NU BALIWAG

Digitizing Attendance and Records: A Web-Based Information System Utilizing Abstract Data Types

Presented to the Faculty of

School of Engineering and Technology

National University Bulacan

Baliwag, Bulacan

In partial fulfillment of the requirements

for the course CCDATCRL: Data Structures and Algorithms

Written by:

Espino, James Bryant DP.

Pariñas, John Carlo

I. INTRODUCTION

St. Luke's School of San Rafael is a small, community-focused, private educational institution committed to providing quality primary and secondary education. Its mission emphasizes fostering holistic student development, nurturing critical thinking, and promoting ethical values to prepare students for lifelong learning. The vision reflects a forward-looking approach, aspiring to cultivate well-rounded individuals who contribute meaningfully to society and adapt to evolving academic and technological landscapes. The school offers a variety of programs, catering to a small student population, while maintaining a focus on personalized guidance and academic excellence.

Despite its dedication to quality education, SLSSR faces several operational limitations that affect efficiency and overall institutional growth. Current administrative processes rely heavily on manual methods: attendance tracking is paper-based, grades and financial records are managed via disparate spreadsheets, and student/teacher interactions are not fully digitized. These constraints contribute to repeated data entry, slow retrieval of information, and higher potential for errors. In addition, the school experiences resource limitations in marketing and staffing, which further impact its ability to efficiently manage administrative duties and maintain stakeholder engagement.

To address these challenges, a proposed lightweight web-based information system aims to digitize critical administrative workflows. By implementing role-based access for students, teachers, and administrators, the system will streamline attendance recording, grade management, and financial tracking. Integration of optional two-factor authentication enhances security for sensitive operations, while modular JSON-based data storage ensures persistence and ease of future scaling. Ultimately, the system seeks to improve operational efficiency, reduce administrative burdens, and provide transparent, timely access to key academic and

financial information, aligning with SLSSR's mission to support student development and institutional growth.

II. INTERVIEW SUMMARY

During the interview conducted with the Principal of St. Luke's School of San Rafael, it was revealed that the school still relies heavily on manual and paper-based methods for managing attendance, student records, and financial transactions. Teachers record attendance using logbooks while students grades are stored in personal spreadsheets. The Finance office maintains ledgers for tuition tracking, which often results in duplication of entries, misplaced data, and time-consuming reconciliations. Staff members expressed frustration over these inefficiencies and the difficulty of retrieving student information, especially during enrollment and report card preparation.

The Principal agreed that the absence of unified digital system contributes to delays and inconsistencies in reporting. They also emphasized the need for transparency and accessibility of records, particularly for parents and students. However, they cited limited technical knowledge and budget constraints as key challenges in adopting existing digital management platforms. Most respondents expressed interest in a lightweight and secure system that could handle attendance logging, grade input, and financial record-keeping in an integrated way without relying on complex databases.

Overall, the interview findings highlight a shared desire among school personnel for a simple, reliable, and secure digital solution. They recognized that implementing a web-based system with built-in user roles and features like attendance tracking, grade management, and two-factor authentication would significantly improve workflow efficiency and reduce errors. The interview feedback served as a vital foundation for designing the proposed information

system, ensuring that the solution directly addresses the institution's operational pain points and aligns with its long-term goal of modernization.

III. IDENTIFIED PROBLEM AND PROPOSED SOLUTION

Identified Problem

The **target private school** currently **relies on manual and fragmented record-keeping**: paper-based attendance, teachers' spreadsheets or local notebooks for grades, and ad-hoc ledgers for finances. These processes cause **repeated data-entry, slow retrieval, higher error rates, and limited transparency for stakeholders** (students, parents, teachers, administrators). Multiple empirical studies indicate that digitizing administrative school records improves efficiency, transparency, and timeliness of interventions (better follow-ups for absenteeism) in similar institutional contexts (Pacheco, 2025; Abari et al., 2024).

