المدرسة الوطنية العليا في الأمن السيبراني
NATIONAL SCHOOL OF CYBERSECURITY

END SEMESTER PROJECT REPORT (ALSDS)

2025

Presented for :
BERGHOUT Yasser Moussa
BELAYADI Djahida

Presented by :
Hassani Fateh
Abbaci Mohamed Sadek
Kassoul Mohamed Ali

Introduction

In May2025 , our team : HassaniFateh, KassoulMohamedAli,and Abbaci Mohamed Sadek , embarked on this project as the capstone for our Data Structures and Algorithms course for the second term of the first year at the national school in cyber security . Throughout development, we honed our abilities to conduct thorough literature reviews, navigate official documentation, and maintain detailed design logs . These research and documentation efforts ensured that every decision was grounded in best practices and that the codebase remained comprehensible and maintainable .

The ALSD project comprises two complementary applications that together showcase a broad spectrum of data structure implementations:

- Security Logs Management System: Designed for real-time ingestion, storage, and analysis of security events. We leverage linked lists for chronological storage, queues for buffering, and stacks for reversed traversals. Parsing routines built on low-level character and string handling facilitate robust timestamp and message management.
- Dictionary Management System: A text-driven dictionary supporting synonyms and antonyms . Core lookup and CRUD operations are implemented on an AVL-balanced binary search tree, supplemented by linked lists for relational data. Recursive algorithms drive tree insertion, deletion, and traversal, ensuring efficient operation even as the dictionary grows.

To foster collaboration and reproducibility, all source code, test suites, and design artifacts are available on GitHub :

Subsequent sections detail architecture, implementation and empirical testing results for each subsystem.

Part 1: Security Logs Management

Part 1: Security Logs Management

Provides a suite of modules to efficiently handle and analyze system security logs, leveraging various data structures:

1. **Singly & Doubly Linked Lists**
   - Core operations: insertion (beginning, end, middle), deletion, search, traversal (forward/backward), merging, and reversing.
   - Advanced features: fixed-size log buffer using circular lists, middle-element deletion, cycle detection for data integrity.
2. **Circular Linked Lists**
   - Seamless wrap-around storage for continuous log buffers.
   - Efficient overwrite of oldest entries, cycle detection with Floyd's algorithm, and complete traversal without sentinel nodes.
3. **Queues**
   - Standard FIFO log processing: enqueue, dequeue, peek, and emptiness/fullness checks.
   - Circular queue variant for memory efficiency and constant-time operations.
4. **Stacks**
   - LIFO-based log operations: push, pop, peek, and state checks.
   - Recursive reversal and cycle detection within log mappings.
5. **Recursion**
   - Recursive reversal of linked lists, factorial/Fibonacci calculators, binary/search algorithms, maximum ID computations, and infix-to-postfix conversions.
6. **Trees**
   - Binary Search Trees for log indexing and retrieval, including insertion, deletion, traversal, and balancing checks.

Linked Linear Lists

1. Insert Log Entry

- Inputs: LogEntry entry, Position pos (enum: BEGINNING, END, INDEX)
- Outputs: bool success
- Description: Depending on pos, the function inserts a new node at the head, tail, or a given index in the linked list.
- Implementation Difficulties:
  - Maintaining list integrity when inserting at arbitrary index (boundary checks, off-by-one errors).
  - Updating head and tail pointers correctly.
  - Handling empty-list cases.
- Design Thinking: We used a two-pointer traversal for middle inserts: one pointer to track the insertion point and one to maintain the previous node. Enum-driven control flow ensures clarity and extensibility.

2. Delete Log Entry

- Inputs: DeleteCriteria criteria (ID, TIMESTAMP, FIRST, LAST), value (for ID or timestamp)
- Outputs: bool success
- Description: Finds a node matching criteria and unlinks it from the list.
- Implementation Difficulties:
  - Searching and unlinking while preserving list continuity.
  - Edge cases: deleting head or tail, empty list, non-existent node.
  - Memory deallocation to avoid leaks.
- Design Thinking: We separated searching from deletion: first locate the target node (with a helper), then perform unlinking. This separation simplified debugging and unit testing.

3. Search Log Entry

- Inputs: SearchCriteria criteria (ID, KEYWORD, TIMESTAMP), value (string or timestamp)
- Outputs: List<LogEntry> matching entries
- Description: Iterates through the list, comparing each node against the search criteria, collecting matches.
- Implementation Difficulties:
  - Parsing and matching against timestamp ranges.
  - Case-insensitive keyword matching requiring string normalization.
  - Efficiently returning multiple results without excessive memory allocation.
- Design Thinking: We implemented a generic iterator pattern with callback predicates, allowing reuse for different criteria and minimizing code duplication.

4. Sort Logs

- Inputs: SortKey key (DATE, SEVERITY)
- Outputs: void (list reordered in-place)
- Description: Applies an in-place merge sort on the linked list, comparing nodes based on the specified key.
- Implementation Difficulties:
  - Implementing merge sort on a linked list without random access.
  - Ensuring stable sorting when keys are equal.
- Design Thinking: We chose merge sort due to its O(nlogn) performance and suitability for linked lists (no extra arrays needed). Recursive divide-and-conquer neatly splits and merges sublists.

5. Reverse the List

- Inputs: none
- Outputs: void (list reversed in-place)
- Description: Reverses the next pointers of the linked list, swapping head and tail.
- Implementation Difficulties:
  - Managing three pointers (prev, current, next) without losing references.
  - Updating head/tail after reversal.
- Design Thinking: We applied the classic iterative reversal algorithm, which traverses once and flips pointers, achieving O(n) time with O(1) extra space.

6. Count Total Logs

- Inputs: none
- Outputs: int count
- Description: Traverses the list, incrementing a counter per node.
- Implementation Difficulties: Minimal; ensuring counters are reset before traversal.
- Design Thinking: A straightforward accumulator pattern, used internally for validations and stats.

