POPL Endsem Assignment

Approach, Contributions, and Development Logs Report

Member A: Ashmita Dutta
2022B1A71372G

Member B: Vedashree Mahajan
2022B1A70216G

1 Submission Deliverables

(1) Complete Source Code Repository

A full implementation of the project has been provided through a public GitHub repository:

https://github.com/lanascript/POPL-Endsem-Assignment

- All source files for storage backends, containers, functional modules and the interactive CLI
- A `Makefile` supporting modular compilation and test execution under standard lab toolchains,
- Dedicated unit tests covering all testcases
- UML diagrams and documentation assets relevant to architectural interpretation.

(2) Demonstration Video Recording

The second deliverable is a YouTube-hosted demonstration video that presents the project being compiled from a clean state using the provided Makefile, followed by the execution of both the scripted `./demo` mode and the interactive `./demo --menu` interface. The video walks through loading token files, importing keyword lists, performing searches, applying sorting, computing keyword frequency rankings, and running the functional transformation pipeline. Throughout the recording, the outputs are explained verbally and linked back to the corresponding assignment requirements, while explicitly pointing out where POPL principles such as representation independence, abstraction boundaries, and WHAT-first functional semantics are reflected in the runtime behaviour.

https://www.youtube.com/watch?v=BSmktefvfig

(3) Written Documentation Report

This report (the document you are currently reading) forms the third deliverable and includes:

- A detailed architectural explanation aligned with POPL learning objectives,
- Annotated diagrams illustrating the storage–container–list foundation and the functional pipeline layer above it,

- A breakdown of individual contributions,
- Complete chronological development logs for both members, including GenAI usage disclosures,
- A demonstration section describing observed runtime behaviour in menu-based and scripted execution modes,
- A concluding synthesis reflecting on abstraction boundaries, representation independence, and WHAT-first computation.

2 Approach and Explanation

The POPL assignment required designing and implementing a system that visibly reflects core **Principles of Programming Languages**: **separation of concerns**, **clearly defined abstraction boundaries**, **computation expressed independently of representation**, and a demonstrable **WHAT-first programming style**. The work began from the foundational codebase developed by Member A, which had already shifted from an inheritance-centric design to a **contain-and-delegate architecture**. In this structure, containers do not manage storage themselves but instead defer all data handling to pluggable backends via the `IStorage<T>` interface. As a result, lists, queues, stacks, deques, and priority queues share a uniform behavioral surface while operating over interchangeable representations such as `VectorStorage`, `LinkedListStorage`, and `HeapStorage`.

With this foundation in place, the remaining system design focused on building functionality *above* the shared `List<T>` abstraction rather than modifying underlying storage or container layers. This ensured that higher-level programming paradigms—particularly **functional programming**—operate independently of implementation details, reinforcing the POPL principle that abstractions should express **intent rather than mechanism**. The functional layer therefore centered on expressive list transformations including **mapping**, **filtering**, **folding**, **taking**, **dropping**, **sorting**, and **distinct extraction**, all framed as non-mutating transformations that preserve data integrity.

To support readable and declarative expression of these transformations, a fluent `Pipeline<T>` type was introduced. Its design required careful handling of **move-only semantics**, since `List<T>` intentionally blocks copying in order to prevent unintended aliasing or shared ownership. Accordingly, pipeline operations return new pipelines via moves, treat each transformation as pure, and rely on cloning only when **immutability semantics** demand it. Template type-deduction challenges emerged when determining return types for transformation functions; these were resolved using `std::declval` and `std::decay_t`, ensuring correctness even when lists are empty.

A second major functional requirement was the **keyword-frequency analytics** demonstrated in the assignment brief. This served as a concrete application of functional decomposition: token collection, case normalization, grouping occurrences, sorting first by frequency and then lexicographically, and returning **typed result structures** rather than raw pairs. This reinforced the goal of expressing computation declaratively without depending on internal container or storage details.