Constraints Shaping The Solution

The project specification forbids SQL-based storage. Therefore, the **prototype** must **use a non-SQL persistent mechanism**. For a small-scale, quick-turn mock implementation we propose **JSON file–based storage** (module-separated JSON files such as `users.json`, `attendance.json`, `grades.json`, `financial.json`, `handbook.json`). File-based **JSON persistence is human-readable and requires no external DB server**, but it carries known limitations (data redundancy, concurrency control, limited query flexibility, and scalability concerns) that must be mitigated with pragmatic engineering controls such as file locking, atomic writes, and capped dataset sizes (LibreTexts, n.d.).

Proposed System

With all the problems and constraints stated, a lightweight web application implemented in PHP + HTML/CSS/JavaScript using JSON files for storage has been proposed to mitigate the issues. Core features include:

- **Role-based authentication** (Student, Teacher, Admin) with optional TOTP 2FA for higher-risk admin/teacher accounts
- **Attendance module**: student self check-in/out; teacher roll call for classes.
- **Grades module**: teacher input and student view.
- **Financial module**: admin record/payments viewable by student.
- **Handbook / references module**: built-in PDF viewer.

Mitigations for JSON/File Limitations

To address file-based risks the system uses:

- **Advisory file locks** (flock) around every read-modify-write operation to prevent concurrent corruption
- **Small, modular files** (separate JSON per module or per-day per-student files when necessary) to reduce write amplification.
- **Input validation + server-side checks** to preserve integrity.
- **Documented migration path**: swap file-IO functions to DB adapters later with minimal logic changes.

IV. ALGORITHM AND PSEUDOCODE

Below are the canonical algorithms (pseudocode) for the system's main workflows: registration/login with 2FA, attendance logging (student + teacher), grades recording, financial record operations, and the file-persist primitives (read/write with locking and id generation).

***Notation**:*

`readJSON(path)` → loads and parses JSON file into array/object.
`writeJSON(path, data)` → stringifies and writes JSON atomically.
`lockFile(path) and unlockFile(file)` → advisory lock wrappers.
`generateID(records)` → returns new integer ID (max+1).
`hashPassword(pw)` → secure password hashing (e.g., bcrypt).
`verifyPassword(pw, hash)` → compares input with stored hash.
`generateTOTP(secret) / verifyTOTP(secret, code)` → per RFC 6238 (M'Raihi et al., 2011).

1) File persistence primitives (atomic + locked)

```
FUNCTION load_json_locked(path):
      OPEN file at path
      LOCK file so no one else can edit right now
      contents = READ entire file
      IF contents is empty:
            data = empty list
      ELSE:
            data = PARSE contents as JSON
      RETURN (file, data)

FUNCTION save_json_and_unlock(file, path, data):
      REWIND file to beginning
      ERASE everything inside
      WRITE JSON-encoded data into file
      FLUSH file to make sure it is saved
      UNLOCK file
      CLOSE file
```

***Rationale**: Always take an exclusive lock for read-modify-write cycles to avoid concurrent writes. PHP's `flock()` provides a portable advisory locking mechanism. This ensures only one person edits the file at a time, preventing corruption.*

2) Unique ID allocator

```
FUNCTION next_id_for(path):
    (file, data) = load_json_locked(path)
    max_id = 0
    FOR EACH record IN data:
        IF record.id > max_id:
            max_id = record.id
    new_id = max_id + 1
    RETURN (file, new_id)
```

***Rationale**: Generates a new unique ID by looking at the biggest ID already in the file. For performance the file can also include an explicit `metadata` object holding `last_id` to avoid `O(n)` scan, but that requires careful locking.*