Bidirectional Linked Lists

1. Move Forward and Backward Through Logs

- Inputs:
  - A reference to the current node (log) in the bidirectional linked list.
  - A direction indicator (forward or backward).
- Outputs:
  - A reference to the next or previous node based on the direction.
- Implementation Difficulties:
  - Ensuring the navigation handles None values gracefully when reaching the beginning or end of the list.
  - Managing edge cases, such as empty lists or nodes without next or previous links.
- Thinking Method Used:
  - Iterative navigation through node references using next and prev pointers.
  - Conditional checks to prevent null pointer exceptions.

2. Delete Middle Element (Remove a Log at a Specific Index)

- Inputs:
  - The head of the list.
  - The index of the element to be deleted.
- Outputs:
  - The modified list with the specific element removed.
- Implementation Difficulties:
  - Navigating to the correct index efficiently.
  - Properly updating prev and next pointers of surrounding nodes to maintain the list structure.
  - Handling edge cases like deleting the head, tail, or an index out of range.
- Thinking Method Used:
  - Two-pointer traversal to locate the target node.
  - Pointer manipulation to bypass the targeted node while maintaining link integrity.

3. Merge Two Log Lists (Combine Logs from Two Sources)

- Inputs:
  - Two head references of two separate bidirectional linked lists.
- Outputs:
  - A single merged list containing elements from both lists.
- Implementation Difficulties:
  - Correctly linking the prev and next pointers of the last node of the first list and the first node of the second list.
  - Handling cases where one or both lists are empty.
  - Maintaining proper order and link continuity during the merge.
- Thinking Method Used:
  - Pointer navigation to the end of the first list and the beginning of the second.

Conditional checks to prevent invalid memory access during linking.

4. Insert Log Entry (at the beginning, at the end, at a specific position)

- Inputs:
  - Log entry data.
  - Insertion position (beginning, end, or specific index).
- Outputs:
  - The modified list with the new log entry inserted at the specified location.
- Implementation Difficulties:
  - Efficiently locating the insertion point.
  - Maintaining the integrity of prev and next links during insertion.
- Thinking Method Used:
  - Pointer traversal and adjustment of node links during insertion.

5. Delete Log Entry (by ID, by timestamp, delete first/last)

- Inputs:
  - Criteria for deletion (ID, timestamp, first, or last entry).
- Outputs:
  - The modified list with the specified log entry removed.
- Implementation Difficulties:
  - Identifying the target node based on different criteria.
  - Properly maintaining list structure after deletion.
- Thinking Method Used:
  - Search traversal followed by pointer adjustment for proper removal.

6. Search Log Entry (by ID, by keyword, by timestamp)

- Inputs:
  - Search criteria (ID, keyword, timestamp).
- Outputs:
  - A reference to the matching log entry or None if not found.
- Implementation Difficulties:
  - Efficiently scanning through the list while minimizing search time.
- Thinking Method Used:
  - Linear traversal with conditional matching.

7. Sort Logs (by date, by severity level)

- Inputs:
  - Sorting criteria (date, severity level).
- Outputs:
  - The list sorted based on the specified criteria.
- Implementation Difficulties:
  - Maintaining pointer integrity while rearranging nodes.
- Thinking Method Used:

Sorting algorithms adapted for linked list manipulation.

8. Reverse the List (to view logs in reverse order)

- Inputs:
  - Head of the list.
- Outputs:
  - The list with the order of logs reversed.
- Implementation Difficulties:
  - Swapping the next and prev pointers efficiently.
- Thinking Method Used:
  - Iterative pointer swapping.

9. Count Total Logs (return the number of log entries)

- Inputs:
  - Head of the list.
- Outputs:
  - Total number of log entries.
- Implementation Difficulties:
  - Ensuring the entire list is traversed without missing any nodes.
- Thinking Method Used:
  - Simple traversal with an incrementing counter.

Circular Linked List

1. Insert Log Entry (at the beginning, at the end, at a specific position)

- Inputs:
  - Log entry data.
  - Insertion position (beginning, end, or specific index).
- Outputs:
  - The circular linked list with the new log entry inserted at the specified location.
- Implementation Difficulties:
  - Efficiently finding the insertion point, especially for specific indices in a circular structure.
  - Ensuring the next reference of the last node points back to the head for circular continuity.
- Thinking Method Used:
  - Pointer navigation with additional checks for circular linkage integrity.

2. Delete Log Entry (by ID, by timestamp, delete first/last)

- Inputs:
  - Criteria for deletion (ID, timestamp, first, or last entry).
- Outputs:
  - The modified circular list with the specified log entry removed.
- Implementation Difficulties:
  - Correctly handling the circular linkage during deletion, especially for the head or tail node.
  - Preventing loss of circular reference.
- Thinking Method Used:
  - Search and pointer adjustments with special handling for head and tail.

3. Search Log Entry (by ID, by keyword, by timestamp)

- Inputs:
  - Search criteria (ID, keyword, timestamp).
- Outputs:
  - A reference to the matching log entry or None if not found.
- Implementation Difficulties:
  - Ensuring that the search does not enter an infinite loop due to circular structure.
- Thinking Method Used:
  - Iterative traversal with a stopping condition at the head.

4. Sort Logs (by date, by severity level)

- Inputs:
  - Sorting criteria (date, severity level).
- Outputs:
  - The list sorted based on the specified criteria.
- Implementation Difficulties:
  - Maintaining circular continuity while rearranging nodes.
- Thinking Method Used:

Sorting algorithms adapted for circular linked lists with careful pointer adjustments.

5. Reverse the List (to view logs in reverse order)

- Inputs:
  - Head of the list.
- Outputs:
  - The circular list with the order of logs reversed.
- Implementation Difficulties:
  - Reversing the next pointers while maintaining the circular nature.
- Thinking Method Used:
  - Iterative pointer swapping and resetting the head reference.

6. Count Total Logs (return the number of log entries)

- Inputs:
  - Head of the list.
- Outputs:
  - Total number of log entries.
- Implementation Difficulties:
  - Ensuring the traversal does not loop infinitely.
- Thinking Method Used:
  - Simple traversal with a stopping condition back at the head.