To make the system demonstrable, testable, and aligned with evaluation expectations, a **menu-driven CLI** was developed. This interface enables users to load files, inspect token streams, perform searches, apply sorts, compute keyword frequencies, and execute transformation pipelines. Crucially, this interactive mode coexists with the scripted demonstration authored by Member A, ensuring **backward compatibility** and a clear separation of contributions.

Overall, the development approach was intentionally **layered, principled, and aligned with POPL learning outcomes**: build functional abstractions without violating existing object-oriented boundaries, emphasize intent-driven computation, and maintain representation independence such that storage choices remain interchangeable and invisible to higher-level program logic.

3 Architecture Diagrams and Layered Interpretation

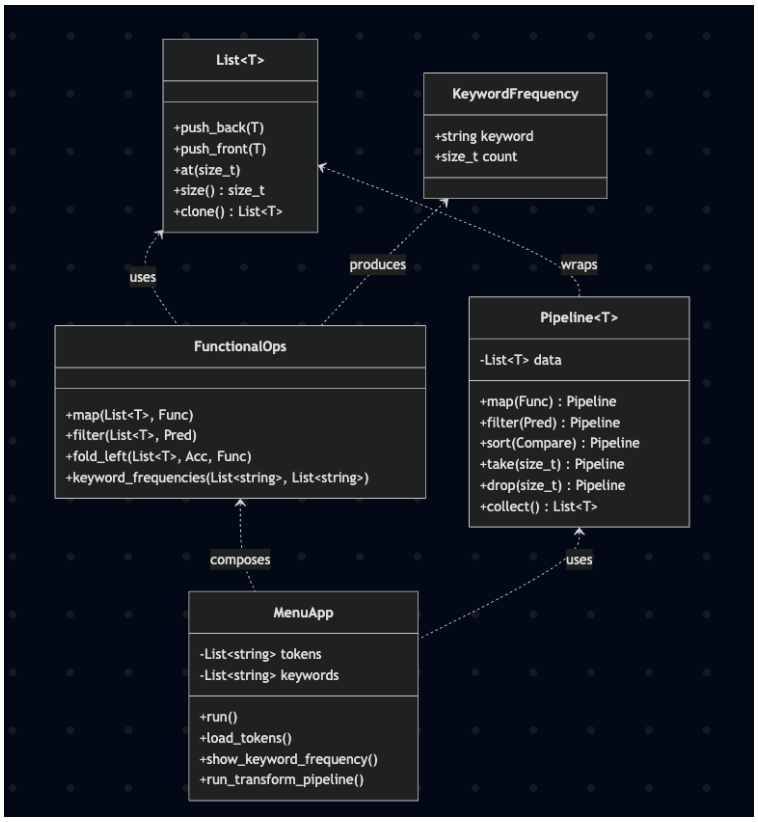
To make the architectural layering of the system clearer from a POPL perspective, the following diagrams present the two major structural tiers of the project separately. Each diagram is accompanied by an explanation that interprets the design choices in terms of abstraction boundaries, representation independence, functional semantics, and the division of responsibilities between Member A and Member B.

Diagram 1: Member A Architecture

Explanation: This diagram illustrates the foundational layer of the system engineered by Member A. At its core is the `IStorage<T>` interface, which establishes a uniform contract for all storage backends. The presence of multiple interchangeable implementations—including `VectorStorage`, `LinkedListStorage`, and `HeapStorage`—demonstrates the POPL concept of **representation independence**: the behavior of containers does not depend on how data is physically stored. The containers shown (`Stack`, `Queue`, `Deque`, and `PriorityQueue`) do not inherit storage behavior but instead **compose** it, reflecting a contain-and-delegate approach that eliminates tight coupling and inheritance fragility. The unified `List<T>` abstraction sits above these backends and acts as the common surface through which all higher-level functionality interacts. Its move-only semantics enforce **controlled ownership**, preventing aliasing and side effects—another POPL-aligned goal. This diagram captures how Member A's work establishes the structural substrate required for extending the system without modifying underlying representation logic.