3.) Registration (with optional 2FA)

```
FUNCTION register(username, email, password, role, enable_2fa):
    (file, users) = load_json_locked("users.json")

    # check if username or email already exists
    FOR EACH u IN users:
        IF u.username == username OR u.email == email:
            save_json_and_unlock(file, "users.json", users)
            RETURN "User already exists"

    new_id = next available ID from users
    password_hash = HASH(password)

    IF enable_2fa is true:
        totp_secret = GENERATE_RANDOM_SECRET()
        is_2fa_enabled = true
    ELSE:
        totp_secret = null
        is_2fa_enabled = false

    user = {
        id: new_id,
        username: username,
        email: email,
        role: role,
        password_hash: password_hash,
        totp_secret: totp_secret,
        is_2fa_enabled: is_2fa_enabled,
        created_at: current time
    }

    ADD user to users
    save_json_and_unlock(file, "users.json", users)

    RETURN "Registration successful" (and QR code if 2FA enabled)
```

***Rationale**: Ensures uniqueness, stores secure hash of password, and optionally sets up 2FA.*
TOTP: Follow RFC 6238 (TOTP) for code generation/verification: codes typically 6 digits, 30s

4.) Login (with optional 2FA)

```
FUNCTION login(username_or_email, password):
    (file, users) = load_json_locked("users.json")
    user = FIND user in users where username OR email matches input
    save_json_and_unlock(file, "users.json", users)

    IF user not found:
        RETURN "Invalid credentials"

    IF password does not match user.password_hash:
        RETURN "Invalid credentials"

    IF user.is_2fa_enabled:
        TEMPORARY_SESSION.user_id = user.id
        TEMPORARY_SESSION.expires = now + 5 minutes
        RETURN "Require 2FA"
    ELSE:
        SESSION.user_id = user.id
        SESSION.auth_complete = true
        RETURN "Login successful"

# 2FA Verification Step
FUNCTION verify_2fa(code):
    IF TEMPORARY_SESSION missing OR expired:
        RETURN "Session expired, please login again"

```

```
    user = FIND user in users.json by TEMPORARY_SESSION.user_id

    IF TOTP_VERIFY(user.totp_secret, code) is true:
        SESSION.user_id = user.id
        SESSION.auth_complete = true
        CLEAR TEMPORARY_SESSION
        RETURN "Login successful"
    ELSE:
        RETURN "Invalid 2FA code"
```

> ***Rationale***: *Splits login into two stages if 2FA is enabled.*
> *TOTP: Follow RFC 6238 (TOTP) for code generation/verification: codes typically 6 digits, 30s*

5.) Student attendance check-in

```
FUNCTION student_check_in(student_id):
    today = CURRENT_DATE
    (file, attendance) = load_json_locked("attendance.json")

    # avoid duplicate check-in
    FOR EACH r IN attendance:
        IF r.student_id == student_id AND r.date == today:
            save_json_and_unlock(file, "attendance.json", attendance)
            RETURN "Already checked in"

    new_id = next available ID
    record = {
        id: new_id,
        student_id: student_id,
        date: today,
        time_in: CURRENT_TIME,
        status: "Present"
    }

    ADD record to attendance
    save_json_and_unlock(file, "attendance.json", attendance)
    RETURN "Check-in successful"
```

6.) Teacher marks class attendance

```
FUNCTION teacher_mark_attendance(teacher_id, class_list, date):
    (file, attendance) = load_json_locked("attendance.json")

    FOR EACH student_id, status IN class_list:
        found = false
        FOR EACH r IN attendance:
            IF r.student_id == student_id AND r.date == date:
                r.status = status
                found = true
        IF found is false:
            new_id = next available ID
            ADD {
                id: new_id,
                student_id: student_id,
                date: date,
                status: status,
                teacher_id: teacher_id
            } to attendance

    save_json_and_unlock(file, "attendance.json", attendance)
```

7.) Grades recording

```
FUNCTION add_grade(teacher_id, student_id, subject, term,
grade_value):
    (file, grades) = load_json_locked("grades.json")
```

```
3.
4.      new_id = next available ID
5.      record = {
6.          id: new_id,
7.          student_id: student_id,
8.          subject: subject,
9.          term: term,
10.         grade: grade_value,
11.         teacher_id: teacher_id,
12.         date_recorded: CURRENT_DATE
13.     }
14.
15.     ADD record to grades
16.     save_json_and_unlock(file, "grades.json", grades)
```

