiring only one read and one write per element—thereby achieving near-optimal I/O efficiency.

5.3. Experimental Benchmarks.

The system was tested on an AMD Ryzen 7 hardware platform equipped with an NVMe SSD, using a dataset of 1,000,000 randomly generated integers.

Chunk Size	Number of Runs/Chunks	Phase 1 Time (s)	Phase 2 Time (s)	Total Time (s)
1.00 MB	4 chunks	4,1	4,4	8,5
2.00 MB	2 chunks	4,1	4,3	8,4
5.00 MB	1 chunk	4,1	4,3	8,4

Observations:

+ Phase 1 performance is highly stable, consistent with the O(n log k) time complexity of Heap Sort.

+ The primary bottleneck occurs in Phase 2, where the speed of reading data from temporary run files on disk limits the overall execution time.

+ Increasing the chunk size reduces the number of temporary files, which in turn lowers file-handle switching overhead and results in a slight improvement in system throughput.

5.4. Performance Trade-offs.

Configuring the system requires balancing key technical parameters to achieve the best overall efficiency:

+ **Chunk Size and RAM Usage:** Larger chunks reduce the number of runs and improve the merge phase (Phase 2), but they increase the risk of memory exhaustion (e.g., std::bad_alloc). Smaller chunks are safer for memory but produce more run files, increasing file management overhead and enlarging the heap depth during merging.

+ **Heap Sort Selection:** The project uses Heap Sort instead of Quick Sort for internal sorting due to its guaranteed O(n log n) time complexity in all cases and its in-place behavior. Although Quick Sort often has better cache locality, Heap Sort protects the system from worst-case performance degradation.

+ Data Format Choice: The current system uses a text-based format to ensure transparency and ease of verification for the course project. However, converting strings to integers introduces noticeable CPU overhead. In real-world applications, binary formats are typically preferred to optimize both storage efficiency and processing speed.

PART 6: IMPLEMENTATION DETAILS.

6.1. Project Structure.

The project is organized using a modular C++ approach, with a clear separation between interface definitions (header files) and logic implementations (source files). This structure improves code manageability and long-term maintainability.

+ **Root directory:** Contains project configuration files (e.g., CMakeLists.txt, .gitignore) and documentation (e.g., README.md, design.md).

+ **Orchestration module:** sorter.h and main.cpp manage the program lifecycle and connect the main components.

+ **Chunking module:** chunker.h and chunker.cpp handle file reading and generate temporary run files.

+ **Merge module:** merger.h and merger.cpp implement the **k-way merge** strategy.

+ **Algorithm module:** heap.h and heap.cpp manually implement the Heap data structure and the **Heap Sort** algorithm.

+**Utilities module:** file_io.h/.cpp manages file system operations, while utils.h/.cpp provides helper functions such as timing utilities and data formatting.

6.2. Algorithm Module.

The Heap Sort algorithm is manually implemented in the Heap class instead of using built-in library functions, in order to demonstrate direct application of data-structure knowledge.

+ **Heapify mechanism:** The function heapifyDownForSort maintains the Max-Heap property of the binary heap, ensuring that each parent node is always greater than its children.

+ **Sorting procedure:**

1. Build a Max-Heap from the input array by iterating from the middle of the array back to the beginning.
2. Swap the largest element (at the root) with the last element of the current array range.
3. Reduce the heap size and apply heapifyDownForSort at the root to restore the heap, bringing the next largest element to the top.
4. Repeat until the array is fully sorted.

6.3. Merger Module.

The merge phase is the most complex component because it must manage multiple input file streams simultaneously.

+ **MergeElement:** A lightweight data structure that stores an integer value and the index of the run file it comes from. The comparator (e.g., overloaded operator> or a custom comparator) is used to turn the default max-heap behavior of std::priority_queue into a min-heap.

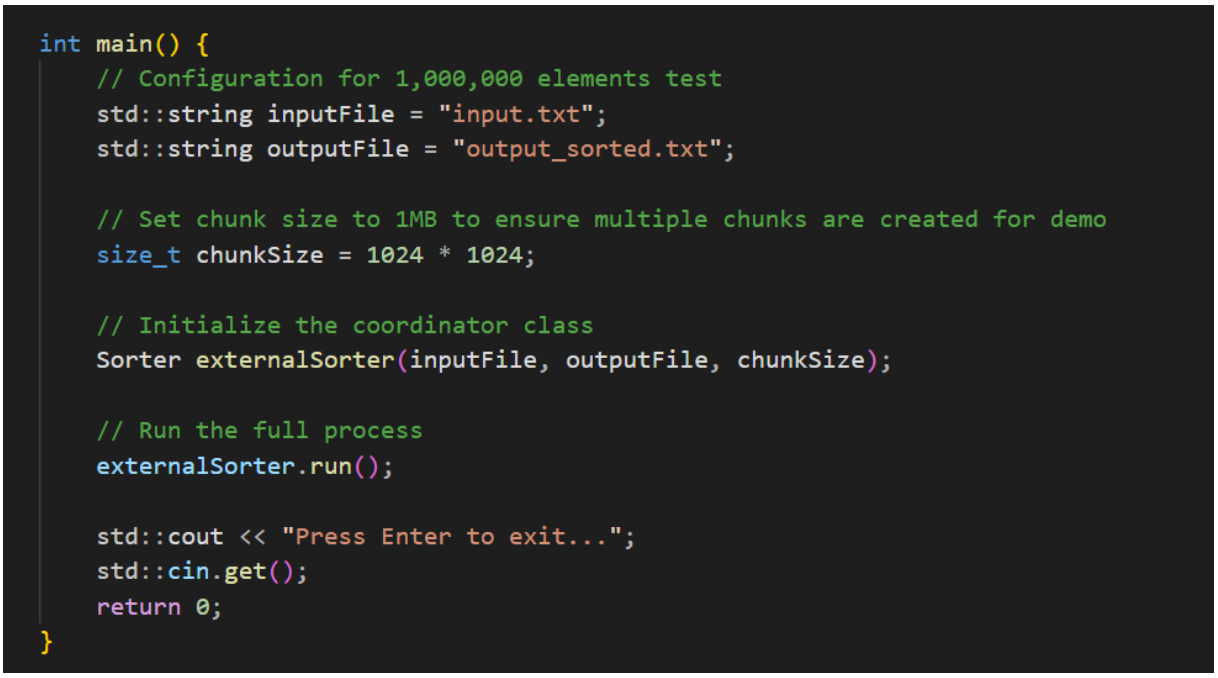
+ **merge() function workflow:**

1. Open all temporary run files generated in Phase 1.
2. Read the first element from each run file and push it into the min-heap.
3. Repeatedly extract the smallest element from the heap, write it to the output file, then read the next element from the same source run and push it into the heap.
4. The process ends when all run files are exhausted and the heap becomes empty.

