ALDD PROJECT PRESENTATION

Presented by :

BENGUEBBOUR MOHAMMED TEDJ EDDINE

BELABBES AMINE

HADJ HACENE FAROUK

Project Explanation

This project focuses on implementing fundamental data structures, such as chars, strings, linked lists, stacks,queues, recursion, and trees, and analyzing their complexity. It mainly consists of two main parts

Catalog

First part :

- Linked Linear Lists
- Bidirectional Linked Lists
- Circular Linked Lists
- Queues
- Stacks
- Recursion
- Trees

Second part :

- Modules based on Linked lists and Queues
- Modules based on Stacks
- Modules based on Binary Search Tree
- Modules based on Recursion

First Part

Linked Linear Lists

- insertAtBeginning
- insertAtEnd
- insertAtPosition
- deleteByID
- deleteByTimestamp
- deleteFirst
- deleteLast

- searchByID
- searchByKeyword
- searchByTimestamp
- sortByDate
- reverseList
- countLogs

insertAtBeginning

```
LogEntry* insertAtBeginning(LogEntry* head, int id, const char* timestamp, int severity, const char* message)
```

***Purpose*:**

Adds a new log at the start of the list.

Steps:

- Create a new node using createNode.
- Set newNode->next to current head.
- Return newNode as the new head.

insertAtEnd

```
LogEntry* insertAtEnd(LogEntry* head, int id, const char* timestamp, int severity, const char* message)
```

***Purpose*:**

Adds a new log at the end of the list.

Steps:

- Create a new node using createNode.
- If list is empty, return new node.
- Traverse to the last node.
- Set lastNode->next = newNode.
- Return the head.

insertAtPosition

```
LogEntry* insertAtPosition(LogEntry* head, int position, int id, const char* timestamp, int severity, const char* message)
```

Purpose:

Inserts a new log at a specified index in the list.

Steps:

- If position is 0 or less, insert at beginning.
- Traverse to the node just before the desired position.
- Create a new node.
- Link new node’s next to the next node.
- Link previous node’s next to new node.
- Return the head.

deleteByID

```
LogEntry* deleteByID(LogEntry* head, int id)
```

***Purpose*:**

Deletes a log from the list based on its ID.

Steps:

- If head matches the ID, free it and return head->next.
- Otherwise, search the node whose next matches the ID.
- Relink and free the matching node.
- Return the updated head.

deleteByTimestamp

```
LogEntry* deleteByTimestamp(LogEntry* head, const char* timestamp)
```

***Purpose*:**

Deletes the first log that matches the given timestamp.

Steps:

- Check if the head node has the timestamp.
- If so, remove and return next.
- Otherwise, traverse and look for matching timestamp.
- Relink and free that node.
- Return the head.

deleteFirst

```
LogEntry* deleteFirst(LogEntry* head)
```

***Purpose*:**

Deletes the first node in the list.

Steps:

- Save pointer to head->next.
- Free the head.
- Return the saved pointer.

deleteLast

LogEntry* deleteLast(LogEntry* head)

***Purpose*:**

Deletes the last node in the list.

Steps:

- If list is empty or has one node, free it and return NULL.
- Traverse to second-last node.
- Free last node and set secondLast->next = NULL.
- Return the head.

searchByID

```
LogEntry* searchByID(LogEntry* head, int id)
```

***Purpose*:**

Finds the first node with the given ID.

Steps:

- Traverse the list comparing each node's id.
- Return the node if found, otherwise NULL.

searchByKeyword

LogEntry* searchByKeyword(LogEntry* head, const char* keyword)

Purpose:

Finds the first log entry that contains the keyword in its message.

Steps:

- Traverse the list.
- Use strstr to check if keyword is in message.
- Return the first matching node or NULL.

searchByTimestamp

Finds thLogEntry* searchByTimestamp(LogEntry* head, const char* timestamp)e first log with the specified timestamp.

***Purpose*:**

Finds the first log with the specified timestamp.

Steps:

- Traverse the list comparing timestamps using strcmp.
- Return the matching node or NULL.

sortByDate

`LogEntry* sortByDate(LogEntry* head)`

***Purpose*:**

Sorts the logs based on their timestamps using bubble sort logic.