8.) Financial recording

```
1. FUNCTION add_charge(student_id, description, amount):
2.      (file, finance) = load_json_locked("financial.json")
3.
4.      new_id = next available ID
5.      record = {
6.          id: new_id,
7.          student_id: student_id,
8.          description: description,
9.          amount: amount,
10.         status: "Unpaid",
11.         date_added: CURRENT_DATE
12.     }
13.
14.     ADD record to finance
15.     save_json_and_unlock(file, "financial.json", finance)
16.
```

V. DATA STRUCTURE JUSTIFICATION

Efficient data handling in information systems depends heavily on choosing the right **abstract data types (ADTs)**. For this project, we focus on two fundamental ADTs: **Stacks (LIFO)** and **Queues (FIFO)**. Each is implemented in three variations to demonstrate their adaptability to different scenarios, system needs, and trade-offs.

Why use a Stack (LIFO)?

A **stack** retrieves the most recently added item first. This "last-in, first-out" property is ideal for situations where recent history matters most.

Use-cases in this system:

1. **Undo / change history** — administrators or teachers may want to revert the last local change (grades, attendance edits).
2. **Login attempt auditing** — most recent failed attempts can be examined to detect suspicious behavior.
3. **Short-lived server task stacks** — temporary storage for pending actions before batch persistence.

Complexity note: In standard implementations, both push (add) and pop (remove) operations are **O(1)** (GeeksforGeeks, 2024).

Stack Implementation A — Array-based (in-memory)

Structure: Use a PHP array with **`array_push()`** and **`array_pop()`.**

- **`push(x)`** → **`array_push(stack, x)`** → **`O(1)`** amortized.
- **`pop()`** → **`array_pop(stack)`** → **`O(1).`**
- **Persistence:** Snapshot entire array into `stack.json` on each change → **`O(n)`** write cost.

Use-case mapping: Undo history in a session; quick throwaway stacks.

Pros: Direct, minimal code.

Cons: Persistence is expensive since each update rewrites the full file.

Stack Implementation B — Linked-list (in-memory)

Structure: Each node = {value, next}, with a pointer to top.

- `push(x)` → new node points to old top; top updated → `O(1)`.
- `pop()` → top node removed; top updated → `O(1)`.
- **Persistence:** Serialize to JSON array → `O(n)`.

Pros: Pure O(1) push/pop without array reindexing.

Cons: Higher memory overhead; persistence requires serialization.

Stack Implementation C — File-backed stack (persistent JSON tail)

Structure: stack.json stores the stack as a JSON array.

- `push(x)` → read file, append, write back → `O(n)`.
- `pop()` → read file, remove last element, write back → `O(n)`.

Pros: Every change is durable on disk.

Cons: Slow due to file I/O; requires flock() to prevent corruption.

Why use a Queue (FIFO)?

A **queue** processes items in the order they arrive — "first in, first out." This guarantees fairness and chronological processing.

Use-cases in this system:

1. **Attendance check-in ordering** — first student logged is processed first.
2. **Payment receipts** — fairness in transaction handling.
3. **Task batching** — temporary buffering of requests before persistence.

Complexity note: `Enqueue` (add) and `dequeue` (remove) can both be `O(1)` in linked-list or circular buffer forms, but naïve arrays can suffer `O(n)` removal cost due to reindexing (GeeksforGeeks, 2024).

Queue Implementation A — Array-based (naïve PHP array)

Structure: Use PHP arrays with array_push() for enqueue and array_shift() for dequeue.

- `enqueue(x)` → `array_push(queue, x)` → `O(1)`.
- `dequeue()` → `array_shift(queue)` → `O(n)` due to reindexing.

Use-case mapping: Tiny demo queues (e.g., classroom check-ins).

Pros: Simple to make.

Cons: Dequeue performance collapses for larger queues.

Queue Implementation B — Linked-list queue

Structure: Nodes with value, next, and pointers to head and tail.

- `enqueue(x)` → add at tail → `O(1)`.
- `dequeue()` → remove from head → `O(1)`.
- **Persistence:** Serialize to JSON → `O(n)`.