7. Implement a Fixed-Size Log Buffer (overwrite old logs automatically)

- Inputs:
  - Maximum buffer size.
- Outputs:
  - Circular buffer with automatic overwrites.
- Implementation Difficulties:
  - Managing pointers efficiently to overwrite the oldest log without breaking the circular reference.
- Thinking Method Used:
  - Modular arithmetic to wrap around the list as needed.

8. Detect Cycles in the List (validate log data consistency)

- Inputs:
  - Head of the list.
- Outputs:
  - Boolean indicating if a cycle is detected.
- Implementation Difficulties:
  - Avoiding infinite loops during traversal.
- Thinking Method Used:
  - Floyd's Cycle-Finding Algorithm (Tortoise and Hare Method).

Queues

1. Enqueue New Log Entry (insert log at the end)

- Inputs:
  - Log entry data.
- Outputs:
  - Queue with the new entry added at the tail.
- Implementation Difficulties:
  - Handling cases where the queue is full (fixed capacity) or uninitialized.
  - Maintaining correct head and tail pointers.
- Thinking Method Used:
  - Pointer or index increment with wrap-around logic (for circular buffer).

2. Dequeue Log Entry (remove log from the front)

- Inputs:
  - None (operates on the current queue state).
- Outputs:
  - The removed log entry and the updated queue.
- Implementation Difficulties:
  - Handling empty queue underflow.
  - Updating head pointer correctly.
- Thinking Method Used:
  - Pointer or index increment with checking for empty conditions.

3. Peek (view the first log without removing it)

- Inputs:
  - None (operates on the current queue state).
- Outputs:
  - A reference to the front log entry or None if empty.
- Implementation Difficulties:
  - Ensuring the operation is O(1) and non-destructive.
- Thinking Method Used:
  - Direct access to head pointer or index.

4. Check if Queue is Empty or Full

- Inputs:
  - None (operates on the current queue state).
- Outputs:
  - Boolean indicators for empty and full states.
- Implementation Difficulties:
  - Distinguishing full vs empty when head == tail (requires extra flag or reserved slot).
- Thinking Method Used:

Capacity tracking or one-slot buffer to differentiate states.

5. Circular Queue Implementation (for efficient memory usage)

- Inputs:
  - Log entry data for enqueue, and control operations.
- Outputs:
  - Fixed-size circular buffer supporting O(1) enqueue/dequeue.
- Implementation Difficulties:
  - Managing wrap-around of head and tail pointers without losing data.
  - Avoiding false full/empty detection.
- Thinking Method Used:
  - Modular arithmetic on head/tail indices and optional count tracking.

Stacks

1. Push New Log Entry (insert log at the top)

- Inputs:
  - Log entry data.
- Outputs:
  - Stack with the new entry added at the top.
- Implementation Difficulties:
  - Handling stack overflow when the maximum capacity is reached (if bounded).
- Thinking Method Used:
  - Pointer/index decrement or list append operation.

2. Pop Log Entry (remove log from the top)

- Inputs:
  - None (operates on the current stack state).
- Outputs:
  - The removed log entry and the updated stack.
- Implementation Difficulties:
  - Handling stack underflow when attempting to pop from an empty stack.
- Thinking Method Used:
  - Pointer/index increment or list pop operation.

3. Peek (view the top log without removing it)

- Inputs:
  - None (operates on the current stack state).
- Outputs:
  - A reference to the top log entry or None if empty.
- Implementation Difficulties:
  - Ensuring O(1) access without modifying stack state.
- Thinking Method Used:
  - Direct access to the top pointer/index or list end.

4. Check if Stack is Empty or Full

- Inputs:
  - None (operates on the current stack state).
- Outputs:
  - Boolean indicators for empty and full states.
- Implementation Difficulties:
  - Differentiating full vs empty when using fixed-size arrays (empty when top == -1, full when top == capacity -1).
- Thinking Method Used:

Compare top index against boundaries or track size.

5. Reverse a Stack Using Recursion

- Inputs:
  - The stack to be reversed.
- Outputs:
  - The stack with elements in reversed order.
- Implementation Difficulties:
  - Managing call stack depth for large stacks.
  - Temporarily holding values during recursion.
- Thinking Method Used:
  - Recursive pop operations to the bottom, then pushing elements back in reversed order.

Recursion

1. Reverse a Linked List Using Recursion

- Inputs:
  - Head of the linked list.
- Outputs:
  - Head of the reversed linked list.
- Implementation Difficulties:
  - Managing pointer reversal during the recursion unwind phase.
  - Base case handling for empty or single-node lists.
- Thinking Method Used:
  - Recursive calls to reverse the rest of the list, then adjusting next pointers.

2. Calculate Factorial and Fibonacci Using Recursion

- Inputs:
  - Integer n.
- Outputs:
  - n! for factorial; the nth Fibonacci number.
- Implementation Difficulties:
  - Exponential time complexity in naive Fibonacci recursion.
  - Stack overflow for large n without memoization.
- Thinking Method Used:
  - Direct recursive definition: factorial(n) = n * factorial(n-1); fib(n) = fib(n-1) + fib(n-2).

3. Find Maximum Log Entry ID Using Recursion

- Inputs:
  - Head of the linked list of logs.
- Outputs:
  - Maximum ID value found in the list.
- Implementation Difficulties:
  - Traversing the entire list recursively without missing nodes.
- Thinking Method Used:
  - Compare current node ID with max from recursion on the remainder of the list.

4. Implement Recursive Binary Search in Sorted Logs

- Inputs:
  - Array or list of sorted log entries.
  - Search key (ID or timestamp).
  - Low and high indices.
- Outputs:
  - Index of the matching entry or -1 if not found.
- Implementation Difficulties:
  - Correctly updating mid, low, high bounds in recursive calls.
- Thinking Method Used:

Divide-and-conquer: compare with mid element, then recurse on appropriate half.

5. Convert an Infix Expression to Postfix Using Recursion

- Inputs:
  - String of infix expression tokens.
- Outputs:
  - String or list of tokens in postfix order.
- Implementation Difficulties:
  - Handling operator precedence and parentheses recursively.
- Thinking Method Used:
  - Recursive descent parsing: process subexpressions in parentheses, apply precedence rules.