Steps:

- Nested loop through the list.
- Compare timestamps using strcmp.
- If out of order, swap values (not pointers).
- Repeat until sorted.

sortBySeverity

LogEntry* sortBySeverity(LogEntry* head)

***Purpose*:**

Sorts the logs based on severity level in ascending order.

Steps:

- Nested loop through the list.
- Compare severity values.
- Swap values of nodes if needed.

reverseList

LogEntry* reverseList(LogEntry* head)

***Purpose*:**

Reverses the order of the list.

Steps:

- Use three pointers: prev, curr, and next.
- Iterate and reverse next links.
- Return the new head (prev).

countLogs

int countLogs(LogEntry* head)

***Purpose*:**

Counts how many logs (nodes) are in the list.

Steps:

- Initialize a counter.
- Traverse the list, incrementing counter.
- Return final count.

First Part

Bidirectional Linked Lists

- insertAtBeginningD
- insertAtEndD
- insertAtPositionD
- deleteByIDD
- deleteFirstD
- deleteLastD
- deleteMiddleNode
- searchByIDD
- searchByTimestampD

- sortByDateD
- reverseListD
- countLogsD
- moveForward
- moveBackward
- mergeDLists

insertAtBeginningD

```
DLogEntry* insertAtBeginningD(DLogEntry* head, int id, const char* timestamp, int severity, const char* message)
```

***Purpose*:**

Insert a node at the beginning of the doubly linked list.

Steps:

- Create new node.
- Point new node’s next to current head.
- Update old head’s prev (if exists).
- Return new node as the new head.

insertAtEndD

```
DLogEntry* insertAtEndD(DLogEntry* head, int id, const char* timestamp, int severity, const char* message)
```

Purpose:

Insert a node at the end of the doubly linked list.

Steps:

- If list is empty, return new node.
- Traverse to the last node.
- Insert new node after it.
- Set prev of new node to last node.

insertAtPositionD

```
DLogEntry* insertAtPositionD(DLogEntry* head, int position, int id, const char* timestamp, int severity, const char* message)
```

***Purpose*:**

Insert node at specific position in the list.

Steps:

- If list is empty or position is 0, insert at beginning.
- Traverse to the given position.
- Insert new node and update next and prev pointers.
- Update head if necessary.

deleteByIDD

```
DLogEntry* deleteByIDD(DLogEntry* head, int id)
```

***Purpose*:**

Delete node with matching ID.

Steps:

- Traverse to node with matching ID.
- Relink surrounding nodes.
- Free the node.
- Return updated head.

deleteFirstD

DLogEntry* deleteFirstD(DLogEntry* head)

***Purpose*:**

Delete the first node.

Steps:

- Move head to next node.
- Update prev of new head to NULL.
- Free old head.
- Return updated head.

deleteLastD

DLogEntry* deleteLastD(DLogEntry* head)

***Purpose*:**

Delete the last node.

Steps:

- Traverse to last node.
- Update next of previous node to NULL.
- Free the last node.
- Return updated head.

deleteMiddleNode

DLogEntry* deleteMiddleNode(DLogEntry* head)

Purpose:

Delete the middle node using slow-fast pointer technique.

Steps:

- Move slow 1 step, fast 2 steps at a time.
- When fast reaches end, slow is at the middle.
- Remove slow from the list.
- Update links and free node.
- Return updated head.

searchByIDD

```
DLogEntry* searchByIDD(DLogEntry* head, int id)
```

***Purpose*:**

Find node with matching ID.

Steps:

- Traverse the list.
- Return node if found, else NULL.

searchByTimestampD

DLogEntry* searchByTimestampD(DLogEntry* head, const char* timestamp)

***Purpose*:**

Find node with specific timestamp.

Steps:

- Traverse and compare timestamps.
- Return node if match found.

searchByKeywordD

```
DLogEntry* searchByKeywordD(DLogEntry* head, const char* keyword)
```

***Purpose*:**

Find first node containing keyword in message.