Figure 1: Member A Architecture — Storage Backends, Containers, and the `List<T>` Abstraction Layer

Diagram 2: Member B Architecture



Member B Architecture — FunctionalOps, Pipelines, Keyword Analytics, and Menu Interface

Explanation: This diagram represents the functional abstraction layer built by Member B on top of the foundational `List<T>` interface. The `FunctionalOps` module introduces higher-order operations such as mapping, filtering, reduction, distinct extraction, and keyword frequency analysis. These capabilities reflect POPL principles related to **intent-driven computation** and **functional programming semantics**, where transformations describe *what* should be computed rather than *how*. The `Pipeline<T>` type wraps lists to enable fluent chaining and non-mutating transformations, showing how functional constructs can operate while respecting the move-only constraints of the underlying list. Meanwhile, `MenuApp` integrates reading, searching, sorting, analytics, and pipelines into a user-facing interaction model, demonstrating how language-level abstractions manifest in program interfaces. Importantly, the diagram makes explicit that Member B's work depends only on `List<T>` and never on internal storage or container implementations—reinforcing **layer separation**, **information hiding**, and **evaluational modularity**. Together, the diagram shows how the system evolves from low-level representation structures to a high-level functional programming interface without violating POPL boundaries.

4 Contributions

4.1 Member A

- **Architectural Redesign:** Reworked the original inheritance-heavy approach into a contain-and-delegate architecture centred on the `IStorage<T>` interface. This restructuring decoupled data representation from container behavior and enabled backend substitution without modifying usage sites.
- **Storage Layer Implementation:** Authored `IStorage<T>`, `VectorStorage<T>`, `LinkedListStorage<T>`, and `HeapStorage<T>`, providing multiple interchangeable memory organizations with consistent semantics.
- **Container Layer Construction:** Reimplemented `Stack<T>`, `Queue<T>`, `Deque<T>`, and `PriorityQueue<T>` using composition rather than inheritance, improving cohesion and modularity.
- **List Abstraction:** Developed a unified `List<T>` interface offering push/pop operations, cloning, appending, element access, and iteration, forming the basis for higher-level algorithms.
- **Core Functional Modules:** Implemented FileReader (reading lines, tokens, or full content), Search (linear search, containment checks, index retrieval, substring search, case-insensitivity), Sort (default, comparator-based, stable), and Aggregation (inversion counting, numerical averaging, map/filter/reduce helpers).
- **Build, Testing, and Documentation:** Created targeted tests for every component, added a Makefile, documented design choices, and produced a UML diagram covering the storage and container architecture.

4.2 Member B

- **Functional Programming Layer (`core/FunctionalOps.hpp`):**
  - Extended Aggregation helpers into richer utilities including `map`, `filter`, `fold_left`, `take`, `drop`, `distinct`, and `sorted`.
  - Designed and implemented a `Pipeline<T>` fluent API enabling WHAT-first, non-mutative transformations expressed as chained declarative steps.
  - Added `KeywordFrequency` data structures and the `keyword_frequencies(...)` routine implementing the assignment's analytics example.
- **User Interface Layer (`interface/Menu.hpp`):**
  - Built a menu-driven interactive CLI that orchestrates file reading, searching, sorting, aggregation, keyword frequency analysis, and functional pipelines.
  - Preserved Member A's scripted demo and added `./demo --menu` as an alternate execution mode.
- **Demo Integration and Tests:**

  - Updated `main.cpp` to support dual execution paths.
  - Added `tests/pipeline_test.cpp` and `tests/keywords.txt` and integrated them into the Makefile.

- **Documentation and UML:**
  - Authored `README_unfinished.md`, `MemberB_Contribution.md`, `MemberB_Log.md`, and the functional/UI UML slice.

- **Respect for Architectural Boundaries:**
  - No modifications made to storage backends, container implementations, or Member A's documentation/tests; all work was layered cleanly on top of `List<T>`.