Trees

1. Binary Search Tree (BST) for Log Searching

- Inputs:
  - None (describes data structure).
- Outputs:
  - A BST where each node holds a log entry keyed by timestamp or ID.
- Implementation Difficulties:
  - Balancing considerations for worst-case performance.
- Thinking Method Used:
  - Node struct with left and right pointers, standard BST invariants.

2. Insert Log into BST (logs sorted by timestamp)

- Inputs:
  - Root of BST.
  - Log entry with timestamp key.
- Outputs:
  - Root of BST with new node inserted.
- Implementation Difficulties:
  - Maintaining BST property during insertion.
- Thinking Method Used:
  - Recursive or iterative traversal comparing keys to place the new node.

3. Delete Log from BST

- Inputs:
  - Root of BST.
  - Key (timestamp or ID) to delete.
- Outputs:
  - Root of BST with the specified node removed.
- Implementation Difficulties:
  - Handling three cases: leaf node, single child, two children (in-order successor).
- Thinking Method Used:
  - Recursive deletion with pointer reassignments for each case.

4. Search Log in BST

- Inputs:
  - Root of BST.
  - Key to search for.
- Outputs:
  - Reference to the node or None.
- Implementation Difficulties:
  - Ensuring O(h) complexity.
- Thinking Method Used:
  - Key comparisons to navigate left or right subtrees.

5. Traverse Logs in Different Orders (In-order, Pre-order, Post-order)

- Inputs:
  - Root of BST.
  - Desired traversal order.
- Outputs:
  - List of log entries in specified order.
- Implementation Difficulties:
  - Implementing each traversal recursively or iteratively without errors.
- Thinking Method Used:
  - Standard tree traversal algorithms.

6. Convert Linked List to BST (to improve search efficiency)

- Inputs:
  - Head of sorted linked list.
- Outputs:
  - Root of height-balanced BST.
- Implementation Difficulties:
  - Choosing mid-point without random access.
  - Ensuring balanced structure.
- Thinking Method Used:
  - Slow/fast pointer to find middle for root, then recursive build on sublists.

7. Implement a Heap Structure for Efficient Log Management

- Inputs:
  - None (describes data structure).
- Outputs:
  - Min-heap or max-heap for logs based on priority (e.g., severity).
- Implementation Difficulties:
  - Maintaining heap invariants during insert and delete operations.
- Thinking Method Used:
  - Array-based implementation with parent/child index calculations.
  -

Part 2: Dictionary Management

Part 2: Dictionary Management

Implements a feature-rich dictionary of words with synonyms and antonyms stored in a text file, parsed and handled via multiple data structures:

1. **Linked Lists & Queues**
   - Import and maintain synonym (=) and antonym (#) relationships through singly, doubly, and circular lists.
   - Queue-based grouping: syllable count organization and pronunciation categories (short, long, diphthong).
2. **Stacks**
   - Stack-based conversions from merged lists, lexicographical sorting, and cycle detection in word mappings.
   - Utilities: push/pop-based lookups, updates, and palindrome checks.
3. **Binary Search Trees**
   - Word indexing by key for fast search, insertion, deletion, and traversal orders (in-order, pre-order, post-order).
   - Advanced BST utilities: height/size computation, LCA, range counting, in-order successor, mirroring, balance checks, and balanced merging.
4. **Recursion**
   - File operations: recursive counting, removal, and replacement of word occurrences.
   - Word algorithms: generation of permutations, subsequences, longest common subsequence, distinct subsequence counting, and palindrome validation purely via recursion.
5. **CRUD & Search Features**
   - Full menu-driven interface enables: fetching word info, updating/deleting entries, substring and similarity filtering, palindrome and pattern queries.

Modules based on Linked lists and Queues

1. getSynWords(File *f)

- Inputs:
  - File pointer to a text file containing words and their synonyms.
- Outputs:
  - TList*: Linked list of nodes where each node contains:
    - A word.
    - Its synonym.
    - Number of characters.
    - Number of vowels.
- Implementation Difficulties:
  - Parsing the file format correctly.
  - Allocating and initializing list nodes dynamically.
  - Accurately counting characters and vowels per word.
- Thinking Method Used:
  - Sequential file read, string tokenization, and append to linked list.

2. getAntoWords(File *f)

- Inputs:
  - File pointer to a text file containing words and their antonyms.
- Outputs:
  - TList*: Linked list of nodes with the same structure as synonyms list.
- Implementation Difficulties:
  - Similar to synonyms: parsing, memory allocation, and attribute computation.
- Thinking Method Used:
  - Mirror approach to getSynWords, storing antonyms instead.

3. getInfWord(TList *syn, TList *ant, char *word)

- Inputs:
  - syn: Synonym list.
  - ant: Antonym list.
  - word: Target word to look up.
- Outputs:
  - Prints or returns:
    - Synonym.
    - Antonym.
    - Number of characters.
    - Number of vowels.
- Implementation Difficulties:
  - Searching two lists efficiently.
  - Handling missing entries gracefully.
- Thinking Method Used:
  - Linear traversal on each list with string comparison.

4. getInfWord2(TList *syn, TList *ant, char *inf)

- Inputs:
  - inf: A synonym or antonym to look up.
- Outputs:
  - Prints or returns the corresponding word, character count, and vowel count.
- Implementation Difficulties:
  - Determining which list contains inf.
- Thinking Method Used:
  - Sequential search on both lists until match found.

5. sortWord(TList *syn)

- Inputs:
  - Synonym list head.
- Outputs:
  - New list sorted alphabetically by word.
- Implementation Difficulties:
  - Linked list sorting: choosing algorithm (e.g., merge sort for O(n log n)).
  - Rewiring node next pointers correctly.
- Thinking Method Used:
  - Recursive merge sort adapted for linked lists.

6. sortWord2(TList *syn)

- Inputs:
  - Synonym list head.
- Outputs:
  - List sorted ascending by character count.
- Implementation Difficulties:
  - Comparing integer attributes instead of strings.
- Thinking Method Used:
  - Similar merge sort, with comparator on charCount.

