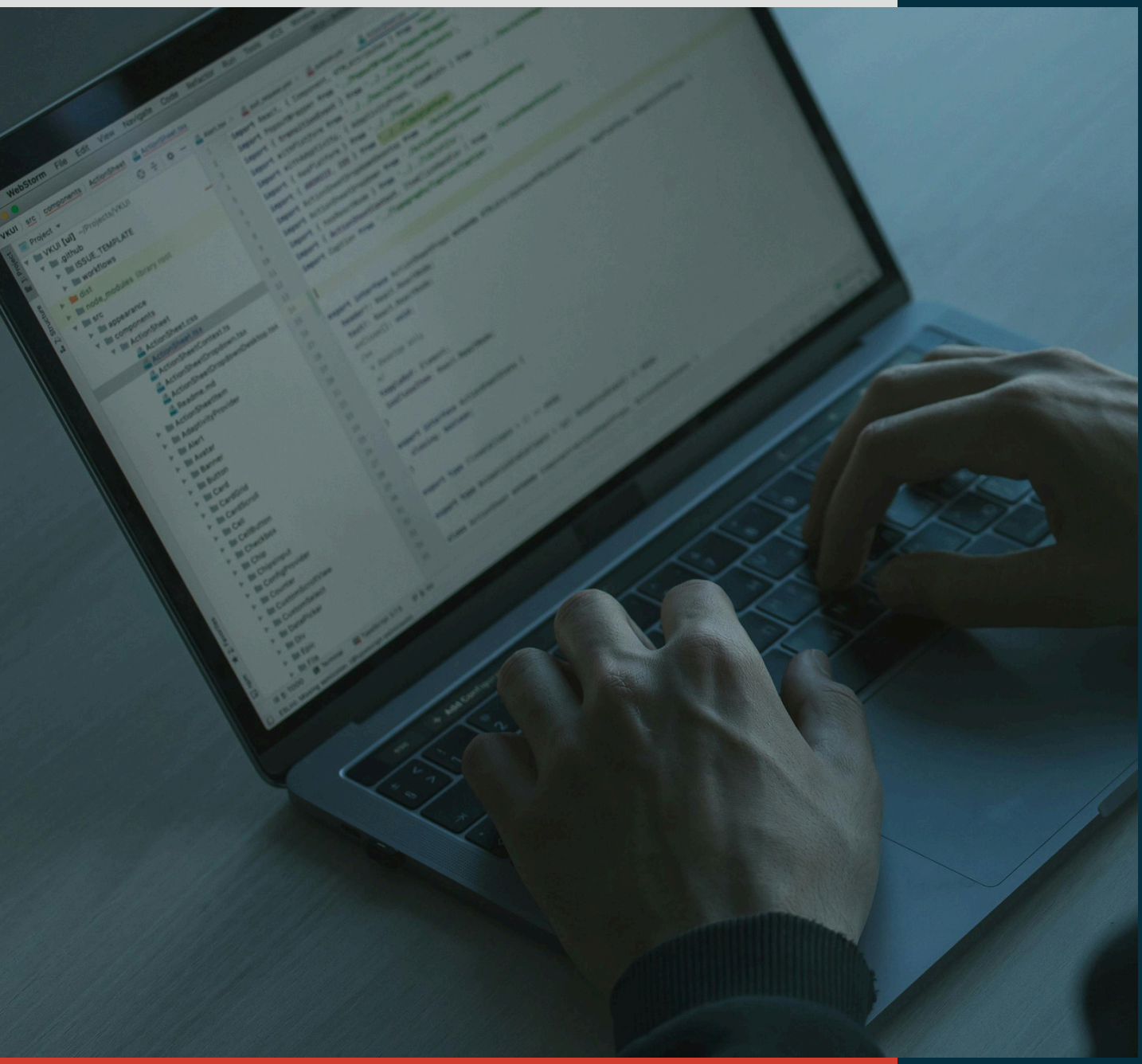
SECOND SEMESTER PROJECT

NATIONAL SCHOOL OF CYBER SCHOOL



Prepared by
MEKCI RAIHAN
NOURREDDIN MALAK

MAY 2025

Table of Contents

Linked Linear Lists

→ Insertion Functions

insertAtBeginning()

- Purpose: Add log to start of list
- Input: Head pointer, new log data
- Steps:
  a. Create new node
  b. Point new node to current head
  c. Update head to new node

insertAtEnd()

- Purpose: Add log to end of list
- Input: Head pointer, new log data
- Steps:
  a. Create new node
  b. Traverse to last node
  c. Link last node to new node

insertAfterPosition()