Steps:

- Traverse and use strstr to check message.
- Return first matching node.

sortByDateD

```
DLogEntry* sortByDateD(DLogEntry* head)
```

Purpose:

Sort list based on timestamps in ascending order.

Steps:

- Use nested loop to compare timestamps.
- Swap values between nodes if out of order.

sortBySeverityD

```
DLogEntry* sortBySeverityD(DLogEntry* head)
```

Purpose:

Sort list by severity in ascending order.

Steps:

- Use nested loop to compare severity.
- Swap values between nodes when needed.

reverseListD

`DLogEntry* reverseListD(DLogEntry* head)`

Purpose:

Reverse the entire doubly linked list.

Steps:

- Swap next and prev for each node.
- Update head pointer at the end.

countLogsD

int countLogsD(DLogEntry* head)

***Purpose*:**

Count total number of log entries.

***Steps*:**

- Traverse the list.
- Increment counter for each node.

moveForward

`DLogEntry* moveForward(DLogEntry* node, int steps)`

***Purpose*:**

Move forward a given number of steps.

Steps:

- While steps > 0, follow next pointer.

moveBackward

DLogEntry* moveBackward(DLogEntry* node, int steps)

***Purpose*:**

Move backward a given number of steps.

Steps:

- While steps > 0, follow prev pointer.

mergeDLists

`DLogEntry* mergeDLists(DLogEntry* list1, DLogEntry* list2)`

***Purpose*:**

Merge two doubly linked lists into one.

Steps:

- Traverse to end of list1.
- Link list1's last node to head of list2.
- Set prev of list2 head to list1's tail.

First Part

Circular Linked Lists

- insertCircular
- insertAtPositionCircular
- deleteByIDCircular
- deleteFirstCircular
- deleteLastCircular
- searchByIDCircular
- searchByTimestampCircular
- searchByKeywordCircular

- sortByDateCircular
- Sort Logs by severity level
- Sort Logs by date
- Reverse the List
- Implement a Fixed-Size Log Buffer
- Detect Cycles in the List

insertCircular

```
CLogEntry* insertCircular(CLogEntry* tail, int id, const char* timestamp, int severity, const char* message)
```

***Purpose*:**

Insert a new node at the end (after tail) of the circular list.

Steps:

- Create a new node.
- If list is empty:
  - Point node to itself.
  - Return it as new tail.
- Otherwise:
  - Insert after tail.
  - Update tail to new node.

insertAtPositionCircular

```
CLogEntry* insertAtPositionCircular(CLogEntry* tail, int position, int id, const char* timestamp, int severity, const char* message
```

Steps:

- If list is empty or position is 0, insert at beginning.
- Traverse to desired index.
- Insert node in that position.
- Update tail if inserted at the end.

Purpose:

Insert a new node at a specified position in the circular list.

deleteByIDCircular

```
CLogEntry* deleteByIDCircular(CLogEntry* tail, int id)
```

Purpose:

Delete the node with a specific ID.

Steps:

- Traverse while comparing IDs.
- If match:
  - Unlink the node.
  - If node is the tail or the only node, update tail accordingly.
  - Free the node.
- Return updated tail.

deleteFirstCircular

```
CLogEntry* deleteFirstCircular(CLogEntry* tail)
```

***Purpose*:**

Delete the first node (head) in the circular list.

Steps:

- If only one node, free and return NULL.
- Otherwise:
  - Remove tail->next.
  - Point tail to new head.

deleteLastCircular

`CLogEntry* deleteLastCircular(CLogEntry* tail)`

***Purpose*:**

Remove the last node in the circular list (the tail).

Steps:

- Traverse to second-last node.
- Unlink and free the tail.
- Set new tail's next to head.

searchByIDCircular

```
CLogEntry* searchByIDCircular(CLogEntry* tail, int id)
```

***Purpose*:**

Find the first node with matching ID.

Steps:

- Traverse list.
- Compare each id.
- Return node if found; otherwise, return NULL.

searchByTimestampCircular

`CLogEntry* searchByTimestampCircular(CLogEntry* tail, const char* timestamp)`

***Purpose*:**

Find the first node with matching timestamp.