7. sortWord3(TList *syn)

- Inputs:
  - Synonym list head.
- Outputs:
  - List sorted descending by vowel count.
- Implementation Difficulties:
  - Reverse ordering in comparator.
- Thinking Method Used:
  - Merge sort with comparator on vowelCount descending.

8. deleteWord(File *f, TList *syn, TList *ant, char *word)

- Inputs:
  - File pointer to original text.
  - Synonym and antonym lists.
  - word to delete.
- Outputs:
  - Updated file and both lists with word removed.
- Implementation Difficulties:
  - Editing file in-place or rewriting it.
  - Maintaining list integrity after deletion.
- Thinking Method Used:
  - Remove file entry via temporary file rewrite.
  - Unlink node from each list and free memory.

9.updateWord(File *f , TList *syn, TList *ant, char *word, char *syne, char *anton)

- Inputs:
  - Original file pointer.
  - Current lists.
  - word, new syne, new anton values.
- Outputs:
  - Updated file and list nodes reflecting new synonym and antonym.
- Implementation Difficulties:
  - Synchronizing file and list updates atomically.
- Thinking Method Used:
  - File rewrite similarly to delete, then update node data.

10. similarWord(TList *syn, char *word, int rate)

- Inputs:
  - Synonym list head.
  - Target word.
  - Minimum match rate (percentage).
- Outputs:
  - New list of words whose similarity ≥ rate.
- Implementation Difficulties:
  - Defining similarity metric (e.g., Levenshtein distance).
  - Filtering by threshold.
- Thinking Method Used:
  - Compute similarity against each node, append matches.

11. countWord(TList *syn, char *prt)

- Inputs:
  - Synonym list head.
  - Substring prt.
- Outputs:
  - New list of words containing prt.
- Implementation Difficulties:
  - Substring search per node.
- Thinking Method Used:
  - Use strstr or similar on each word.

12. palindromWord(TList *syn)

- Inputs:
  - Synonym list head.
- Outputs:
  - Alphabetically sorted list of palindrome words with their data.
- Implementation Difficulties:
  - Palindrome check for each word.
  - Insertion sort into output list for ordering.
- Thinking Method Used:
  - Check word == reverse(word), insert in order.

13. merge(TList *syn, TList *ant)

- Inputs:
  - Two list heads.
- Outputs:
  - Bidirectional linked list merging synonym and antonym into single nodes.
- Implementation Difficulties:
  - Converting singly to doubly linked structure.
- Thinking Method Used:
  - Simultaneous traversal, create new nodes linking prev/next.

14. merge2(TList *syn, TList *ant)

- Inputs:
  - Two list heads.
- Outputs:
  - Circular linked list merging synonym and antonym into single nodes.
- Implementation Difficulties:
  - Maintaining circular next pointer on last node.
- Thinking Method Used:
  - Similar to merge, with circular link at end.

15. addWord(TList *syn, TList *ant, char *word, char *syne, char *anton)

- Inputs:
  - Lists and new word, syne, anton.
- Outputs:
  - Updated lists and appended entry to text file.
- Implementation Difficulties:
  - File append and list node creation.
- Thinking Method Used:
  - Append to file, then create and link new nodes.

16. syllable(TList *syn)

- Inputs:
  - Synonym list head.
- Outputs:
  - TQueue*: Queue sorted by syllable count, words separated by empty strings to denote groups.
- Implementation Difficulties:
  - Counting syllables algorithmically.
  - Group separation using sentinel entries.
- Thinking Method Used:
  - Heuristic syllable count (e.g., vowel groups), enqueue accordingly.

17. prounounciation(TList *syn)

- Inputs:
  - Synonym list head.
- Outputs:
  - Three queues (short, long, diphthong).
- Implementation Difficulties:
  - Determining classification rules.
- Thinking Method Used:
  - Phonetic patterns, enqueue to respective queue.

18. toQueue(TList *merged)

- Inputs:
  - Head of merged list.
- Outputs:
  - TQueue*: Queue containing all list nodes in order.
- Implementation Difficulties:
  - Simple list-to-queue conversion.
- Thinking Method Used:
  - Traverse linked list, enqueue each node.

Modules based on Stacks

1. toStack(TList *merged)

- Inputs:
  - TList* head of merged bidirectional list.
- Outputs:
  - TStack*: Stack containing nodes from the list.
- Implementation Difficulties:
  - Preserving original list order when pushing to stack.
- Thinking Method Used:
  - Traverse list and push each node's data onto stack.

2. getInfWordStack(TStack *stk, char *word)

- Inputs:
  - stk: Stack of log entries.
  - word: Target word.
- Outputs:
  - Data structure or printed output with synonym, antonym, character count, vowel count.
- Implementation Difficulties:
  - Iterating through stack without destroying its content.
- Thinking Method Used:
  - Pop elements to auxiliary stack for inspection then restore.

3. sortWordStack(TStack *stk)

- Inputs:
  - stk: Stack of words.
- Outputs:
  - New stack sorted alphabetically.
- Implementation Difficulties:
  - Sorting stack using only stack operations.
- Thinking Method Used:
  - Use auxiliary stack and recursive insertion for order.

4. deleteWordStack(TStack *stk, char *word)

- Inputs:
  - stk: Original stack.
  - word: Word to delete.
- Outputs:
  - Updated stack without the target word.
- Implementation Difficulties:
  - Preserving stack order while removing an arbitrary element.
- Thinking Method Used:
  - Pop to auxiliary stack until word found, discard, then rebuild.

5. updateWordStack(TStack *stk, char *word, char *syne, char *anton)

- Inputs:
  - stk: Stack.
  - word, new syne, new anton.
- Outputs:
  - Stack with updated node.
- Implementation Difficulties:
  - Finding and updating element within stack constraints.
- Thinking Method Used:
  - Similar pop-and-restore approach, modify when matching.