- Purpose: Insert log at specific position
- Input: Head pointer, new log data, position number
- Steps:
  a. Verify valid position
  b. Traverse to position
  c. Insert new node

Linked Linear Lists

→ Deletion Functions

insertAtBeginning()

- Purpose: Remove log by ID
- Input: Head pointer, target ID
- Steps:
  a. Find matching ID
  b. Adjust neighbor pointers
  c. Free memory

insertAtEnd()

- Purpose: Remove first log
- Input: Head pointer
- Steps:
  a. Point head to second node
  b. Free original head

Linked Linear Lists

→ Search&Utility Functions

searchByID()

- Purpose: Find log by ID
- Input: Head pointer, target ID
- Output: Found node or NULL
- Steps: Linear search through list

sortByDate()

- Purpose: Sort logs chronologically
- Input: Head pointer
- Steps: Bubble sort comparing timestamps

reverse()

- Purpose: Reverse log order
- Input: Head pointer
- Steps: Iteratively flip node pointers

Queues

→ Queue Implementation

enqueueLog()

- Purpose: Add log to queue end
- Input: Queue pointer, new log data
- Steps:
  a. Create new node
  b. Add to rear (or set as first if empty)
  c. Update rear pointer

dequeueLog()

- Purpose: Remove log from queue front
- Input: Queue pointer
- Steps:
  a. Check if empty
  b. Remove front node
  c. Update front pointer
  d. Free memory

Queues

→ Queue Implementation

peekLog()

- Purpose: View front log without removing
- Input: Queue pointer
- Steps:
  - Check if empty
  - Display front log details

isEmptyLinkedList()

- Purpose: Check if queue is empty
- Input: Queue pointer
- Output: True if empty, False otherwise

Queues

→ Circular Queue Implementation

The circular queue is a fixed-size FIFO (First-In-First-Out) data structure that efficiently reuses empty spaces by connecting the end of the array back to its start. Key functions include:

- ***initCircularQueue():*** Initializes the queue with front and rear set to -1.
- ***enqueueCircular():*** Adds a log entry at the rear, wrapping around if space is available.
- ***dequeueCircular():*** Removes the front entry and adjusts the front pointer circularly.
- ***isEmptyCircular():*** Checks if the queue is empty (front == -1).
- ***isFullCircular():*** Checks if the queue is full (next position of rear equals front).

This avoids memory waste and supports O(1) enqueue/dequeue operations.

Trees

→ Core BST Operations

insertLog()

- Purpose: Inserts log into BST sorted by timestamp
- Input: Tree root, log data
- Steps:
  a. Create new node if tree is empty
  b. Recursively insert left (if timestamp < current) or right (if timestamp ≥ current)
  c. Returns updated tree

deleteLog()

- Purpose: Removes log with matching timestamp
- Input: Tree root, target timestamp
- Steps:
  a. Recursively search for timestamp
  b. Handles 3 cases:
    - No children → simple delete
    - One child → replace with child
    - Two children → replace with inorder successor
  c. Returns updated tree

searchLog()

- Purpose: Finds log by timestamp
- Input: Tree root, target timestamp
- Steps:
  a. Recursive binary search
  b. Returns matching node or NULL

Trees

→ Tree Traversals

inOrderTraversal()

- Output: Logs sorted by timestamp (ascending)
- Process: Left → Root → Right

preOrderTraversal()

- Output: Root-first order
- Process: Root → Left → Right

postOrderTraversal()

- Output: Leaves-first order
- Process: Left → Right → Root

Trees

→ Linked List Conversion

findMiddle()