Steps:

- Traverse list.
- Use strcmp() to compare timestamps.
- Return match if found.

searchByKeywordCircular

```
CLogEntry* searchByKeywordCircular(CLogEntry* tail, const char* keyword)
```

***Purpose*:**

Find the first node where message contains a keyword.

Steps:

- Traverse list.
- Use strstr() to find the keyword.
- Return the node.

sortByDateCircular

`CLogEntry* sortByDateCircular(CLogEntry* tail)`

***Purpose*:**

Sort the list based on timestamps in ascending order.

Steps:

- Use nested loops to compare timestamp strings.
- Swap contents (not pointers) if needed.
- Maintain circular structure.

sortBySeverityCircular

`CLogEntry* sortBySeverityCircular(CLogEntry* tail)`

***Purpose*:**

Sort the list by severity (ascending).

Steps:

- Nested traversal and compare severity.
- Swap contents as needed.

reverseCircular

```
CLogEntry* reverseCircular(CLogEntry* tail)
```

***Purpose*:**

Reverse the order of the circular list.

Steps:

- Use three pointers: prev, curr, next.
- Reverse all next links in a loop.
- Update tail to the new last node.

countLogsCircular

`int countLogsCircular(CLogEntry* tail)`

***Purpose*:**

Count number of nodes in the circular list.

Steps:

- Start from tail->next.
- Traverse once through list, incrementing a counter.
- Return count.

traverseCircular

void traverseCircular(CLogEntry* tail)

***Purpose*:**

Print all nodes in the circular list.

Steps:

- Start from tail->next.
- Loop until you circle back.
- Print each node's content.

createFixedSizeBuffer

`CLogEntry* createFixedSizeBuffer(CLogEntry* tail, int id, const char* timestamp, int severity, const char* message, int* size)`

***Purpose*:**

Implements a circular log buffer with a maximum size. When full, it overwrites the oldest entry.

Steps:

- If not full:
  - Insert normally and increment size.
- If full:
  - Remove head.
  - Insert new log at end (tail).
- Maintains FIFO logic.

detectCycle

int detectCycle(CLogEntry* head)

***Purpose*:**

Verify that a cycle exists (for consistency check).

Steps:

- Use Floyd’s Tortoise and Hare algorithm.
- If slow and fast pointers meet → cycle exists.
- Otherwise → no cycle.

First Part

Queues

- isQueueEmpty
- isQueueFull
- enqueue

- dequeue
- peekQueue

isQueueEmpty

```
int isQueueEmpty(Queue* q)
```

***Purpose*:**

Checks whether the queue is empty.

Steps:

- Return 1 (true) if size == 0, otherwise 0.

isQueueFull

`int isQueueFull(Queue* q)`

***Purpose*:**

Checks whether the queue is full.

Steps:

- Return 1 if size == MAX_QUEUE_SIZE, otherwise 0.

enqueue

void enqueue(Queue* q, QLogEntry log)

Purpose:

Adds a new log entry to the end (rear) of the queue.

Steps:

- Check if the queue is full.
  - If full, print a message and return.
- Increment rear using circular logic:
  - rear = (rear + 1) % MAX_QUEUE_SIZE.
- Store the log at entries[rear].
- Increment the size.

dequeue

QLogEntry dequeue(Queue* q)

Purpose:

Removes and returns the log entry at the front of the queue.

Steps:

- Check if the queue is empty.
  - If empty, return an empty log and prin
    message.
- Retrieve the log at entries[front].
- Increment front using circular logic:
  - front = (front + 1) % MAX_QUEUE_SIZE.
- Decrement size.
- Return the retrieved log.

peekQueue

QLogEntry peekQueue(Queue* q)

***Purpose*:**

Returns the front log without removing it.

Steps:

- If queue is empty:
  - Print a message and return an empty log.
- Return entries[front].

First Part

Stacks

- Push New Log Entry
- Pop Log Entry
- Peek

- Check if Stack is Empty or Full.
- Reverse a Stack Using Recursion

isEmpty

```
int isEmpty(Stack* stack)
```

Purpose:

Check if the stack is empty.

Steps:

- Return 1 (true) if top == -1, else return 0.

isFull

```
int isFull(Stack* stack)
```

***Purpose*:**

Check if the stack is full.