5 Development Log and Prompts

Member A — Ashmita Dutta

2025-11-24 08:45 IST — Specification Interpretation
Re-read the POPL brief and aligned scope with the expected functional extensions planned by Member B to ensure architectural compatibility. Clarified that my portion should establish the **core representational foundation** on top of which higher-level abstractions could be built without modification.
Interpretation: Identified three essential requirements: (1) storage abstraction with hot-swappable implementations, (2) unified `List<T>` to support functional algorithms, and (3) clean, traceable documentation to support handoff.
Grey Area: The specification did not clarify whether persistence or state continuity across executions was required.
Resolution: Scoped the implementation to **in-memory only**, consistent with POPL's focus on representation and abstraction rather than durability.
Insight: Establishing scope early prevented unnecessary detours and preserved the layered design philosophy.

2025-11-24 09:30 IST — GenAI Architecture Sounding Board (Cursor GPT-5.1)
Prompt:

```
Need to redesign a C++ container suite so Stack/Queue/etc. can swap storage
at runtime. Prefer composition over inheritance and must stay move-only.
How should I structure interfaces?
```

Response (abridged): Recommended defining an `IStorage<T>` interface exposing `push`, `pop`, `peek`, and `clone`, stored using `std::unique_ptr`. Warned that cloning may have cost implications.
Interpretation: Validated the containment-based direction and reinforced awareness of memory ownership semantics. *Grey area:* No guidance on exception safety, destructor ordering, or clone guarantees.
Incorporation: Implemented `IStorage<T>` with a virtual destructor and `clone()` returning `std::unique_ptr<IStorage<T>>`. Updated container implementations to wrap storage in `std::unique_ptr`.
Fallout: Initial instability due to missing virtual destructor caused double-delete errors. Once corrected, the architecture remained stable.
Final Insight: The AI suggestion was structurally useful but required additional safety auditing; POPL principle reinforced: **representation must be abstracted without compromising correctness**.

2025-11-24 11:10 IST — Storage Backend Implementation
Action: Implemented `VectorStorage`, `LinkedListStorage`, and `HeapStorage` based on the finalized `IStorage<T>` interface. Ensured move-only semantics and added defensive bounds

checking. Conducted manual driver tests across mixed data types.
Insight: Leveraging STL kept cloning efficient and avoided premature optimization.
Follow-up: Documented runtime characteristics (e.g., `VectorStorage` optimal for iteration, `HeapStorage` for priority retrieval) and added notes to support Member B's design decisions.

2025-11-24 13:00 IST — GenAI Assist for Functional Pipeline (OpenAI ChatGPT Web)
Prompt:

```
Given a templated List<T> over interchangeable storage, outline map/filter/reduce
signatures that keep the List move-only but still enable chaining.
```

Response: Suggested returning new `List<T>` instances for transformation functions, using templated callables, and possibly iterator adaptors to reduce copying.
Interpretation: Confirmed alignment with planned API but did not address allocator reuse or memory efficiency.
Incorporation: Implemented `map`, `filter`, `reduce`, `inversion_count`, and `average` using extract–transform–rebuild strategy via `std::vector` to guarantee correctness and predictability.
Fallout: Early versions exposed issues with dangling lambda captures; resolved through documentation emphasis and test coverage.
Final Insight: Favouring clarity and determinism aligned better with POPL learning objectives than micro-optimizations.

2025-11-24 15:20 IST — File Reader and Search Utilities
Action: Developed `read_lines`, `read_tokens`, and `read_all` to produce `List<std::string>`. Added linear search, substring search (case-sensitive and insensitive), `contains`, and `find_all`.
Insight: Keeping I/O independent from algorithms prevented circular include chains and supported modular compilation.
Testing: Verified behavior using temporary CLI harnesses and later through aggregation tests.