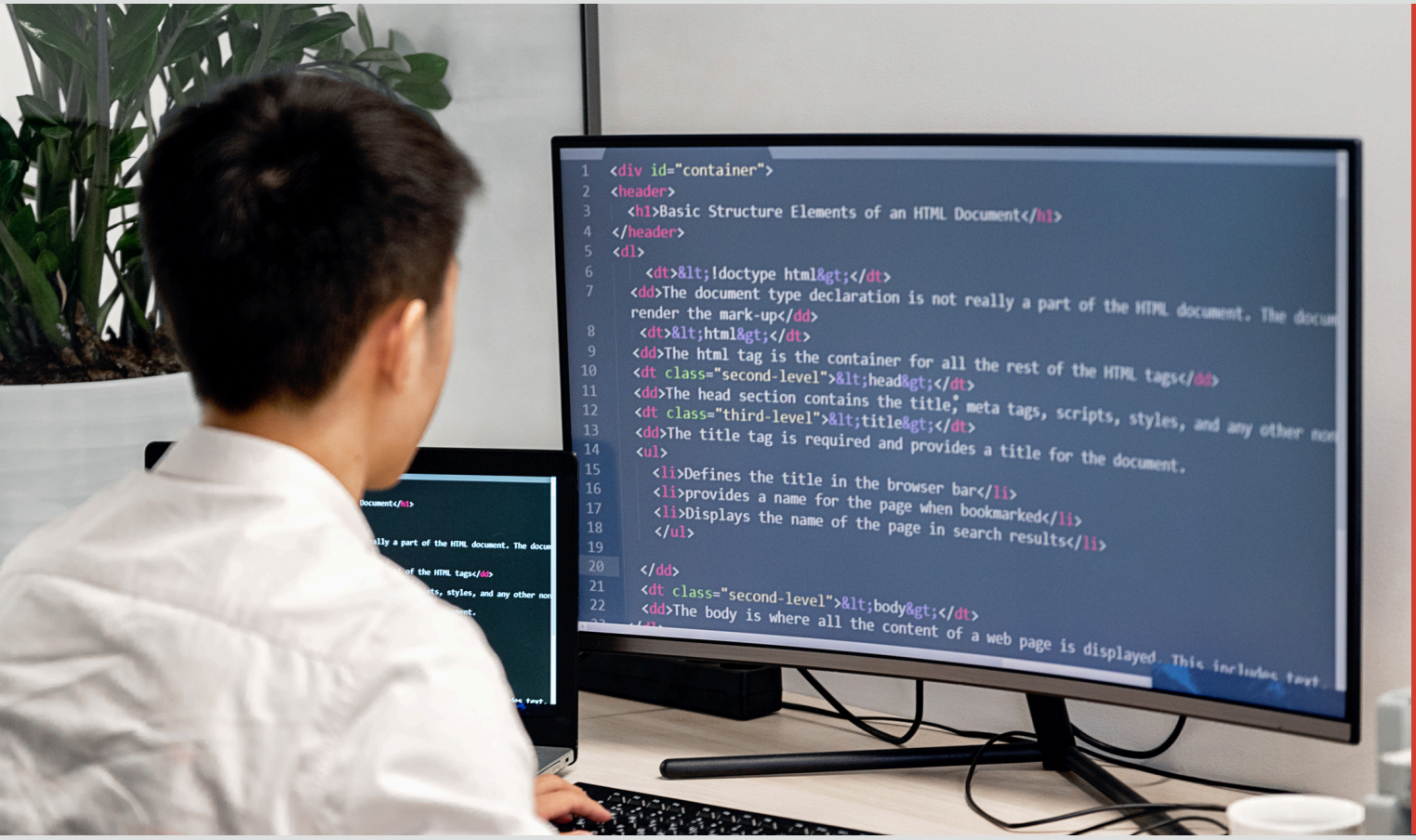
- Purpose: Finds middle node of linked list
- Method: Fast/slow pointer technique

sortedListToBST()

- Purpose: Converts sorted linked list to balanced BST
- Steps:
  a. Find middle node (becomes root)
  b. Recursively build left/right subtrees
  c. Returns BST root

Second part

- Linked lists
  Queues
- Recursion
- Trees
- Stacks



Word List Creation

TLIST *GETSYNWORDS(FILE *F)
PURPOSE: CREATES LINKED LIST OF WORDS WITH SYNONYMS
NODE CONTENTS: WORD, SYNONYM, CHAR COUNT, VOWEL COUNT
PROCESS: READS FILE AND POPULATES LIST

TLIST *GETANTOWORDS(FILE *F)
PURPOSE: CREATES LINKED LIST OF WORDS WITH ANTONYMS
NODE CONTENTS: WORD, ANTONYM, CHAR COUNT, VOWEL COUNT
PROCESS: READS FILE AND POPULATES LIST

Word Information Retrieval

VOID GETINFWORD(TLIST *SYN, TLIST *ANT, CHAR *WORD)

- PURPOSE: GETS WORD INFORMATION
- OUTPUT: SYNONYM, ANTONYM, CHAR/VOWEL COUNTS
- PROCESS: SEARCHES BOTH LISTS FOR MATCHING WORD

VOID GETINFWORD2(TLIST *SYN, TLIST *ANT, CHAR *INF)