6. stackToQueue(TStack *stk)

- Inputs:
  - stk: Stack returned by toStack.
- Outputs:
  - TQueue*: Queue with elements in sorted order (stack's natural LIFO order).
- Implementation Difficulties:
  - Converting LIFO order to FIFO in a new structure.
- Thinking Method Used:
  - Pop all elements, enqueue to queue, then optionally restore stack.

7. StacktoList(TStack *stk)

- Inputs:
  - stk: Stack returned by toStack.
- Outputs:
  - TList*: Doubly linked list sorted based on stack order.
- Implementation Difficulties:
  - Maintaining list pointers while building from stack.
- Thinking Method Used:
  - Pop each element and append as node in list.

8. addWordStack(TStack *stk, char *word, char *syne, char *anton)

- Inputs:
  - stk: Sorted stack.
  - New word, syne, anton.
- Outputs:
  - Updated stack with new element in sorted position.
- Implementation Difficulties:
  - Inserting into correct sorted location using only stack ops.
- Thinking Method Used:
  - Temporarily remove elements until insertion point, push new, restore.

9. syllableStack(TStack *stk)

- Inputs:
  - stk: Stack of words.
- Outputs:
  - Stack sorted by syllable count, separated by sentinel entries.
- Implementation Difficulties:
  - Counting syllables and preserving grouping in stack context.
- Thinking Method Used:
  - Use auxiliary structures to categorize then rebuild stack.

10. prounounciationStack(TStack *stk)

- Inputs:
  - stk: Stack of words.
- Outputs:
  - Three stacks (short, long, diphthong) with categorized entries.
- Implementation Difficulties:
  - Identifying phonetic type and sorting within stack.
- Thinking Method Used:
  - Classify during pop, push to respective stacks.

11. getSmallest(TStack *stk)

- Inputs:
  - stk: Stack.
- Outputs:
  - char*: Smallest word in lexicographical order.
- Implementation Difficulties:
  - Scanning entire stack without destruction.
- Thinking Method Used:
  - Pop to auxiliary stack while tracking minimum, then restore.

12. cycleSearch(TStack *stk)

- Inputs:
  - stk: Stack containing word-synonym/antonym mappings.
- Outputs:
  - Prints any cycles where following mappings loops.
- Implementation Difficulties:
  - Detecting cycles across mapped pairs within stack.
- Thinking Method Used:
  - Convert stack to map/dictionary then Floyd's algorithm on mapping links.

13. isPalyndromeStack(char *word)

- Inputs:
  - word: String to check.
- Outputs:
  - bool: True if palindrome, else false.
- Implementation Difficulties:
  - Using stack operations for character comparison.
- Thinking Method Used:
  - Push all chars, then pop and compare to original sequence.

14. StackRev(TStack *stk)

- Inputs:
  - stk: Stack to reverse.
- Outputs:
  - Reversed stack.
- Implementation Difficulties:
  - Recursion depth and stack content integrity.
- Thinking Method Used:
  - Recursively pop and reinsert at bottom for reversal.

Modules based on Binary Search Tree (BST)

1. toTree(TStack *stk)

- Inputs:
  - stk: Stack of merged word entries.
- Outputs:
  - TTree*: Root of BST containing all stack entries keyed by word or timestamp.
- Implementation Difficulties:
  - Maintaining BST ordering when inserting from stack (LIFO to sorted tree).
- Thinking Method Used:
  - Pop each stack element and insert into BST using standard insert logic.

2. fillTree(File *f)

- Inputs:
  - File pointer f containing word entries with synonyms and antonyms.
- Outputs:
  - TTree*: BST filled with nodes for each word in the file.
- Implementation Difficulties:
  - Parsing file and performing dynamic inserts without skewing tree too much.
- Thinking Method Used:
  - Read file line by line and insert into BST immediately.

3. getInfWordTree(TTree *tr, char *word)

- Inputs:
  - tr: Root of BST.
  - word: Target word to lookup.
- Outputs:
  - Prints or returns: synonym, antonym, character count, vowel count.
- Implementation Difficulties:
  - BST search logic for string keys.
- Thinking Method Used:
  - Standard BST search comparing word to node keys.

4. AddWordBST(TTree *tr, char *word, char *syne, char *anton)

- Inputs:
  - tr: Current BST root.
  - word, new syne, new anton strings.
- Outputs:
  - TStack* or updated tree reference: Newly inserted node returned or entire stack of added path.
- Implementation Difficulties:
  - Resolving tree rebalancing if needed (not automatically balanced).
- Thinking Method Used:
  - Standard BST insertion; optionally push path onto stack if returning a stack.

5. deleteWordBST(TTree *tr, char *word)

- Inputs:
  - tr: BST root.
  - word: Key to delete.
- Outputs:
  - TTree*: New root after deletion.
- Implementation Difficulties:
  - Handling three deletion cases: leaf, one child, two children (in-order successor).
- Thinking Method Used:
  - Recursive delete, find replacement if two children.

6. UpdateWordBST(TTree *tr, char *word, char *syne, char *anton)

- Inputs:
  - tr: BST root.
  - word: Key to update.
  - syne, anton: New values.
- Outputs:
  - TTree*: BST with updated node data.
- Implementation Difficulties:
  - Locating node and modifying its payload without altering tree structure.
- Thinking Method Used:
  - Search then update payload in-place.

7. TraversalBSTinOrder(TTree *tr)

- Inputs:
  - tr: BST root.
- Outputs:
  - List or array of nodes in in-order sequence.
- Implementation Difficulties:
  - Deep recursion on skewed trees.
- Thinking Method Used:
  - Recursive left-root-right traversal.

8. TraversalBSTpreOrder(TTree *tr)

- Inputs:
  - tr: BST root.
- Outputs:
  - Nodes in pre-order (root-left-right).
- Implementation Difficulties:
  - Same recursion depth concerns.
- Thinking Method Used:
  - Recursive root-left-right traversal.

9. TraversalBSTpostOrder(TTree *tr)

- Inputs:
  - tr: BST root.
- Outputs:
  - Nodes in post-order (left-right-root).
- Implementation Difficulties:
  - Same recursion depth concerns.
- Thinking Method Used:
  - Recursive left-right-root traversal.