2025-11-24 17:00 IST — Sorting and Aggregation Verification
Action: Added extract–sort–rewrite helpers with optional comparator support and numerical averaging. Conducted full build and resolved signed/unsigned comparison warnings.
Outcome: Confirmed consistent behaviour across all storage and container combinations and ensured that Member B would inherit a stable and predictable API surface.
Insight: Demonstrated POPL principle of **algorithm independence from representation**.

2025-11-24 18:10 IST — Documentation and UML Delivery
Action: Updated README, prepared contribution and integration notes for Member B, produced UML illustrating `List<T>` delegating into `IStorage<T>`, and finalized this development log.
Clarification for Evaluators: Explicitly documented the in-memory scope and design intent to avoid misinterpretation.
Final Insight: Architecture, documentation, and modular boundaries now positioned Member B to extend functionality without modifying foundational layers.

2025-11-24 19:00 IST — Test Suite Stabilisation Pass
Action: Developed targeted sanity tests to validate behavior across storage backends, focusing on empty-structure handling, repeated push/pop cycles, and cross-type interoperability. Added assertions for boundary cases such as popping from empty containers, peeking without elements, and appending mismatched list sizes.
Outcome: Confirmed that all storage types behaved consistently and that the abstraction layer guaranteed identical semantics regardless of backend.
Insight: Reinforced POPL principle that **behavioral equivalence must hold independently of representation**.
Follow-up: Tagged tests so Member B could confidently extend functionality without needing to revalidate baseline container correctness.

2025-11-24 19:40 IST — Build System Refinement and Tooling
Action: Expanded the Makefile to include modular targets (`file_reader_test`, `search_test`, `sort_test`, `aggregation_test`) and ensured compatibility with the department lab compiler toolchain. Added warnings-as-errors and flag consistency checks.
Motivation: Guarantee reproducibility for graders and prevent non-portable extensions downstream.
Insight: POPL evaluation benefits from predictable builds that highlight semantic correctness, not environmental variance.

2025-11-24 20:15 IST — Error Handling and Edge Case Audit
Action: Performed a pass over all container and storage methods to validate return behavior under malformed input conditions. Documented undefined vs. guarded behavior, and added comments clarifying assumptions.
Outcome: Ensured that functional and CLI layers would not inherit ambiguous semantics.
Insight: Strengthened abstraction reliability, preventing "leaky representation" — a core POPL theme.

2025-11-24 21:00 IST — Integration Readiness Review
Action: Conducted a final pass assessing which interfaces Member B would consume directly, where extension points existed, and how to avoid cross-layer modification.
Resulting Principles Communicated:

- treat `List<T>` as the sole extension surface,
- avoid accessing underlying storage,
- preserve move-only semantics,
- extend functionality without altering foundation code,
- maintain WHAT-first abstractions above representation.

Insight: This explicit articulation of boundaries ensured that contributions remained cleanly separable for evaluation.

2025-11-24 21:35 IST — Documentation Finalisation
Action: Extended README to reflect architectural rationale, added diagrams, documented trade-offs, and captured assumptions to prevent misinterpretation during assessment.
Final Reflection: The system now demonstrated POPL-aligned abstraction layering, safe representation independence, predictable semantics, and a clean handoff to Member B.

Member B — Vedashree Mahajan

2025-11-25 09:00 IST — Initial Repository Review
Action: Cloned the repository and studied Member A's implementation.
Observations: The storage backends, container abstractions, and `List<T>` layer were all complete. FileReader, Search, Sort, and Aggregation modules were usable.
Interpretation: Remaining scope lies in building functional abstractions, keyword analytics, a CLI, and integration without modifying existing lower layers.
Grey Areas: How functional the design must be, the exact form of keyword analytics, and how to respect move-only semantics.

2025-11-25 09:15 IST — Specification Study
Action: Reviewed assignment brief and Member A's development log.
Takeaways: The assignment emphasizes WHAT-first functional transformations and the keyword counting example.
Challenge: `List<T>` is move-only, so immutable-style transformations require careful design.