Pros: True `O(1)` operations; stable for larger queues.

Cons: Requires memory management and serialization.

NU BALIWAG

Queue Implementation C — Circular buffer (array ring)

Structure: Fixed-size array with head and tail indexes, wrapping around modulo capacity.

- `enqueue(x)` → place at tail, advance index → `O(1)`.
- `dequeue()` → take from head, advance index → `O(1)`.
- **Resizing:** If buffer fills, resize cost `O(n)`.

Use-case mapping: Bounded attendance lists (e.g., one class of 40 students).

Pros: Memory efficient, constant-time operations.

Cons: Implementation complexity; resizing policy required for large data.

Comparison Table

Implementation	Push/Enqueue	Pop/Dequeue	Persistence cost	Best for
Array stack	O(1)	O(1)	O(n)	Undo history, sessions
Linked-list stack	O(1)	O(1)	O(n)	Frequent push/pop
File-backed stack	O(n)	O(n)	O(n)	Persistent audit trail
Array queue	O(1)	O(n)	O(n)	Tiny demo queues
Linked-list queue	O(1)	O(1)	O(n)	Production queue logic
Circular buffer	O(1)	O(1)	O(n)	Bounded attendance buffer

Weaknesses and Mitigations

1. **File persistence overhead**: Writing the full JSON on every change is `O(n)`. For small-scale prototypes this is acceptable, but in production this would not scale. *Mitigation:* Batch writes or migrate to document databases.
2. **Concurrency hazards**: Without locking, two simultaneous writes may corrupt files. *Mitigation:* Use advisory locks (`flock`) and atomic writes (`ftruncate + rewind + fflush`).
3. **2FA storage risks**: Time-based one-time password (TOTP) secrets must tolerate small clock skews and should ideally be encrypted at rest. *Mitigation:* Use server time sync (NTP) and encrypt stored secrets (IETF, 2011).
4. **Array inefficiency in PHP**: `array_shift()` is linear because of reindexing. *Mitigation:* Prefer linked-list or circular buffer structures for real workloads.

VI. DIAGRAMS

VII. COMPLEXITY ANALYSIS

Efficient algorithm design requires evaluating the **time complexity** (speed of operations as the dataset grows) and **space complexity** (memory usage). For this educational information system, the chosen **abstract data types** (ADTs) are **LIFO** (Last-In First-Out / Stack) and **FIFO** (First-In First-Out / Queue). (Cormen, Leiserson, Rivest, & Stein, 2009).

These ADTs are used in multiple parts of the system:

- **Stacks (LIFO)**: useful for login session history, undoing grade entries, and handling last attendance corrections.
- **Queues (FIFO)**: useful for processing attendance logs in arrival order, financial payment processing, and grade report generation in sequence.

Each ADT is implemented in three common ways to demonstrate flexibility: arrays, linked lists, and file/JSON persistence.

1. LIFO Implementations (Stack)

a. Array-based Stack

Operations:

- `push(item)` → append at end
- `pop()` → remove from end
- `peek()` → view last item without removing

Complexity:

- `Push: O(1)` (amortized, unless array resizes)
- `Pop: O(1)`
- `Peek: O(1)`
- `Space: O(n)`

Rationale: Works well for session history (e.g., last user actions).

b. Linked List Stack

Operations:

- `push(item)` → add node to head
- `pop()` → remove node from head
- `peek()` → return head node

Complexity:

- `Push: O(1)`
- `Pop: O(1)`

- Peek: O(1)
- Space: O(n) (extra overhead for node pointers)

Rationale: No resizing cost, suitable if history grows unpredictably.

c. File-based Stack (JSON persistence)

Operations:

- push(item) → load JSON, append, save back
- pop() → load JSON, remove last, save back
- peek() → load JSON, return last element

Complexity:

- Push/Pop/Peek: O(n) (file must be re-written each time)
- Space: O(n) in disk size

Rationale: Less efficient but persistent; useful for system logs that must survive crashes.