10. HighSizeBST(TTree *tr)

- Inputs:
  - tr: BST root.
- Outputs:
  - Prints the height and size (node count) of the tree.
- Implementation Difficulties:
  - Efficiently computing both metrics in a single traversal.
- Thinking Method Used:
  - Recursive traversal returning height and accumulating node count.

11. LowestCommonAncestor(TTree *tr, char *word1, char *word2)

- Inputs:
  - tr: BST root.
  - word1, word2: Two keys.
- Outputs:
  - TTree*: Node pointer to LCA.
- Implementation Difficulties:
  - Ensuring both keys exist; handling edge cases.
- Thinking Method Used:
  - BST property: traverse from root, branching based on comparisons.

12. CountNodesRanges(TTree *tr, int l, int h)

- Inputs:
  - tr: BST root.
  - l, h: Numeric range bounds.
- Outputs:
  - int: Count of nodes with keys in [l, h].
- Implementation Difficulties:
  - Pruning subtrees outside range to optimize.
- Thinking Method Used:
  - Recursive traversal with range checks and subtree skipping.

13. inOrderSuccessor(TTree *tr, char *word)

- Inputs:
  - tr: BST root.
  - word: Reference key.
- Outputs:
  - TTree*: Node pointer to in-order successor.
- Implementation Difficulties:
  - Two cases: node has right subtree vs. climbing ancestors.
- Thinking Method Used:
  - If right child exists, find leftmost descendant; else track ancestor.

14. BSTMirror(TTree *tr)

- Inputs:
  - tr: BST root.
- Outputs:
  - TTree*: Root of mirrored tree.
- Implementation Difficulties:
  - Creating new nodes or swapping children in-place.
- Thinking Method Used:
  - Recursive swap of left and right subtrees.

15. isBalancedBST(TTree *tr)

- Inputs:
  - tr: BST root.
- Outputs:
  - bool: True if tree is height-balanced.
- Implementation Difficulties:
  - Checking heights of subtrees without repeated traversal.
- Thinking Method Used:
  - Post-order recursion returning height and balanced flag.

16. BSTMerge(TTree *tr1, TTree *tr2)

- Inputs:
  - tr1, tr2: Two BST roots.
- Outputs:
  - TTree*: Root of merged, balanced BST.
- Implementation Difficulties:
  - Merging tree structures without duplicates and balancing.
- Thinking Method Used:
  - Traverse both trees to sorted arrays, merge arrays, build balanced BST.

Modules Based on Recursion

1. countWordOccurence(File *f, char *word)

- Inputs:
  - File pointer f.
  - Target word to count.
- Outputs:
  - int: Number of occurrences of word in file.
- Implementation Difficulties:
  - Reading file recursively line by line or word by word.
  - Managing end-of-file base case.
- Thinking Method Used:
  - Recursive read: on each call, read next token, compare and increment count, then recurse until EOF.

2. removeWordOccurence(File *f, char *word)

- Inputs:
  - File pointer f.
  - word to remove.
- Outputs:
  - File*: New file object with all occurrences removed.
- Implementation Difficulties:
  - Recursive file rewrite while preserving other content.
- Thinking Method Used:
  - Recursively read tokens, write tokens not equal to word to a new file, return when EOF.

3. replaceWordOccurence(File *f, char *word, char *rep)

- Inputs:
  - File pointer f.
  - word to replace.
  - Replacement string rep.
- Outputs:
  - File*: New file object with replacements applied.
- Implementation Difficulties:
  - Ensuring recursion handles end-of-file and writes replacements correctly.
- Thinking Method Used:
  - Recursive read and write: if token matches word, write rep; else write original.

4. wordPermutation(char *word)

- Inputs:
  - word: Input string.
- Outputs:
  - Prints all permutations of word.
- Implementation Difficulties:
  - Managing character swaps and backtracking efficiently.
- Thinking Method Used:
  - Recursive swapping: for each index, swap with current position, recurse, then backtrack.

5. subseqWord(char *word)

- Inputs:
  - word: Input string.
- Outputs:
  - Prints all subsequences of word.
- Implementation Difficulties:
  - Generating 2^n subsequences without duplication.
- Thinking Method Used:
  - Recursive include/exclude decision for each character.

6. longestSubseqWord(char *word1, char *word2)

- Inputs:
  - Two strings word1, word2.
- Outputs:
  - int: Length of longest common subsequence.
- Implementation Difficulties:
  - Exponential time complexity without memoization.
- Thinking Method Used:
  - Recursive LCS: if chars match, +1 + recurse; else max of two recursive calls.

7. distinctSubseqWord(char *word)

- Inputs:
  - word: Input string.
- Outputs:
  - int: Number of distinct subsequences.
- Implementation Difficulties:
  - Avoiding recounting duplicates; exponential growth in recursion.
- Thinking Method Used:
  - Recursive inclusion/exclusion with a data structure to track seen subsequences or dynamic programming.

8. isPalindromWord(char *word)

- Inputs:
  - word: Input string.
- Outputs:
  - bool: True if word is palindrome.
- Implementation Difficulties:
  - Recursion base cases for length 0 or 1.
- Thinking Method Used:
  - Compare first and last chars, then recurse on substring excluding them. (TTree *tr1, TTree *tr2)
- Inputs:
  - tr1, tr2: Two BST roots.
- Outputs:
  - TTree*: Root of merged, balanced BST.
- Implementation Difficulties:
  - Merging tree structures without duplicates and balancing.
- Thinking Method Used:
  - Traverse both trees to sorted arrays, merge arrays, build balanced BST.

Optional Part: Graphical User Interface (GUI)

Optional Part: Graphical User Interface (GUI)

To enhance user interaction, an optional GUI can be developed in C using libraries as GTK+:

1. **Event-Driven Architecture**
   - Utilize GTK+ signals to handle menu selections, button clicks, and input fields.
2. **Widgets and Layouts**
   - Design windows, dialogs, list views, and tree views to display logs and dictionary entries.
   - Employ boxes and grids for organized placement of controls.
3. **Data Binding**
   - Link C data structures (linked lists, trees, queues) with GTK+ models (e.g., GtkListStore, GtkTreeStore).
   - Update views dynamically when data changes.
4. **User Input Forms**
   - Entry widgets for word searches, insertions, and updates.
   - Validation and feedback through dialogs or status bars.
5. **Cross-Platform Compatibility**
   - GTK+ supports Linux, Windows, and macOS, facilitating portable deployment.