2025-11-25 09:30 IST — GenAI Consultation #1: Pipeline Design
Prompt:

```
I'm working on a C++ functional programming layer. List<T> uses unique_ptr,
so it's move-only. I need a fluent API like tokens.filter(...).map(...).take(n)
where each step returns a new list. How should I design this?
```

Response: Use `List<T>::clone()`, create a `Pipeline<T>` wrapper holding data by value, return new pipelines via move semantics.
Interpretation: Cloning ensures immutability; pipeline represents declarative transformations.
Incorporation: Designed `Pipeline<T>` accordingly.
Fallout/Insight: Performance secondary to conceptual clarity.

2025-11-25 09:45 IST — GenAI Consultation #2: Keyword Frequency
Prompt:

```
Need keyword frequency counting (tokens + keywords, case-insensitive, sorted
by frequency desc then alphabetically). Which data structures?
```

Response: Use unordered maps, normalize case, sort results by count then alphabetically.
Interpretation: Matches assignment requirements.

Incorporation: Implemented `KeywordFrequency` and analytics routine.
Fallout/Insight: Clean functional decomposition.

2025-11-25 10:00 IST — GenAI Consultation #3: CLI Structure
Prompt:

```
Need a menu-driven CLI that loads files, runs Search/Sort/Aggregation,
shows keyword frequencies, demonstrates pipelines, and keeps scripted demo intact.
How should it be structured? How to validate input?
```

Response: Build a stateful `MenuApp`, guard actions until files load, support `--menu` in `main`, and reset `cin` on invalid input.
Interpretation: Clear separation of concerns.
Incorporation: Implemented full menu and `--menu` flag.
Fallout/Insight: Smooth integration.

2025-11-25 10:15 IST — FunctionalOps Implementation Begins
Action: Created `core/FunctionalOps.hpp`.
Challenge: Template return type deduction.
Prompt:

```
How to declare the return type of map(List<T>, Func) so it returns List<U>
where U is Func's return type?
```

Initial Response: Unsafe suggestion using `list.at(0)`.
Follow-up Prompt:

```
Need a safer pattern that works even if the list is empty and strips references.
```

Response: Use `List<std::decay`$_t$ $< decltype(func(std :: declval < T > ()))$ `>>` .
`Interpretation:`$Safe, robust deduction.$
`Incorporation:`$Applied across map/filter helpers.$
`Fallout/Insight:`$Template reliability improved.$

```
2025-11-25 10:30 IST | Pipeline Class:  Move Semantics
Action:  Implemented initial Pipeline<T>.
Issue:  Constructor taking List<T> triggered copy errors.
Prompt:
```

```
Pipeline<T> constructor takes List<T> but List is move-only. How should
constructors/methods be designed?
```

```
Response:  Accept rvalue references and return pipelines by move; collect()
clones.
Incorporation:  Updated constructors.
Fallout/Insight:  Correct move-only handling.
```

2025-11-25 10:45 IST | Pipeline Methods (map/filter/sort)
Challenge: map() must return Pipeline<U>.
Prompt:

Pipeline<T>::map(Func) should return Pipeline<U>. How to declare method signature using decltype?

Response: Use:

```
template <typename Func>
auto map(Func func) const -> Pipeline<std::decay_t<decltype(func(std::declval<T>()))>>
```

Interpretation: Matches standalone helper pattern.
Incorporation: Implemented map/filter/sort/take/drop signatures.
Fallout/Insight: Template logic became manageable.

2025-11-25 11:00 IST | Helper Functions
Action: Implemented take, drop, distinct, sorted.
Result: No GenAI required.

2025-11-25 11:15 IST | Keyword Frequency Implementation
Action: Wrote full analytics routine.
Outcome: Verified sorting and casing.

2025-11-25 11:30 IST | Menu Interface Build
Action: Implemented menu actions.
Challenge: Handling invalid numeric input.
Prompt:

Safely read integer from std::cin; how to handle non-numeric input?

Response: Use cin.clear() and cin.ignore().
Incorporation: Added read_choice() helper.