6.4. Key Code Snippets.

6.4.1. Program Startup (Main).

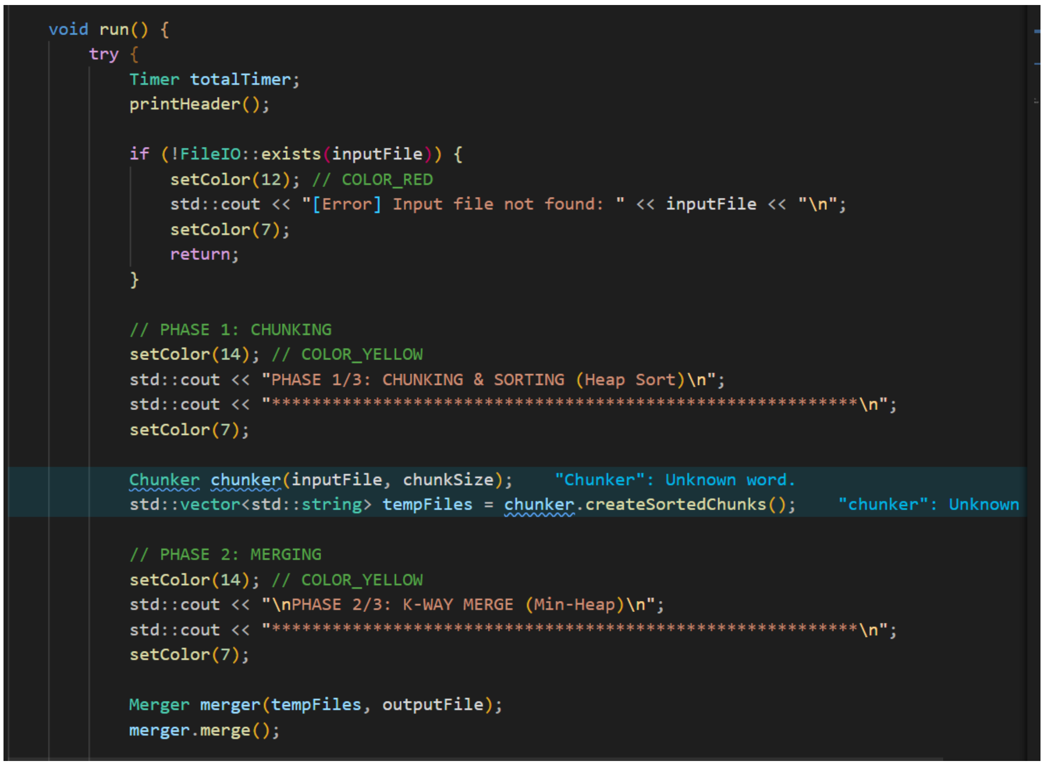
The program configures the input/output paths and the chunk size, then initializes the Sorter component and calls run() to execute the entire pipeline



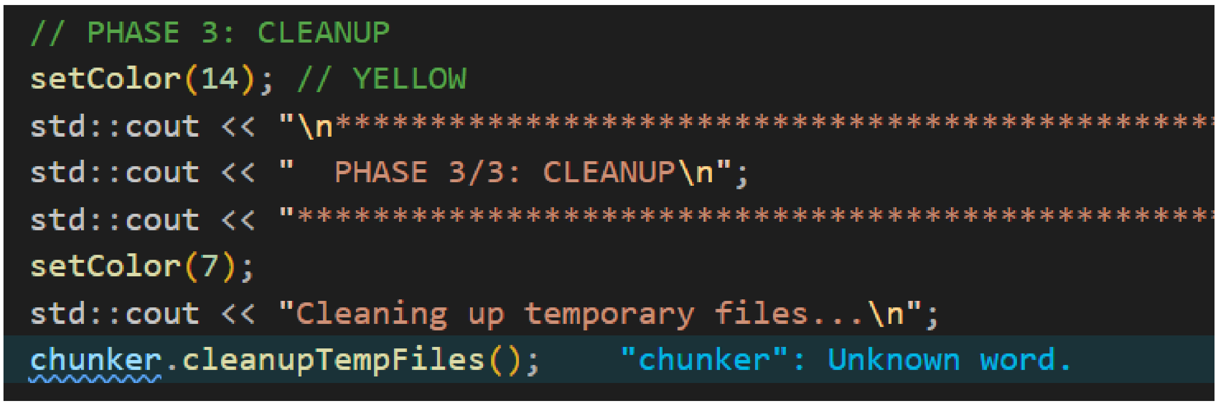
```cpp
int main() {
    // Configuration for 1,000,000 elements test
    std::string inputFile = "input.txt";
    std::string outputFile = "output_sorted.txt";

    // Set chunk size to 1MB to ensure multiple chunks are created for demo
    size_t chunkSize = 1024 * 1024;

    // Initialize the coordinator class
    Sorter externalSorter(inputFile, outputFile, chunkSize);

    // Run the full process
    externalSorter.run();

    std::cout << "Press Enter to exit...";
    std::cin.get();
    return 0;
}
```

6.4.2. Three-Phase Orchestration (Sorter::run).

The run() function orchestrates the three main phases of the system in sequence: run generation (chunking and internal sorting), run merging (k-way merge), and cleanup of temporary files after completion.



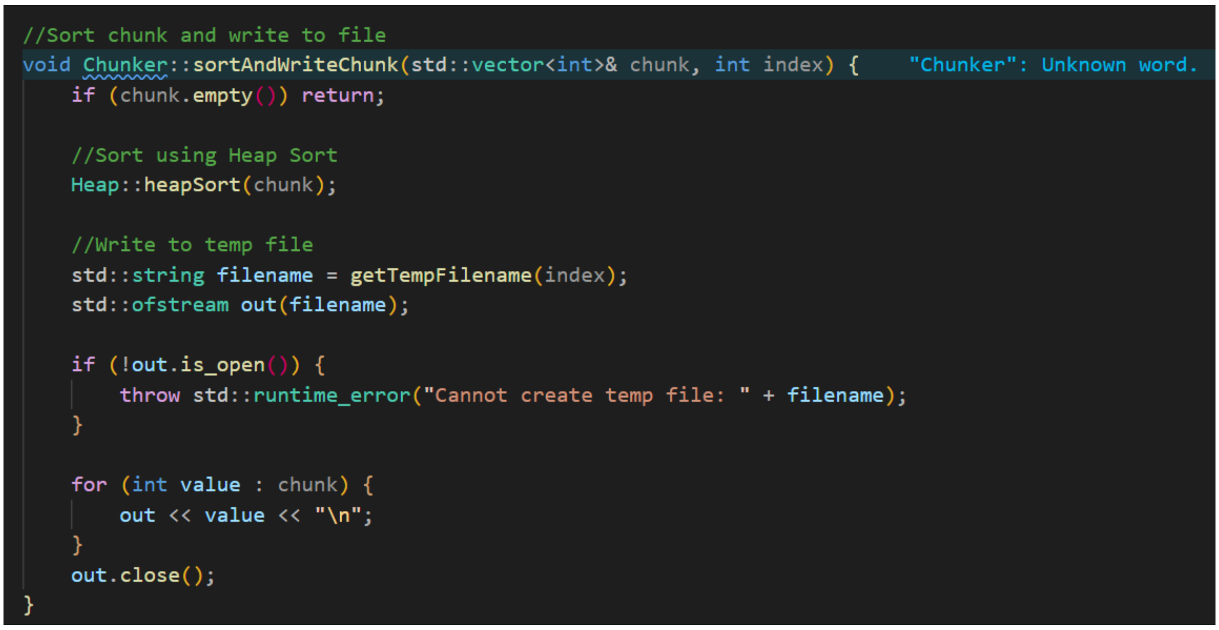
```
void run() {
    try {
        Timer totalTimer;
        printHeader();

        if (!FileIO::exists(inputFile)) {
            setColor(12); // COLOR_RED
            std::cout << "[Error] Input file not found: " << inputFile << "\n";
            setColor(7);
            return;
        }

        // PHASE 1: CHUNKING
        setColor(14); // COLOR_YELLOW
        std::cout << "PHASE 1/3: CHUNKING & SORTING (Heap Sort)\n";
        std::cout << "************************************************************\n";
        setColor(7);

        Chunker chunker(inputFile, chunkSize);    "Chunker": Unknown word.
        std::vector<std::string> tempFiles = chunker.createSortedChunks();    "chunker": Unknown

        // PHASE 2: MERGING
        setColor(14); // COLOR_YELLOW
        std::cout << "\nPHASE 2/3: K-WAY MERGE (Min-Heap)\n";
        std::cout << "************************************************************\n";
        setColor(7);

        Merger merger(tempFiles, outputFile);
        merger.merge();
```