- *PURPOSE: FINDS ORIGINAL WORD BY SYNONYM/ANTONYM*
- *OUTPUT: ORIGINAL WORD WITH CHAR/VOWEL COUNTS*
- *PROCESS: SEARCHES LISTS FOR MATCHING SYNONYM/ANTONYM*

Sorting Functions

TLIST *SORTWORD(TLIST *SYN)

- *PURPOSE: ALPHABETICAL SORT (A-Z)*
- *METHOD: BUBBLE/MERGE SORT ON WORD STRINGS*

TLIST *SORTWORD2(TLIST *SYN)

- *PURPOSE: SORT BY CHARACTER COUNT (ASCENDING)*
- *METHOD: COMPARES NODE CHAR COUNTS*

TLIST *SORTWORD3(TLIST *SYN)

- *PURPOSE: SORT BY VOWEL COUNT (DESCENDING)*
- *METHOD: COMPARES NODE VOWEL COUNTS*

Word Management

TLIST *DELETEWORD(FILE *F, TLIST *SYN, TLIST *ANT, CHAR *WORD)

- *PURPOSE: DELETES WORD FROM FILE AND BOTH LISTS*
- *STEPS:*
  - A. *REMOVES FROM FILE*
  - B. *DELETES MATCHING NODES FROM BOTH LISTS*
  - C. *RETURNS UPDATED LISTS*

TLIST *UPDATEWORD(FILE *F, TLIST *SYN, TLIST *ANT, CHAR *WORD, CHAR *SYNE, CHAR *ANTON)

- *PURPOSE: UPDATES WORD'S SYNONYM/ANTONYM*
- *STEPS:*
  - A. *MODIFIES FILE*
  - B. *UPDATES BOTH LINKED LISTS*
  - C. *RETURNS UPDATED LISTS*

TLIST *SIMILARWORD(TLIST *SYN, CHAR *WORD, FLOAT RATE)

- *PURPOSE: FINDS SIMILAR WORDS BY MATCH RATE*
- *OUTPUT: LIST OF WORDS WITH SIMILARITY ≥ RATE*
- *METHOD: STRING COMPARISON ALGORITHM (E.G., LEVENSHTEIN DISTANCE)*

Word Search and Analysis

TLIST *COUNTWORD(TLIST *SYN, CHAR *PRT)

- *PURPOSE: FINDS ALL WORDS CONTAINING SUBSTRING PRT*
- *PROCESS:*
  - *SCANS EACH WORD IN SYNONYM LIST*
  - *USES STRSTR() TO CHECK FOR SUBSTRING MATCHES*
  - *RETURNS FILTERED LIST OF MATCHING WORDS WITH THEIR FULL INFORMATION*

TLIST *PALINDROMWORD(TLIST *SYN)

- PURPOSE: IDENTIFIES AND SORTS PALINDROME WORDS
- PROCESS:
  A. CHECKS EACH WORD FOR PALINDROME PROPERTY (SAME FORWARDS/BACKWARDS)
  B. INSERTS PALINDROMES INTO NEW LIST IN SORTED ORDER
  C. MAINTAINS ALL ORIGINAL WORD INFORMATION (SYNONYM, COUNTS)
  D. RETURNS SORTED PALINDROME LIST

List Merging Operations

TLIST *MERGE(TLIST *SYN, TLIST *ANT)

- PURPOSE: CREATES BIDIRECTIONAL LIST FROM TWO INPUT LISTS
- PROCESS:
  - COMBINES SYNONYM AND ANTONYM NODES INTO SINGLE NODES
  - SETS BOTH NEXT AND PREVIOUS POINTERS
  - PRESERVES ALL WORD DATA (WORD, SYNONYM, ANTONYM, COUNTS)
  - RETURNS HEAD OF NEW BIDIRECTIONAL LIST

TLIST *MERGE2(TLIST *SYN, TLIST *ANT)

- PURPOSE: CREATES CIRCULAR LIST FROM TWO INPUT LISTS
- PROCESS:
  - SIMILAR TO MERGE() BUT MAKES LIST CIRCULAR
  - LAST NODE POINTS BACK TO HEAD
  - MAINTAINS ALL WORD INFORMATION
  - RETURNS HEAD OF CIRCULAR LIST

STUDIO
SHODWE

Word Management Extensions

TLIST *ADDWORD(TLIST *SYN, TLIST *ANT, CHAR *WORD, CHAR *SYNE, CHAR *ANTON)