Steps:

- Return 1 if top == MAX_STACK_SIZE - 1, else 0.

push

void push(Stack* stack, SLogEntry log)

***Purpose*:**

Push a new log entry onto the stack (top of the stack).

Steps:

- If the stack is full, print a warning and return.
- Increment top.
- Assign the log to entries[top].

pop

SLogEntry pop(Stack* stack)

***Purpose*:**

Remove and return the log at the top of the stack.

Steps:

- If stack is empty, print a warning and return an empty log.
- Return the current top log and decrement top.

peek

```
SLogEntry peek(Stack* stack)
```

Purpose:

Return the top log without removing it.

Steps:

- If stack is empty, return an empty log with a message.
- Return entries[top].

insertAtBottom

```
void insertAtBottom(Stack* stack, SLogEntry log)
```

***Purpose*:**

Helper function to insert a log at the bottom of the stack using recursion.

Steps:

- If stack is empty, push the log.
- Otherwise:
  - Pop the top.
  - Recursively call insertAtBottom().
  - Push the popped element back.

reverseStack

```
void reverseStack(Stack* stack)
```

***Purpose*:**

Reverse the entire stack using recursion.

Steps:

- If the stack is empty, return.
- Pop the top element.
- Recursively reverse the remaining stack.
- Use insertAtBottom() to insert the popped element at the bottom.

First Part

Recursion

- Reverse a Linked List
- Calculate Factorial and Fibonacci
- Find Maximum Log Entry ID

- Recursive Binary Search in Sorted Logs
- Convert an Infix Expression to Postfix

reverseListRecursive

RLogEntry* reverseListRecursive(RLogEntry* head)

Purpose:

Recursively reverses a singly linked list.

Steps:

- Base case: if head is NULL or only one node, return head.
- Recurse to the end of the list (head->next).
- Set head->next->next = head to reverse link.
- Set head->next = NULL to mark the new end.
- Return the new head of the reversed list.

fibonacci

int fibonacci(int n)

***Purpose*:**

Calculate the nth Fibonacci number using recursion.

Steps:

- If n == 0, return 0.
- If n == 1, return 1.
- Otherwise, return fibonacci(n - 1) + fibonacci(n - 2).

findMaxID

int findMaxID(RLogEntry* head)

Purpose:

Recursively find the maximum id value in a linked list of log entries.

Steps:

- Base case: if node has no next, return head->id.
- Recursively call findMaxID on the rest of the list.
- Compare current head->id with max from the rest.
- Return the greater value.

binarySearchLog

int binarySearchLog(RLogEntry* logs[], int left, int right, int targetID)

***Purpose*:**

Perform a recursive binary search to find a log by its ID in a sorted array.

Steps:

- If left > right, return -1 (not found).
- Calculate mid = (left + right) / 2.
- If logs[mid]->id == targetID, return mid.
- If logs[mid]->id < targetID, search the right half.
- Else, search the left half.

infixToPostfix

void reverseStack(Stack* stack)

Purpose:

Convert a simple infix expression to postfix using recursion (basic digit/operator support).

Steps:

- Base case: if end of string (*infix == '\0'), return.
- If current character is a digit:
  - Append to postfix and increment index.
- If it's an operator (+ - * /):
  - Recurse first on the rest of the expression.
  - Then append operator (post-order behavior).
- For other characters (e.g., parentheses), just recurse.

First Part

Trees

- insertBST
- deleteBST
- searchBST
- inOrderTraversal
- preOrderTraversal

- postOrderTraversal
- countListNodes
- sortedListToBST
- buildHeap

insertBST

```
TLogEntry* insertBST(TLogEntry* root, int id, const char* timestamp, int severity, const char* message)
```

***Purpose*:**

Insert a log into a Binary Search Tree (BST) based on timestamp order.

Steps:

- If tree is empty, create and return new node.
- If timestamp < root->timestamp, recurse left.
- Else, recurse right.
- Return the root after insertion.

deleteBST

`TLogEntry* deleteBST(TLogEntry* root, const char* timestamp)`

Purpose:

Delete a log entry from the BST using its timestamp.