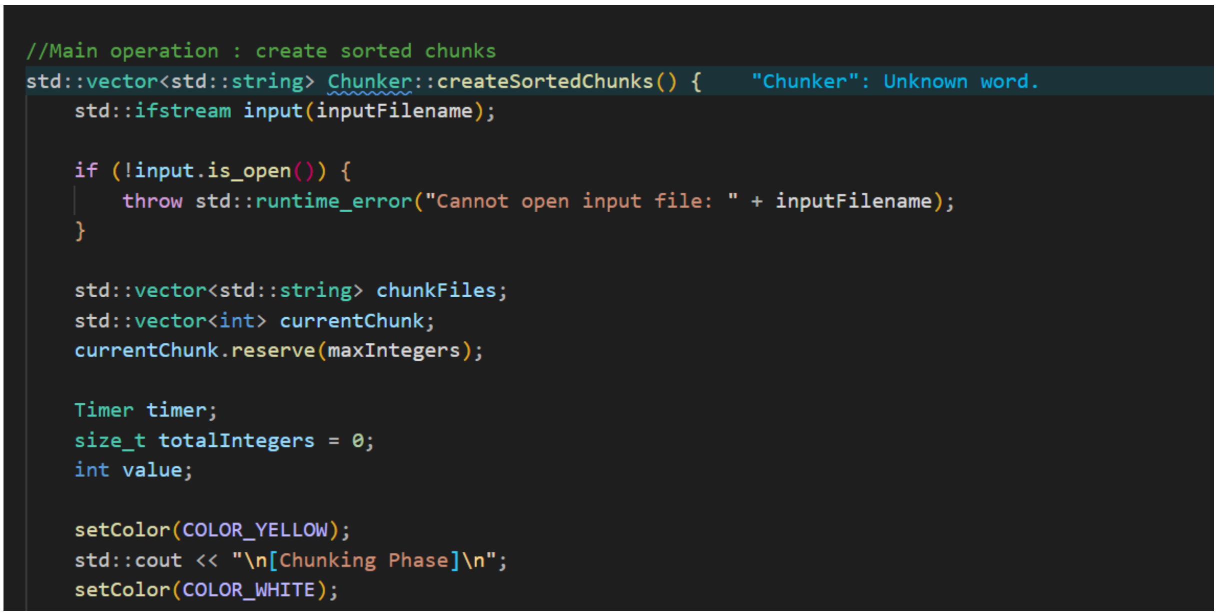
```
// PHASE 3: CLEANUP
setColor(14); // YELLOW
std::cout << "\n*******************************************
std::cout << "  PHASE 3/3: CLEANUP\n";
std::cout << "*********************************************
setColor(7);
std::cout << "Cleaning up temporary files...\n";
chunker.cleanupTempFiles();    "chunker": Unknown word.
```

<u>*6.4.3. Run Generation (Chunker).*</u>

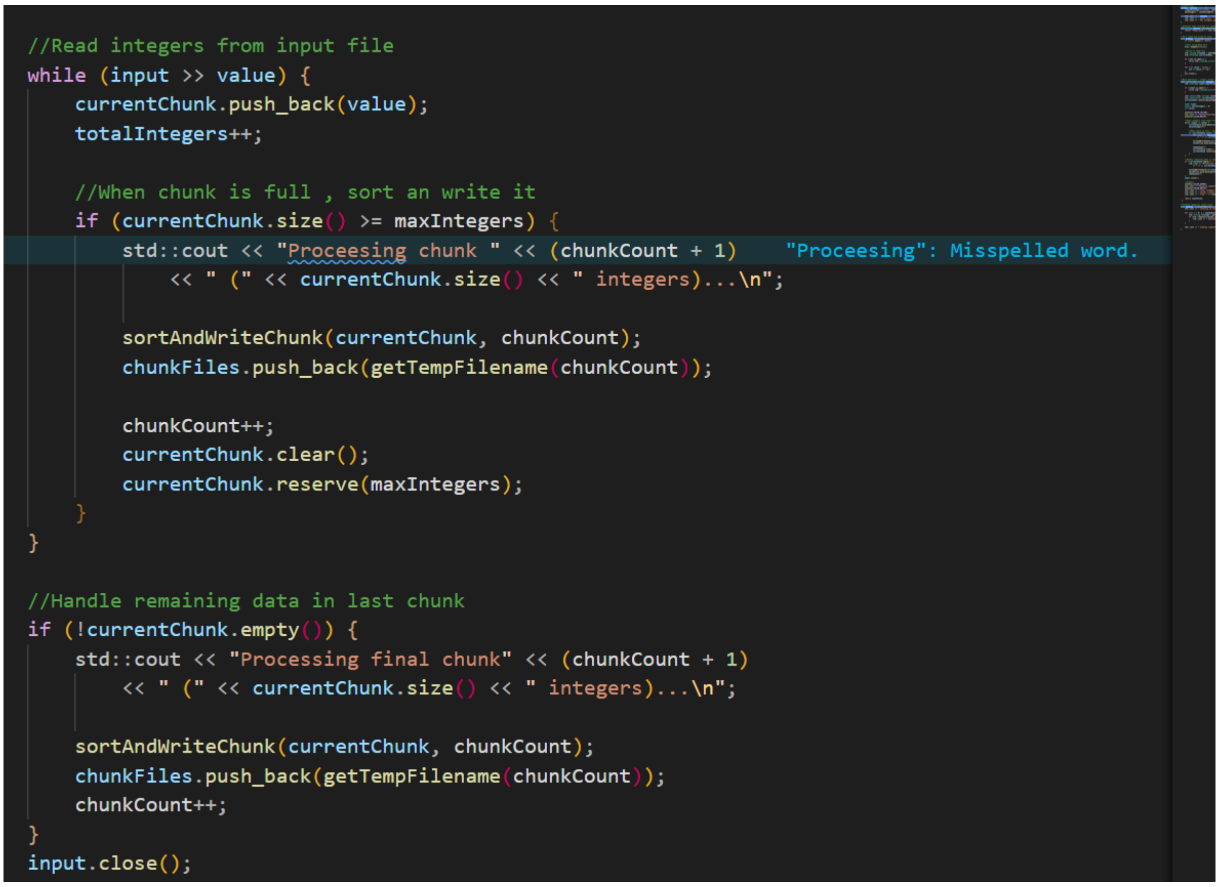
The Chunker reads the input data in memory-sized chunks, performs internal sorting on each chunk, and writes the sorted results to temporary files named run_i. These run files serve as the input for the subsequent merge phase.



```
//Sort chunk and write to file
void Chunker::sortAndWriteChunk(std::vector<int>& chunk, int index) {    "Chunker": Unknown word.
    if (chunk.empty()) return;

    //Sort using Heap Sort
    Heap::heapSort(chunk);

    //Write to temp file
    std::string filename = getTempFilename(index);
    std::ofstream out(filename);

    if (!out.is_open()) {
        throw std::runtime_error("Cannot create temp file: " + filename);
    }

    for (int value : chunk) {
        out << value << "\n";
    }
    out.close();
}
```