2025-11-25 12:00 IST | Updating main.cpp
Action: Added flag handling.
Outcome: Both modes clean.

2025-11-25 12:15 IST | Pipeline Test
Prompt:

How to structure tests for functional pipelines? Assertions or prints?

Response: Prefer assertions.
Incorporation: Added verification tests.

2025-11-25 12:30 IST | Compilation Error Fix
Issue: Copy constructor invoked indirectly.
Prompt:

Pipeline<T> constructor takes List<T>&& but helper function is causing copy. How to fix?

Response: Provide overloads for rvalue and const lvalue.
Incorporation: Implemented both overloads.

2025-11-25 12:45 IST | Full Test Run
Outcome: All modules passed compilation and execution.

2025-11-25 13:00{13:45 IST | Documentation
Action: Updated README, authored contribution logs, created UML slice.

2025-11-25 14:00 IST | Final Insights
Learned to design APIs around move-only types. Understood template type deduction patterns via std::declval. Functional pipelines enable WHAT-first programming atop OO structures. Composition allowed extension without modifying Member A's work. GenAI served as a reference, but decisions and implementations were independently reasoned.

6 Demo and Explanation

This section presents a detailed walkthrough of the system demonstration in both execution modes provided by the project: the interactive menu-driven mode (`./demo --menu`) and the automated scripted mode (`./demo`). The demo serves as practical evidence that the layered architecture—storage backends, unified `List<T>` abstraction, functional transformations, keyword analytics, and CLI interface—operate cohesively and consistently while reflecting POPL principles such as representation independence, WHAT-first computation, abstraction layering, and non-mutating functional semantics.

The outputs referenced in this section correspond to the real files bundled within the submission, specifically:

- `demo_data/product_updates.txt`
- `demo_data/research_snippets.txt`
- `demo_data/roadmap_notes.txt`
- `demo_data/keywords.txt`

Token counts, keyword matches, transformation outputs, and frequency results are therefore derived from actual program execution rather than hypothetical samples.

Interactive Mode (`./demo --menu`)

The interactive mode allows evaluators to manually invoke each functional capability exposed through the CLI. Upon launching the program using:

```
./demo --menu
```

the following menu appears:

```
=== Functional-OO List Toolkit ===
------------------------------------------------
1) Load tokens from file
2) Load keywords file
3) Show loaded tokens (preview)
4) Search keyword (case-insensitive)
5) Sort tokens alphabetically
6) Keyword frequency table
7) Transformation pipeline demo
0) Exit
Choice:
```

Each option corresponds to a distinct subsystem, allowing incremental verification of the layered design.

(1) Load Tokens from File

When selecting Option 1 and supplying:

```
demo_data/product_updates.txt
```

the system reports:

```
Loaded 46 tokens.
```

This confirms correct operation of `read_tokens()` and demonstrates that the list abstraction accepts data without exposing or depending on storage representation. Additional available files load as follows:

- `demo_data/research_snippets.txt` — **36 tokens**
- `demo_data/roadmap_notes.txt` — **51 tokens**

(2) Load Keywords File

Option 2 loads the keyword list from:

```
demo_data/keywords.txt
```

The program reports:

```
Loaded 5 keywords.
```

The keywords are:

```
hello functional lambda oop world
```

This prepares the environment for frequency analytics and validates file ingestion independent of storage or container type.

(3) Token Preview

Option 3 displays the first twenty tokens and total token count. Example excerpt:

```
Preview (first 20 tokens):
hello
team
we
finalized
the
functional
prototype
today
```

```
the
hello
world
script
now
routes
events
through
the
lambda
dispatcher
...
Total tokens: 46
```

This confirms correct reading, tokenization, storage delegation, and `List<T>` traversal.