These GUI modules complement the core data structure implementations, providing a user-friendly interface for interacting with the logs management and dictionary systems.

Conclusion

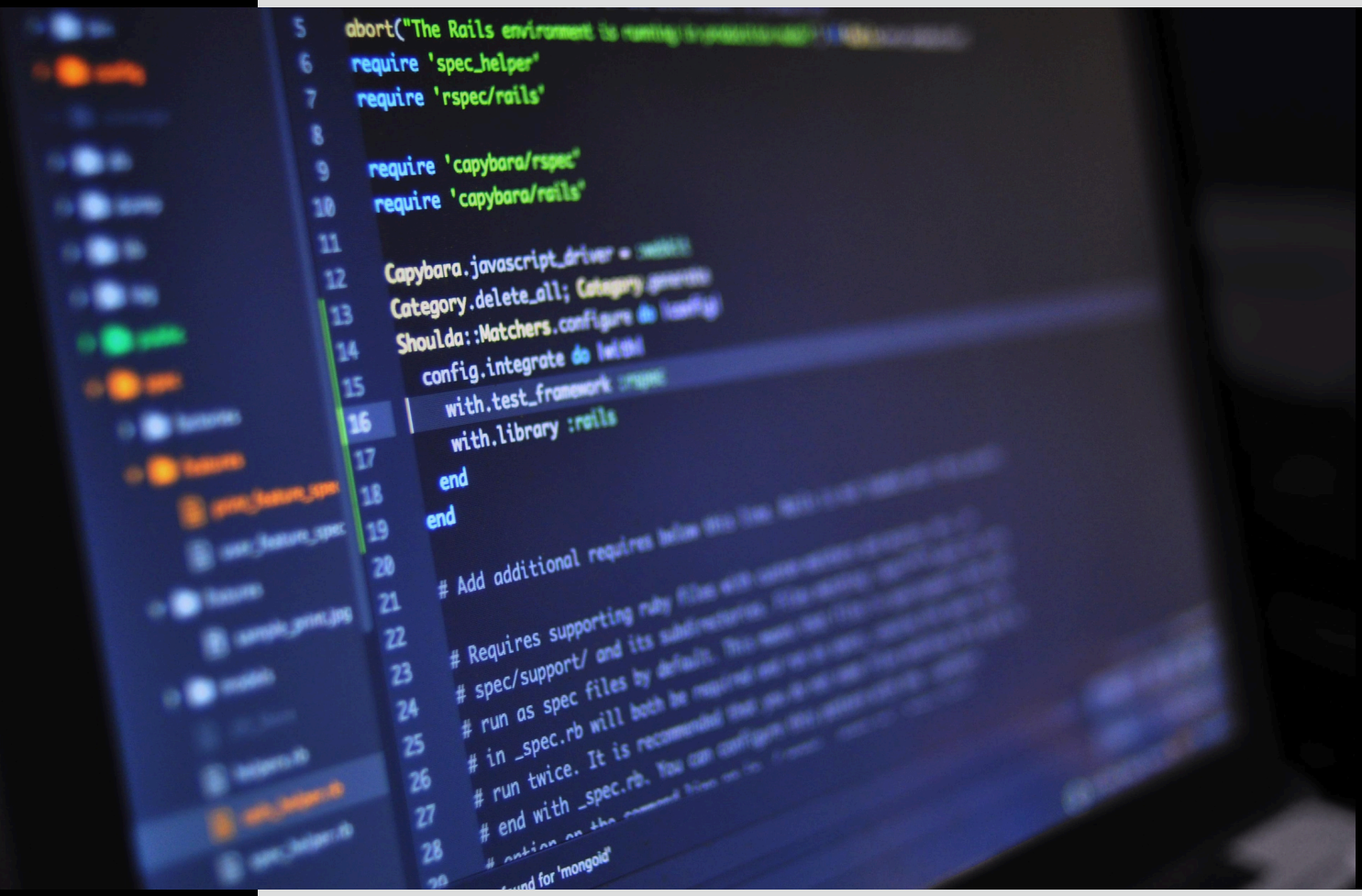
This project successfully demonstrates the implementation and integration of fundamental data structures and algorithms in C to address two distinct domains: security log management and dictionary management. Key achievements include :

- We built and analyzed singly, doubly, and circular linked lists , stacks , queues , recursive algorithms , binary search trees ... Each module was crafted with careful attention to pointer manipulation, memory management, and algorithmic correctness.
- The security logs system provides full CRUD operations, traversal in multiple orders, cycle detection, and priority handling. It demonstrates real-world utility in monitoring and maintaining system integrity.
- The dictionary management module leverages file parsing, data structure conversions, and recursive string algorithms to offer synonym and antonym lookups, sorting, filtering, and advanced tree-based searches. The menu-driven interface empowers users to manipulate linguistic data intuitively.
- For each operation, we assessed time and space complexities, highlighting trade-offs between linear versus recursive approaches, pointer-based versus array-based storage, and balanced versus unbalanced tree structures.
- The proposed GTK+ interface outlines a path toward a responsive, cross-platform, event-driven application, bridging low-level C implementations with modern user experiences.

Overall, this project underscores the essential role of data structures and algorithms in building efficient, scalable, and maintainable software. It provides a solid foundation for further exploration into advanced topics such as self-balancing trees, concurrent data structures, and performance optimization in C.

END SEMESTER PROJECT

NSCS
المدرسة الوطنية العليا في الأمن السيبراني
NATIONAL SCHOOL OF CYBERSECURITY



Thank you !

For further information regarding this project, please contact us :

f.hassani@enscs.edu.dz
m.abbaci@enscs.edu.dz
a.kassoul@enscs.edu.dz