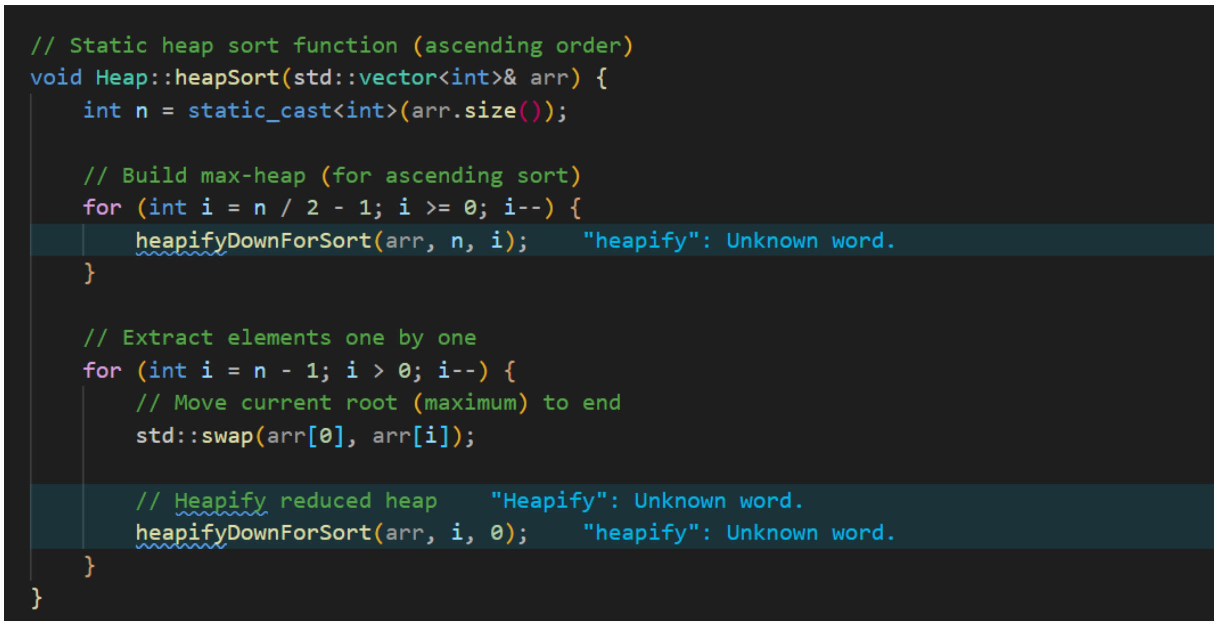
```
//Main operation : create sorted chunks
std::vector<std::string> Chunker::createSortedChunks() {    "Chunker": Unknown word.
    std::ifstream input(inputFilename);

    if (!input.is_open()) {
        throw std::runtime_error("Cannot open input file: " + inputFilename);
    }

    std::vector<std::string> chunkFiles;
    std::vector<int> currentChunk;
    currentChunk.reserve(maxIntegers);

    Timer timer;
    size_t totalIntegers = 0;
    int value;

    setColor(COLOR_YELLOW);
    std::cout << "\n[Chunking Phase]\n";
    setColor(COLOR_WHITE);
```



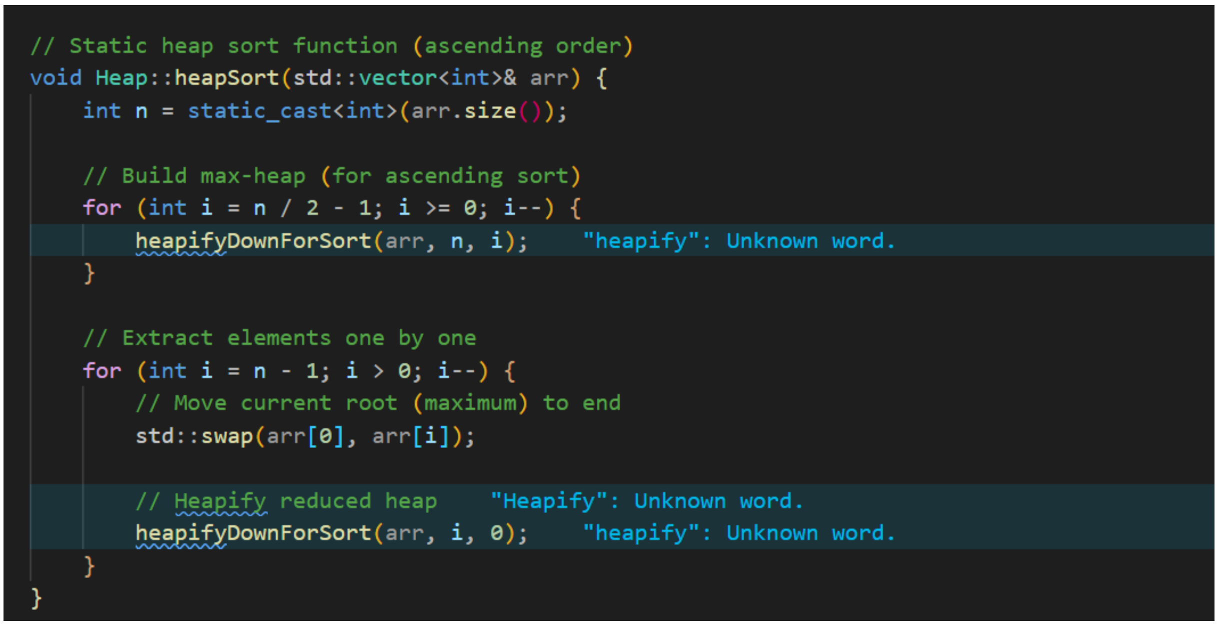
```cpp
//Read integers from input file
while (input >> value) {
    currentChunk.push_back(value);
    totalIntegers++;

    //When chunk is full , sort an write it
    if (currentChunk.size() >= maxIntegers) {
        std::cout << "Proceesing chunk " << (chunkCount + 1)    "Proceesing": Misspelled word.
            << " (" << currentChunk.size() << " integers)...\n";

        sortAndWriteChunk(currentChunk, chunkCount);
        chunkFiles.push_back(getTempFilename(chunkCount));

        chunkCount++;
        currentChunk.clear();
        currentChunk.reserve(maxIntegers);
    }
}

//Handle remaining data in last chunk
if (!currentChunk.empty()) {
    std::cout << "Processing final chunk" << (chunkCount + 1)
        << " (" << currentChunk.size() << " integers)...\n";

    sortAndWriteChunk(currentChunk, chunkCount);
    chunkFiles.push_back(getTempFilename(chunkCount));
    chunkCount++;
}
input.close();
```