- PURPOSE: ADDS NEW WORD ENTRY TO SYSTEM
- PROCESS:
  A. APPENDS WORD TO TEXT FILE
  B. ADDS TO SYNONYM LIST (WORD + SYNONYM)
  C. ADDS TO ANTONYM LIST (WORD + ANTONYM)
  D. CALCULATES AND STORES CHAR/VOWEL COUNTS
  E. RETURNS UPDATED LISTS

STUDIO
SHODWE

Queue Processing Functions

TQUEUE *SYLLABLE(TLIST *SYN)

- PURPOSE: ORGANIZES WORDS BY SYLLABLE COUNT
- PROCESS:
  - COUNTS SYLLABLES IN EACH WORD
  - SORTS WORDS INTO QUEUE SECTIONS BY SYLLABLE COUNT
  - SEPARATES GROUPS WITH EMPTY MARKERS
  - RETURNS ORGANIZED QUEUE

TQUEUE *PROUNOUNCIATION(TLIST *SYN)

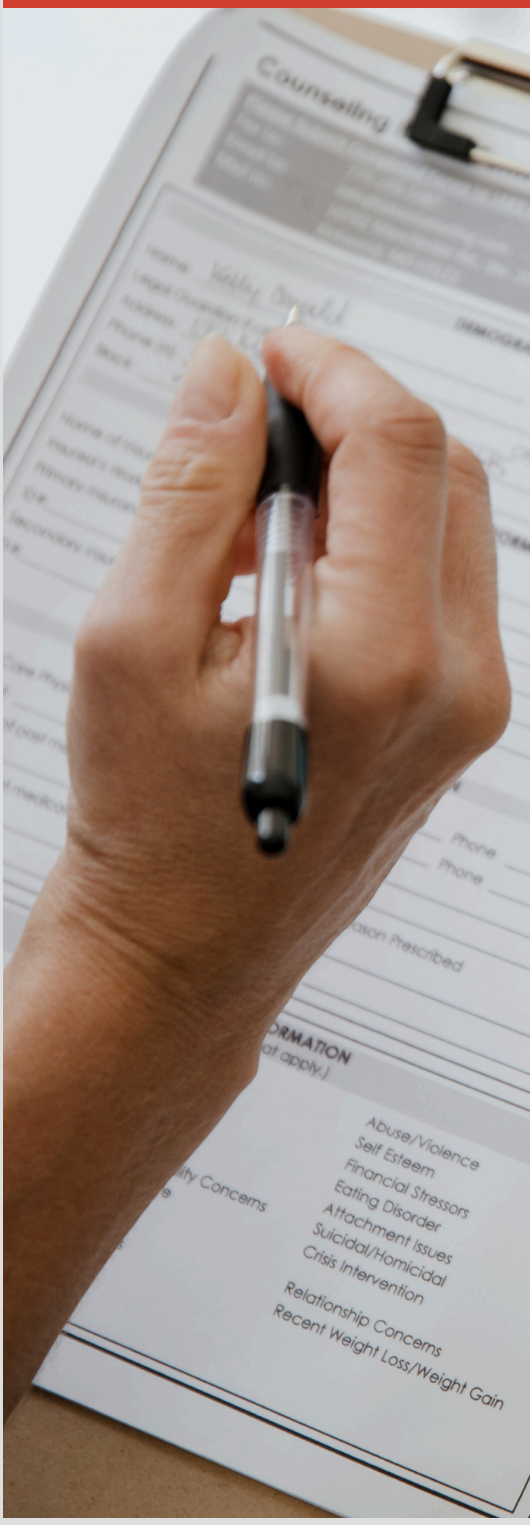
- PURPOSE: CATEGORIZES WORDS BY PRONUNCIATION
- PROCESS:
  - ANALYZES EACH WORD'S PRONUNCIATION PATTERN
  - SORTS INTO THREE QUEUES:
    I. SHORT VOWELS (1 MORA)
    II. LONG VOWELS (2 MORAE)
    III. DIPHTHONGS (COMPLEX VOWELS)
  - RETURNS TRIPLE QUEUE STRUCTURE

TQUEUE *TOQUEUE(TLIST*MERGED)

- PURPOSE: CONVERTS MERGED LIST TO QUEUE
- PROCESS:
  - TAKES OUTPUT FROM MERGE() FUNCTION
  - CREATES FIFO QUEUE STRUCTURE
  - PRESERVES ALL WORD INFORMATION
  - RETURNS QUEUE HEAD POINTER

STUDIO
SHODWE

Stack Conversion & Basic Operations



TStack *toStack(TList *merged)

- Purpose: Converts merged bidirectional list to stack (LIFO)
- Process: Pushes each node from list onto stack