Steps:

- If tree is empty, return NULL.
- Compare timestamp:
  - If smaller, recurse left.
  - If greater, recurse right.
  - If equal:
    - If node has no left child, return right.
    - If node has no right child, return left.
    - If both children exist:
      - Find in-order successor (findMin on right subtree).
      - Copy successor's data into current node.
      - Delete the successor node from right subtree.
  - Return updated root.

searchBST

```
TLogEntry* searchBST(TLogEntry* root, const char* timestamp)
```

***Purpose*:**

Search for a log entry in the BST by its timestamp.

Steps:

- If root is NULL or timestamps match, return root.
- If target timestamp is smaller, search left subtree.
- Otherwise, search right subtree.

inOrderTraversal

void inOrderTraversal(TLogEntry* root)

***Purpose*:**

Prints logs in ascending timestamp order (Left, Root, Right).

Steps:

- Recurse on left subtree.
- Print root timestamp.
- Recurse on right subtree.

preOrderTraversal

`void preOrderTraversal(TLogEntry* root)`

***Purpose*:**

Prints logs in preorder (Root, Left, Right).

Steps:

- Print root timestamp.
- Recurse on left subtree.
- Recurse on right subtree.

postOrderTraversal

```
void postOrderTraversal(TLogEntry* root)
```

***Purpose*:**

Prints logs in postorder (Left, Right, Root).

Steps:

- Recurse on left subtree.
- Recurse on right subtree.
- Print root timestamp.

countListNodes

int countListNodes(TLogEntry* head)

Purpose:

Count the number of nodes in a right-linked sorted list (used for BST conversion).

Steps:

- Initialize a counter.
- Traverse the list using right pointers.
- Increment and return the count.

sortedListToBST

`TLogEntry* sortedListToBST(TLogEntry** headRef, int n)`

Purpose:

Converts a sorted linked list to a height-balanced BST.

Steps:

- Recursively build left subtree using first n/2 nodes.
- Set the next node as root.
- Recursively build right subtree with remaining nodes.
- Return the root node.

buildHeap

void buildHeap(TLogEntry arr[], int n)

Purpose:

Converts an array of log entries into a valid max-heap based on timestamps.

Steps:

- Start from the last non-leaf node: n/2 - 1.
- Call heapify on each node up to the root.

Second Part

Modules based on Linked lists and Queues

- Get Synonym Words List
- Get Antonym Words List
- Get Word Information (by Word)
- Get Word Information (by Synonym/Antonym)
- Sort Words Alphabetically
- Sort Words by Character Count
- Sort Words by Vowel Count
- Delete Word from File and Lists
- Update Word Synonym and Antonym
- Find Similar Words by Match Rate
- Find Words Containing a Substring
- Get Sorted Palindrome Words
- Merge Synonym and Antonym Lists (Doubly)
- Merge Synonym and Antonym Lists (Circular)
- Add New Word to Lists and File
- Sort Words by Syllables into Queue
- Sort Words by Pronunciation Type
- Convert Merged List to Queue

getInfWord2

```
void getInfWord2(TList *syn, TList *ant, char *inf)
```

Steps:

- Search synonym list:
- Traverse the synonym linked list nodes, comparing the relatedWord field with inf.
- If a match is found, save the corresponding word, number of characters, and vowels.

Output:

- If found, print the word with its length and vowel count. Otherwise, print a message saying it was not found.

Purpose:

Given a synonym or antonym string (inf), find the original word that corresponds to it along with that word's length and vowel count.

sortWord

TList *sortWord(TList *syn)

Steps:

- Use bubble sort on the linked list:
- Iterate through the list multiple times until no swaps are needed.
- Compare the word strings of adjacent nodes lexicographically.
- If out of order, swap the contents of the two nodes.
- Return the head pointer of the sorted list.

Purpose:

Sort the linked list nodes alphabetically by the word field.

sortWord2

TList *sortWord2(TList *syn)

Steps:

- Similar to sortWord, perform a bubble sort traversal.
- Compare numChars of adjacent nodes.
- Swap nodes' contents if they are out of order (greater first).
- Repeat until the list is sorted by ascending length.