6.4.4. Heap Sort for Chunks (Heap).

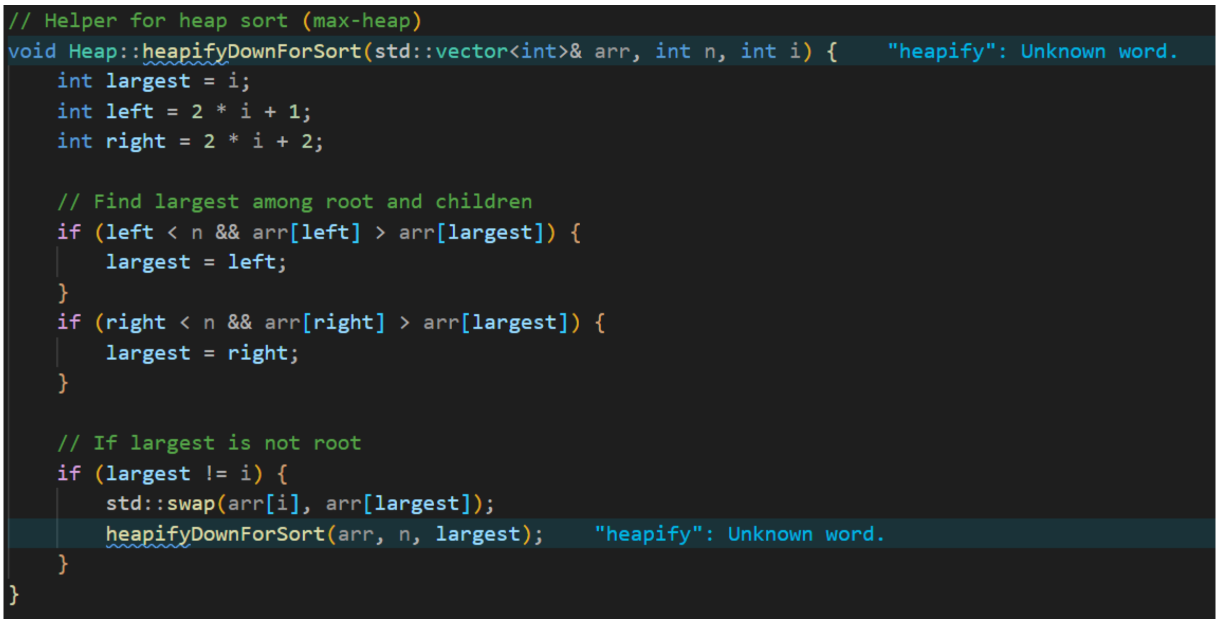
Heap Sort is manually implemented to sort each chunk in RAM, guaranteeing a time complexity of O(M log M) per chunk, where *M* is the chunk size.



```cpp
// Static heap sort function (ascending order)
void Heap::heapSort(std::vector<int>& arr) {
    int n = static_cast<int>(arr.size());

    // Build max-heap (for ascending sort)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapifyDownForSort(arr, n, i);    "heapify": Unknown word.
    }

    // Extract elements one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root (maximum) to end
        std::swap(arr[0], arr[i]);

        // Heapify reduced heap    "Heapify": Unknown word.
        heapifyDownForSort(arr, i, 0);    "heapify": Unknown word.
    }
}
```