2. FIFO Implementations (Queue)

a. Array-based Queue

Operations:

- enqueue(item) → add to end
- dequeue() → remove from front

Complexity:

- Enqueue: O(1)
- Dequeue: O(n) if naïve (shifting elements); can be improved to O(1) with circular buffer indexing
- Space: O(n)

Rationale: Handles attendance check-ins in arrival order.

b. Linked List Queue

Operations:

- enqueue(item) → add node to tail
- dequeue() → remove node from head

Complexity:

- `Enqueue: O(1)`
- `Dequeue: O(1)`
- `Space: O(n)`

Rationale: More efficient than arrays for continuous `enqueue/dequeue` cycles.

c. File-based Queue (JSON persistence)

Operations:

- `enqueue(item)` → load JSON, append at end, save
- `dequeue()` → load JSON, remove first, save

Complexity:

- `Enqueue: O(n)` (full file rewrite)
- `Dequeue: O(n)` (shifting + rewrite)
- `Space: O(n)` in disk size

Rationale: Ensures financial transactions or attendance logs are persistent even if the system shuts down.

Summary Table

ADT	Implementation	Push/Enqueue	Pop/Dequeue	Space	Use Case Example
Stack	Array	O(1)	O(1)	O(n)	Login session undo
Stack	Linked List	O(1)	O(1)	O(n)	Attendance corrections
Stack	File/JSON	O(n)	O(n)	O(n)	Persistent system logs
Queue	Array	O(1) / O(n)	O(n)	O(n)	Attendance order

Queue	Linked List	O(1)	O(1)	O(n)	Processing grade inputs
Queue	File/JSON	O(n)	O(n)	O(n)	Persistent financial records

Discussion

The system leverages **fast in-memory structures** for live interactions and **file-based persistence** for long-term security. This dual design balances efficiency with reliability: students checking in must be recorded instantly, while financial and grading data must survive even system crashes. By presenting three implementations of each ADT, the design demonstrates both adaptability and awareness of computational trade-offs, aligning with best practices in algorithm analysis (Cormen et al., 2009; Goodrich et al., 2014).

VIII. CONCLUSION

The development of a lightweight web-based information system for St. Luke School of San Rafael responds directly to its operational challenges in attendance monitoring, grade management, and financial record keeping. By replacing manual, paper-based, and fragmented processes with a role-based digital platform, the system provides a foundation for efficiency, transparency, and reliability in administrative functions.

Through the integration of secure login features, optional two-factor authentication, and modular JSON file persistence, the prototype demonstrates a practical yet scalable approach within the given technical constraints. The system also incorporates fundamental data structures, stacks and queues, with multiple implementations, highlighting the importance of computational efficiency and persistence in handling real-world school data.

While limited in scope as a mock implementation, the project demonstrates the concrete advantages of digitization in small schools: lower administrative burden, quicker access to information, and enhanced control of both academic and financial records. These enhancements directly facilitate the school's mission of integrated student development by releasing employees to attend more to instruction and learner support.

Looking forward, the system establishes a clear migration pathway toward more advanced technologies such as database-driven storage, cloud-based solutions, and broader analytics capabilities. Thus, the project not only addresses current operational gaps but also positions St. Luke School of San Rafael to embrace future innovations, ensuring that its commitment to quality education remains resilient in an increasingly digital academic environment.

To further extend the capabilities of the system and make it sustainable in the long run, it is suggested that St. Luke School of San Rafael considers transitioning switch away from JSON file storage towards a strong relational database management system (RDBMS) like MySQL or PostgreSQL. This would enhance data integrity, allow multi-user access with concurrency, and facilitate advanced reporting and analytics. Cloud-based hosting and backup services can offer secure, scalable access while protecting against data loss. Also, adopting role-based dashboards, automated alerts, and mobile-enabled interfaces would simplify workflows and improve user experience for staff and students alike. In the longer term, adding analytics tools could provide insights into trends in student performance, attendance, and finance, to inform data-driven decision-making. Lastly, examining interoperability with learning management systems (LMS) or third-party educational platforms can put the school in a position to capitalize on wider digital education trends, so that the system keeps pace with upcoming technological and academic demands.