Purpose:

Sort the linked list nodes in ascending order based on the number of characters in each word.

sortWord3

TList *sortWord3(TList *syn)

Steps:

- Use bubble sort again on the linked list.
- Compare numVowels of adjacent nodes.
- Swap nodes’ contents if the previous has fewer vowels than the next (to achieve descending order).
- Continue passes until fully sorted.

Purpose:

Sort the linked list nodes in descending order based on the number of vowels in each word.

deleteWord

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

Steps:

- Delete from linked lists:
- Traverse each list and remove nodes where node->word matches word.
- Carefully update pointers to maintain the list integrity.
- Free memory of removed nodes.
- Delete from file:
- Open the original file for reading.
- Open a temporary file for writing
- Copy all lines except those starting with word into the temp file.
- Close both files.
- Delete the original file and rename the temp file to original filename
- Return the updated synonym list pointer (or both lists as needed).

Purpose:

Delete a word from the synonym and antonym linked lists and remove it from the original file.

updateWord

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

Steps:

1.Update linked lists:

- Find the node in synonym list matching word, replace its relatedWord with new synonym syne.
- Do the same in antonym list with new antonym anton.

2.Update file:

- Read the file line by line.
- When the line contains word, rewrite that line with updated synonym and antonym.
- Write all other lines unchanged into a temporary file.
- Return the updated synonym list pointer (or both lists as needed).
- Replace original file with the temp file.

Purpose:

Update the synonym and antonym of a given word both in linked lists and in the original file.

similarWord

Steps:

1.Define similarity metric:

- TList *similarWord(TList *syn, char *word, double rate)
- Calculate similarity as the ratio of sequential matching characters (case-insensitive) between word and each candidate word.

2.Search for similar words:

- Traverse synonym list.
- For each word, compute similarity score with input word.
- If similarity ≥ rate, add that word (and its related word) to a new linked list.
- Return the head of this new list containing all similar words.

Purpose:

Return a linked list of words similar to word with a similarity rate ≥ rate.

countWord

`TList *countWord(TList *syn, char *prt)`

Purpose:

Return a new linked list containing all words from the syn list where the substring prt appears anywhere in the word (case-insensitive).

Steps:

- Iterate through each node in the syn list.
- Calculate similarity as the ratio of sequential matching characters (case-insensitive) between word and each candidate word.
- After processing all nodes, return the new list containing all matching words.

palindromWord

TList *palindromWord (TList *syn)

Purpose:

Return a sorted linked list of all palindrome words from the syn list, including their information.

Steps:

- Iterate over each node in the syn list.
- For each word, check if it is a palindrome (reads the same backward and forward, case-insensitive).
- If it is, create a new node with that word and its info.
- Insert this node into the result list at the correct position to keep the list alphabetically sorted by word.
- Return the sorted palindrome list.

merge

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

Purpose:

Merge two singly linked lists (syn and ant) into one bidirectional (doubly linked) list.

Steps:

- Create a new empty doubly linked list.
- Alternately take nodes from syn and ant lists (or as available), create new nodes, and append them to the new list.
- For each new node, set both next and prev pointers correctly to form a doubly linked list.
- Return the head of the merged doubly linked list.

merge2

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

Purpose:

Merge two singly linked lists (syn and ant) into one circular doubly linked list.

Steps:

- Similar to merge, create a new doubly linked list from syn and ant.
- After all nodes are added, link the last node’s next pointer back to the head node, and the head’s prev pointer back to the last node, making the list circular.
- Return the head of the circular doubly linked list.

addWord

TList *addWord(TList *syn, TList *ant, const char *word, const char *syne, const char *anton, const char *filename)

Purpose:

Add a new word along with its synonym and antonym to the syn and ant linked lists, and append the new entry to the text file.

Steps:

- Create a new node for the synonym list with word and syne, and append it to the end of the syn list.
- Create a new node for the antonym list with word and anton, and append it to the end of the ant list.
- Open the text file in append mode.
- Write a new line with the word, synonym, and antonym separated by spaces.
- Close the file.
- Return the updated synonym list pointer (you can handle updating the antonym list separately if needed).

syllable

```
TQueue *syllable(TList *syn)
```

Steps:

- Count the syllables for each word using a helper function (like counting groups of vowels).