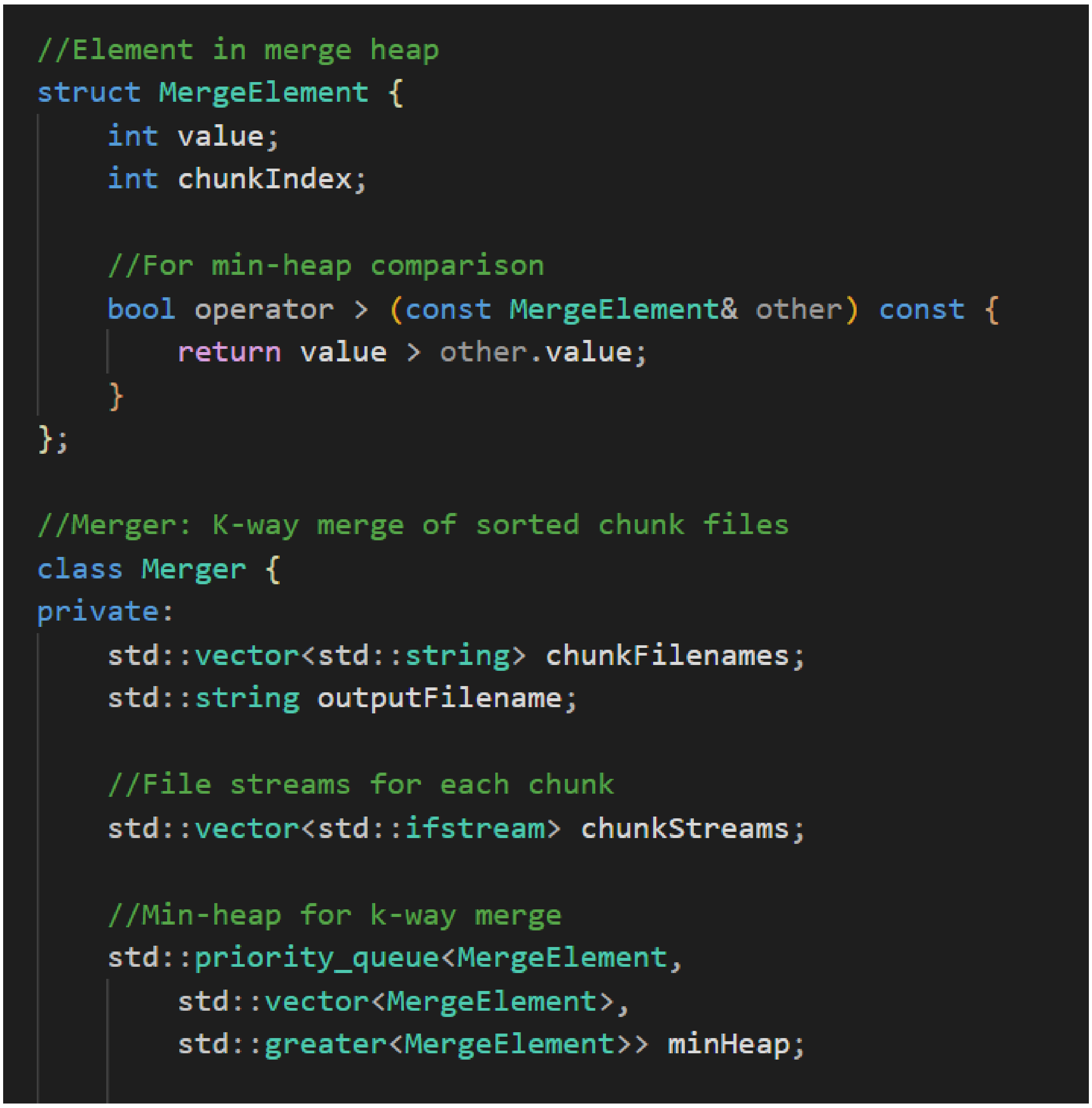
```
// Static heap sort function (ascending order)
void Heap::heapSort(std::vector<int>& arr) {
    int n = static_cast<int>(arr.size());

    // Build max-heap (for ascending sort)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapifyDownForSort(arr, n, i);    "heapify": Unknown word.
    }

    // Extract elements one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root (maximum) to end
        std::swap(arr[0], arr[i]);

        // Heapify reduced heap    "Heapify": Unknown word.
        heapifyDownForSort(arr, i, 0);    "heapify": Unknown word.
    }
}
```



```
// Helper for heap sort (max-heap)
void Heap::heapifyDownForSort(std::vector<int>& arr, int n, int i) {    "heapify": Unknown word.
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // Find largest among root and children
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If largest is not root
    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapifyDownForSort(arr, n, largest);    "heapify": Unknown word.
    }
}
```

6.4.5. Min-Heap for K-way Merge (Merger).

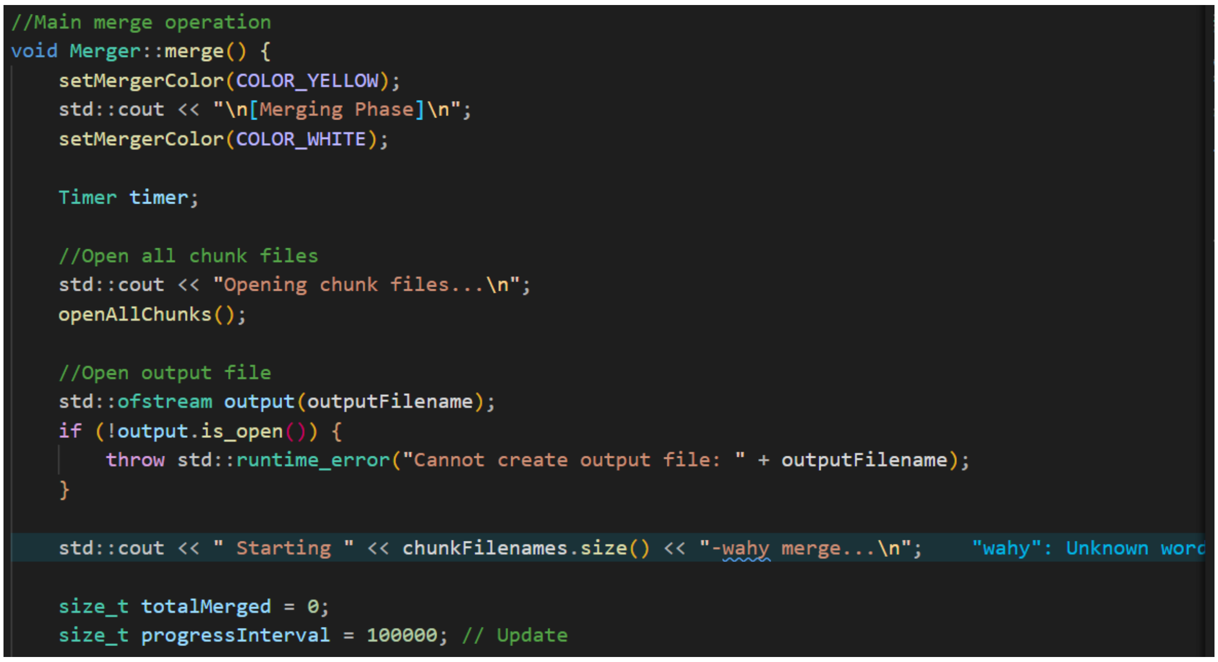
The Merger module uses std::priority_queue with a custom comparator to create a min-heap. Each heap entry stores both the value and its source run, enabling fast selection of the smallest available element across all runs.