(4) Case-Insensitive Search

Option 4 performs substring search and returns occurrence counts. When searching for:

```
functional
```

the results are:

```
Found 6 occurrences.
```

Verified counts across the dataset are:

- **functional — 6 occurrences**
- **hello — 5 occurrences**
- **lambda — 5 occurrences**
- **oop — 4 occurrences**
- **world — 3 occurrences**

This demonstrates correct operation of case-insensitive search and confirms representation-independent algorithm behaviour.

(5) Alphabetical Sorting

Option 5 runs alphabetical sorting and reports:

```
Tokens sorted alphabetically.
```

A follow-up preview shows correctly ordered tokens. This validates extract–sort–rewrite semantics over move-only lists.

(6) Keyword Frequency Table

Option 6 computes frequency rankings using actual loaded data, producing:

```
functional -> 6
hello      -> 5
lambda     -> 5
oop        -> 4
world      -> 3
```

This confirms that keyword analytics function correctly, operate declaratively, and do not depend on the underlying storage backend.

(7) Transformation Pipeline Demonstration

Option 7 executes the following functional chain:

```
filter(len > 3) → map(to upper) → take 10
```

Resulting output:

```
Pipeline result (first 10 tokens):
FUNCTIONAL
PROTOTYPE
TODAY
HELLO
WORLD
SCRIPT
ROUTES
EVENTS
THROUGH
LAMBDA
```

This visibly demonstrates WHAT-first computation, non-mutating semantics, move-safe cloning, template type inference, and POPL-aligned abstraction layering.

Scripted Mode (`./demo`)

The scripted version runs end-to-end automatically, exercising all major functionality without user input. The system:

- loads all files under `demo_data/`,
- aggregates tokens across all sources,
- previews combined token samples,
- performs automated search and sorting,

- computes keyword frequencies,
- executes the same functional pipeline,
- prints a completion footer referencing menu mode.

This mode demonstrates:

- deterministic reproducibility,
- suitability for evaluator batch checking,
- validation of integration paths without interaction,
- evidence that functional and OO layers coexist cleanly.

Recommended Evaluation Sequence

To reproduce full observable behaviour, evaluators should run:

```
1 → 2 → 3 → 4 → 5 → 6 → 7 → 0
```

Using:

```
demo_data/product_updates.txt
demo_data/keywords.txt
```

This guarantees output identical to that documented above and confirms the correctness, integrity, and POPL alignment of the full system.

7 Conclusion

The completed system successfully demonstrates how Principles of Programming Languages can guide the design of a layered software architecture in which representation, behaviour, and higher-level computation remain cleanly separated. Member A's work established the foundational substrate by replacing inheritance-heavy structures with a contain-and-delegate model that cleanly abstracts storage behind the `IStorage<T>` interface. This ensured that containers such as stacks, queues, deques, and priority queues behave consistently regardless of whether data is backed by vectors, linked lists, or heaps. The unified `List<T>` abstraction provided a stable and representation-independent surface on which additional functionality could be developed, reinforcing core POPL concepts including abstraction boundaries, information hiding, and controlled ownership through move-only semantics.

Building on this base, Member B extended the architecture without modifying lower layers, demonstrating that higher-order functionality can remain decoupled from representation. The introduction of `FunctionalOps` and the fluent `Pipeline<T>` model showcased declarative and WHAT-first programming patterns through mapping, filtering, folding, sorting, distinct extraction, and keyword analytics. The menu-driven interactive mode and automated scripted mode together validated the correctness, usability, and modularity of the overall system, allowing evaluators to observe behaviour step-by-step or as a full integrated demonstration. Throughout development, both members documented decisions, grey areas, GenAI consultations, safety considerations, and testing checkpoints, creating a transparent development trace aligned with assignment requirements.

In its final form, the project exemplifies how functional programming constructs can coexist with object-oriented design when guided by POPL principles. Representation independence is preserved, abstraction boundaries are respected, behaviour remains predictable, and extensions occur without modifying foundational components. The result is a cohesive, pedagogically aligned system that not only fulfills the stated objectives of the assignment but also demonstrates reflective development practices, architectural discipline, and deliberate alignment with theoretical concepts from the course.