IX. REFERENCES

Abari, A. A., Adewuyi, A. A., & Jimoh, M. F. (2024). Influence of digitized record-keeping practices on administrative effectiveness. *Educational Perspectives, 12*(2), 45–57. Retrieved from https://educationalperspectives.org.ng/upload_file/Abari%20et%20al.%202024%20%281%29.pdf

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Engineering LibreTexts. (2024). File structures and complexity analysis. Retrieved from https://eng.libretexts.org

GeeksforGeeks. (2024). *Queue data structure*. Retrieved from https://www.geeksforgeeks.org/queue-data-structure

GeeksforGeeks. (2024). *Stack data structure*. Retrieved from https://www.geeksforgeeks.org/stack-data-structure

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.

IETF Datatracker. (2011). RFC 6238: TOTP: Time-based one-time password algorithm. Internet Engineering Task Force. Retrieved from https://datatracker.ietf.org/doc/html/rfc6238

Pacheco, A. (2025). Impact of digitization on educational management: Results from private institutions. *Journal of Educational Technology, 15*(1), 33–47. Retrieved from https://www.sciencedirect.com/science/article/pii/S2451958825000077

NU BALIWAG

Project Presentation Rubric – 50 Points

Concept Adaptation – LIFO (Stack) (10 points)

- Excellent (10): All 3 LIFO implementations are present, accurate, and naturally adapted to the application. Clear demonstration of LIFO concept.
- Very Good (8): All 3 LIFO present, mostly accurate, minor adaptation issues.
- Satisfactory (6): Only 2 correct LIFO, 1 weakly applied.
- Needs Improvement (4): Only 1 LIFO correctly used; adaptation forced.
- Poor (2): No clear LIFO concept shown.

Concept Adaptation – FIFO (Queue) (10 points)

- Excellent (10): All 3 FIFO implementations are present, accurate, and naturally adapted to the application. Clear demonstration of FIFO concept.
- Very Good (8): All 3 FIFO present, mostly accurate, minor adaptation issues.
- Satisfactory (6): Only 2 correct FIFO, 1 weakly applied.
- Needs Improvement (4): Only 1 FIFO correctly used; adaptation forced.
- Poor (2): No clear FIFO concept shown.

Error Handling & Smooth Transition (10 points)

- Excellent (10): Application runs flawlessly with smooth navigation and no noticeable errors. Transitions are seamless and professional.
- Very Good (8): Runs well with only minor, non-disruptive issues.
- Satisfactory (6): Some errors or confusing transitions but still functional.
- Needs Improvement (4): Frequent errors or rough transitions disrupt flow.
- Poor (2): Application fails to function properly; transitions break presentation.

Fit of Concepts to Process (10 points)

- Excellent (10): LIFO & FIFO are strongly integrated into real processes and add clear value to the application.
- Very Good (8): Concepts are well-adapted with small mismatches.
- Satisfactory (6): Concepts adapted but feel somewhat forced or superficial.
- Needs Improvement (4): Weak connection between concepts and the process.
- Poor (2): No real connection between concepts and the process.

Punctuality & Preparedness (10 points)

- Excellent (10): Arrives on time, fully prepared, confident, explains clearly without relying on notes.
- Very Good (8): Slightly late or minor reliance on notes, but overall clear.
- Satisfactory (6): Moderately prepared, explanations somewhat unclear or heavy note reliance.
- Needs Improvement (4): Unprepared, struggles with explanations, poor presentation delivery.
- Poor (2): Very unprepared, late, unable to explain concepts.

Scoring

5 Criteria × 10 points = 50 points total

Grade Scale:

45–50 = Excellent (A)
40–44 = Very Good (B)
30–39 = Satisfactory (C)
20–29 = Needs Improvement (D)
19 and below = Poor (F)

Total Score: _____

_____

Panel's Signature