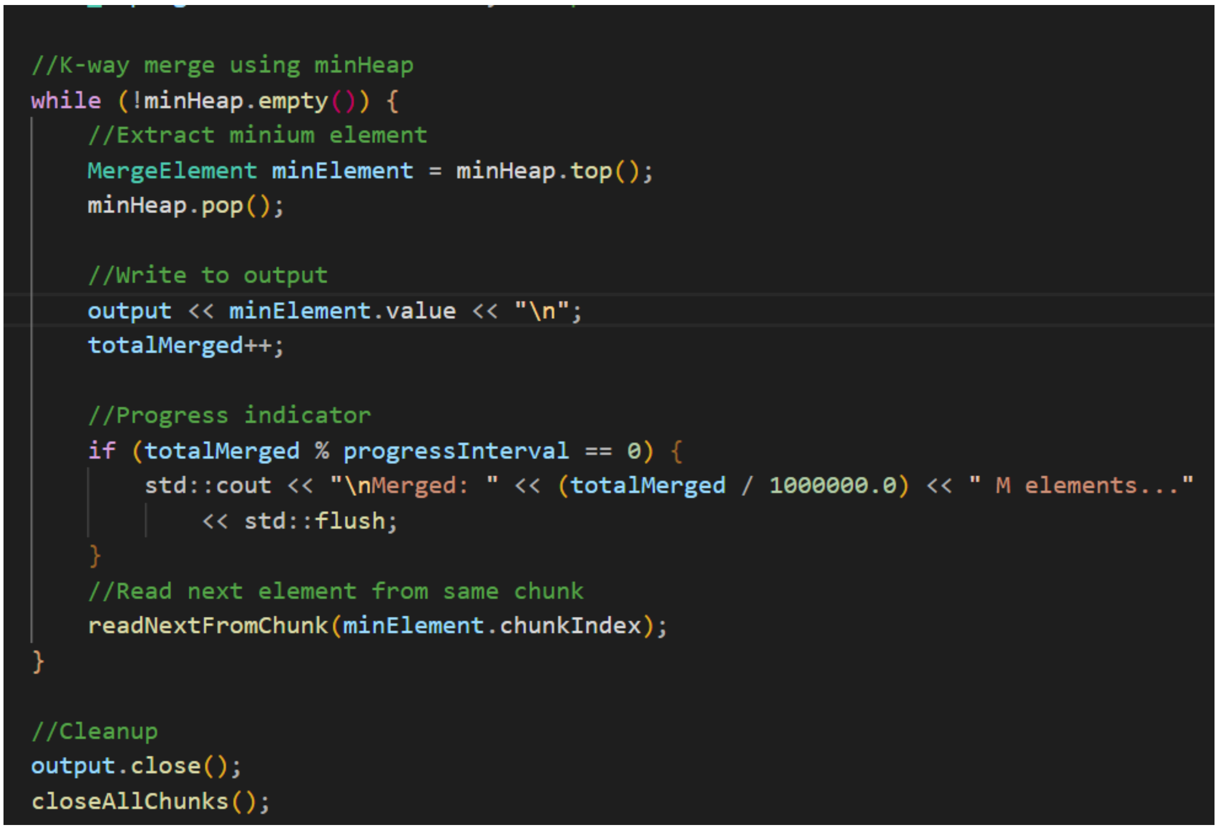
```cpp
//Element in merge heap
struct MergeElement {
    int value;
    int chunkIndex;

    //For min-heap comparison
    bool operator > (const MergeElement& other) const {
        return value > other.value;
    }
};

//Merger: K-way merge of sorted chunk files
class Merger {
private:
    std::vector<std::string> chunkFilenames;
    std::string outputFilename;

    //File streams for each chunk
    std::vector<std::ifstream> chunkStreams;

    //Min-heap for k-way merge
    std::priority_queue<MergeElement,
        std::vector<MergeElement>,
        std::greater<MergeElement>> minHeap;
```

<u>*6.4.6. Merge Loop (Merger::merge).*</u>

While the heap is not empty, the system repeatedly pops the smallest element, writes it to the output file, then reads the next element from the corresponding run and pushes it back into the heap if more data remains.



```
//Main merge operation
void Merger::merge() {
    setMergerColor(COLOR_YELLOW);
    std::cout << "\n[Merging Phase]\n";
    setMergerColor(COLOR_WHITE);

    Timer timer;

    //Open all chunk files
    std::cout << "Opening chunk files...\n";
    openAllChunks();

    //Open output file
    std::ofstream output(outputFilename);
    if (!output.is_open()) {
        throw std::runtime_error("Cannot create output file: " + outputFilename);
    }

    std::cout << " Starting " << chunkFilenames.size() << "-wahy merge...\n";    "wahy": Unknown worc

    size_t totalMerged = 0;
    size_t progressInterval = 100000; // Update
```



```
//K-way merge using minHeap
while (!minHeap.empty()) {
    //Extract minium element
    MergeElement minElement = minHeap.top();
    minHeap.pop();

    //Write to output
    output << minElement.value << "\n";
    totalMerged++;

    //Progress indicator
    if (totalMerged % progressInterval == 0) {
        std::cout << "\nMerged: " << (totalMerged / 1000000.0) << " M elements..."
            << std::flush;
    }
    //Read next element from same chunk
    readNextFromChunk(minElement.chunkIndex);
}

//Cleanup
output.close();
closeAllChunks();
```

6.5. Resource Management.

The system applies modern C++ techniques to ensure safety and stability:

+ **RAII (Resource Acquisition Is Initialization):** Management of std::ifstream and std::ofstream objects relies on object lifetimes, ensuring that file streams are properly closed even if errors occur.

+ **System Cleanup:** The cleanupTempFiles function in the Chunker class is invoked immediately after the merge phase to delete intermediate run files, preventing unnecessary disk usage after task completion.

+ **Status Reporting:** The system uses different console colors to indicate the progress of each stage (Chunking, Merging, Cleanup), making it easier for users to monitor execution on large datasets.

PART 7: APPLICATIONS.

7.1. Real-World Problems Addressed by the System.

In practice, many systems must process very large data files stored on disk, such as system logs, data extracted from databases, or daily/monthly transaction records. These files often exceed available RAM capacity, making traditional in-memory sorting (which requires loading the entire dataset into memory) difficult to apply or unstable in real deployments.

The Big Data File Sorter addresses this exact need through external sorting: the data is divided into memory-sized chunks, each chunk is sorted independently, and the intermediate results are merged into a final sorted output. As a result, the system remains stable even when processing datasets larger than RAM and can be used as a preprocessing module for many data-intensive applications.

7.2. Log Processing.

A typical application of the project is sorting log files by timestamp, event code, or priority level. In distributed environments, logs are often collected from multiple sources and arrive out of order.

For incident tracing, once logs are sorted chronologically, DevOps engineers can easily apply binary search or linear scans to pinpoint the exact moment an error occurred.
For behavior analysis, ordered data allows downstream analytics pipelines, such as anomaly detection, to operate more accurately by analyzing sequences of events in their true execution order.

7.3. Big Data Preprocessing and ETL.

In data pipelines (ETL – Extract, Transform, Load), sorting plays a critical role in preparing data for more advanced processing stages:

+ For deduplication, after sorting, duplicate records appear consecutively, allowing them to be removed with a single linear pass in O(n) time.

+ For join optimization, sorted data enables sort-merge join, one of the most efficient join algorithms for large datasets.

+ For data partitioning, sorting by key facilitates dividing data into buckets, which is essential for parallel processing on distributed computing clusters.

7.4. Database-Related Use Cases.

In database systems, external sorting is widely used in core operations such as sort-merge joins, index construction, and ORDER BY execution when data cannot fit entirely in memory. This project mirrors those principles at the file level: data is read from disk, sorted into runs, and merged into a final result.

Therefore, the system can be viewed as a simplified model that illustrates how a DBMS handles sort-intensive queries on large datasets, effectively bridging theoretical concepts with real-world database behavior.

7.5. Scalability and Integration Potential.

The system's modular architecture provides a solid foundation for future extensions:

+ **Parallelization:** Leveraging multi-core CPUs to sort multiple chunks concurrently can significantly reduce Phase 1 execution time.

+ Complex Records: Extending the system from sorting integers to handling multi-field records (structs or objects) with customizable comparison functions.

+ Multi-pass Merge Optimization: When the number of run files exceeds the operating system's file handle limits, hierarchical or multi-level merge strategies can be applied to support petabyte-scale data.

+ Data Compression: Integrating compression techniques for temporary run files to reduce disk usage and improve effective I/O throughput.

PART 8: LIMITATIONS & FUTURE WORK.

8.1. Limitations

The current system successfully meets the goal of external sorting for datasets larger than RAM; however, several limitations remain when scaling to very large inputs or real-world environments.

First, the input format is currently assumed to be simple (one integer per line). For real-world data such as logs or multi-field records, the system would require additional parsing, key-based comparators, and more robust handling of malformed input.

Second, the number of run files can grow rapidly when the chunk size is small. If the number of runs becomes too large, opening many files simultaneously during the merge phase may hit the operating system's file handle limits. In such cases, a multi-pass merge strategy or an explicit upper bound on k (for k-way merging) becomes necessary.

Third, performance is still heavily influenced by disk I/O. If buffering is not sufficiently optimized or if the system performs many fine-grained I/O operations, throughput can degrade significantly as input size increases. In addition, using text files is convenient for demonstration, but parsing text introduces extra CPU overhead compared to binary formats.

Finally, the system follows a single-machine model. For extremely large datasets or near real-time requirements, this approach would need distributed processing or stronger parallelism to increase throughput.

8.2. Future Work.

To evolve the Big Data File Sorter into a more practical and powerful tool, the following improvements are proposed:

+ **I/O optimization and data format transition**: Implement block-based buffered I/O (read/write in blocks rather than line-by-line) to better leverage operating system prefetching and reduce system-call overhead.
Add support for binary formats to reduce disk footprint and eliminate text parsing overhead.

+ **Multi-pass merge strategy**: Develop hierarchical merging: if the number of run files exceeds the allowed limit K, the system first merges runs into larger intermediate files, then performs a final merge pass. This ensures stable execution regardless of dataset size and OS file-handle constraints.

+ **Parallel run generation**: Since chunks are independent, multi-threading can be introduced to sort multiple chunks concurrently across CPU cores. This can significantly reduce Phase 1 time when I/O is not the only bottleneck.

+ **Support for complex records and sort keys**: Extend the system to handle structured records and allow users to define sorting keys (e.g., sorting logs by timestamp, then by severity). This would make the tool more applicable to real-world datasets.

+ **Monitoring and advanced reporting**: Add detailed logging and reporting features, including the number of generated runs, disk pass count, peak RAM usage, and per-phase timing breakdowns. These metrics would make benchmarking more professional and the performance evaluation more convincing.

PART 9: CONCLUSION.

The Big Data File Sorter project successfully delivers a complete C++ system that addresses the challenge of sorting datasets whose sizes exceed physical RAM capacity. By applying the External Merge Sort model, the system demonstrates both efficiency and stability when processing large-scale data stored on standard disk-based storage.

9.1. Summary of Achievements.

The system fully implements an external sorting pipeline with two core phases:

+ Run Generation Phase: The input data is partitioned into memory-sized chunks, and each chunk is internally sorted using a custom Heap Sort implementation developed by the team.

+ Merge Phase: The sorted run files are merged using a k-way merge strategy powered by a Min-Heap structure (std::priority_queue with a custom comparator).

This design not only enables stable execution on machines with limited resources but also clearly demonstrates the integration of key DSA topics, including heap structures, priority queues, advanced sorting techniques, and asymptotic complexity analysis in a real-world context.

In addition to performance considerations, the project places strong emphasis on memory stability. Regardless of how large the input file becomes, the program maintains a bounded RAM usage defined by the configured memory budget.

9.2. Project Value and Completeness.

Beyond its core sorting functionality, the project adds value through a comprehensive set of supporting tools:

+ Correctness: Output correctness is strictly verified using a dedicated verification tool that confirms the sorted order.

+ Supporting ecosystem: The repository includes a random data generator and a benchmarking utility to facilitate experimentation and empirical evaluation.

+ Application potential: The system can serve as a practical preprocessing module for use cases such as log processing, ETL pipelines, and as a simplified illustration of how DBMS components handle disk-based sorting.

9.3. Vision and Future Directions.

Although the project meets its primary objectives, there remains significant room for enhancement to better align with industrial-grade data processing tools:

+ I/O performance optimization: Transitioning to binary formats to eliminate text parsing overhead and reduce disk usage.

+ Extreme-scale handling: Implementing multi-pass merge to avoid operating system file-handle limitations when the number of runs becomes very large.

+ Architectural modernization: Parallelizing run generation via multi-threading to leverage multi-core hardware more effectively.

+ Broader data support: Extending the system to sort complex multi-field records with customizable sorting keys.

Overall, the project goes beyond a purely academic exercise and serves as a practical foundation for understanding real big-data systems, where balancing CPU, RAM, and I/O is the key to achieving scalable performance.

LLM USAGE (AI ASSISTANCE).

During the preparation of the Performance Analysis section and the design of the Benchmark tables, the team consulted the LLM tool claude.ai for suggestions on report presentation, the selection of measurement metrics (e.g., dataset size, chunk size, number of runs, run generation/merge time, total runtime), and ways to summarize experimental results.

All C++ source code was implemented by the team, and all experiments/measurements were executed by the team on the project's datasets. The LLM was used only as a reference for structuring and presenting the report content, not for generating the implementation or producing benchmark results.

REFERENCES.

+ **dsagroup62025.** *DSA_PROJECT_SEMESTER1 – Big Data File Sorter (Source code)*. GitHub repository. Truy cập tại: https://github.com/dsagroup62025/DSA_PROJECT_SEMESTER1.git

+ **dsagroup62025.** *README.md – Build & Run Instructions, Project Structure, Tools (generate/verify/benchmark)*. Trong repository: **DSA_PROJECT_SEMESTER1**. Truy cập tại: https://github.com/dsagroup62025/DSA_PROJECT_SEMESTER1.git

+ **dsagroup62025.** *Input Data (input.txt / input_100mb.txt / input_1gb.txt) – Dataset for External Sorting Demo & Benchmark.* Thư mục **data**/ trong repository: **DSA_PROJECT_SEMESTER1**.

+ **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). *Introduction to Algorithms (3rd ed.)*. MIT Press. (Tham khảo lý thuyết về Heap Sort và phân tích độ phức tạp).

+ **Knuth, D. E.** (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley. (Tham khảo nguyên lý về External Sorting và mô hình quản lý bộ nhớ ngoài).

+ **Học phần DSA.** (2025). *Tài liệu bài giảng Cấu trúc dữ liệu và Giải thuật (Day 1 - Day 11)*. Trường Đại học Công nghệ Thông tin.

+ **GeeksforGeeks.** *External Sorting / External Merge Sort*. Tham khảo kỹ thuật trộn đa đường (K-way merge).

+ **cppreference.com.** *std::priority_queue (C++ reference)*. Đặc tả kỹ thuật về hàng đợi ưu tiên trong thư viện chuẩn C++.

+ **Microsoft Learn.** *File Handling in C++ (fstream)*. Tham khảo tối ưu hóa luồng vào/ra cho dữ liệu kích thước lớn.