- Find the maximum number of syllables among all words. This defines how many syllable groups there will be.

- For each syllable count from 1 to the maximum:

1. Loop through all words.
2. Enqueue those with the current syllable count into the queue.
3. After all words for this count are enqueued, enqueue a NULL pointer as a separator.

- Return the queue, which now contains words grouped by syllable count separated by NULLs.

Purpose:

This function groups words by their number of syllables into a queue. Between groups of words with different syllable counts, it inserts empty (NULL) separators.

prounounciation

TQueue *prounounciation(TList *syn)

Steps:

- Create three empty queues for short, long, and diphthong pronunciation groups.
- Find the maximum number of syllables among all words. This defines how many syllable groups there will be.
- For each word in the input list:
  - Check if it contains a diphthong substring (like "ai", "ei", etc.). If yes, enqueue it to the diphthong queue.
  - Else, check if it contains a long vowel (double letters like "aa", "ee"). If yes, enqueue to the long vowel queue.
  - Otherwise, enqueue it to the short vowel queue.
- Return one or more of these queues as needed (you can create a structure to return all three queues if desired).

Purpose:

This function sorts words into three queues based on how their vowels are pronounced:

- Short vowels
- Long vowels (double vowels)
- Diphthongs (vowel pairs pronounced together)

toQueue

TQueue *toQueue(TList *merged)

Purpose:

Convert a doubly linked list (merged) into a queue structure.

Steps:

- CREATE AN EMPTY QUEUE.
- TRAVERSE THE DOUBLY LINKED LIST FROM ITS HEAD TO THE END.
- FOR EACH NODE IN THE LIST, ENQUEUE A POINTER TO THAT NODE INTO THE QUEUE.
- AFTER THE TRAVERSAL IS COMPLETE, RETURN THE QUEUE WHICH NOW CONTAINS ALL NODES IN THE ORDER THEY APPEARED IN THE LIST.

Second Part

Modules based on Stacks

- toStack
- getInfWordStack
- sortWordStack
- deleteWordStack
- updateWordStack
- stackToQueue
- StacktoList
- addWordStack

- syllableStack
- prounounciationStack
- getSmallest
- cycleSearch
- isPalyndromeStack
- Reverse Stack

toStack

TStack *toStack(TList *merged)

Purpose:

CONVERTS A SINGLY LINKED LIST INTO A STACK (REVERSING THE ORDER)

Steps:

- **INITIALIZE EMPTY STACK**
- **ITERATE THROUGH INPUT LIST**
- **FOR EACH NODE:**
  - **ALLOCATE NEW STACK NODE**
  - **DUPLICATE STRING DATA (WORD, SYNONYM, ANTONYM)**
  - **PUSH ONTO STACK (LIFO ORDER)**
- **RETURN RESULTING STACK**

getInfWordStack

TWordInfo *getInfWordStack(TStack *stk, const char *word)

Purpose:

RETRIEVES DETAILED INFORMATION ABOUT A SPECIFIC WORD IN THE STACK

Steps:

- **SEARCH STACK FOR MATCHING WORD .**
- **IF FOUND:**
  - **ALLOCATE TWORDINFO STRUCT .**
  - **DUPLICATE WORD DATA .**
  - **CALCULATE CHARACTER AND VOWEL COUNTS .**
  - **RETURN STRUCT POINTER .**
- **ELSE:**
  - **RETURN NULL .**

sortWordStack

TStack *sortWordStack(TStack *syn)

Purpose:

SORTS STACK ALPHABETICALLY BY WORD FIELD

Steps:

- **COUNT NODES IN STACK**
- **CREATE ARRAY OF NODE POINTERS**
- **SORT ARRAY USING QSORT()**
- **REBUILD STACK FROM SORTED ARRAY**
- **RETURN NEW HEAD**

deleteWordStack

`TStack *deleteWordStack(TStack *stk, const char *word)`

Steps:

- SEARCH FOR WORD IN STACK
- IF FOUND:
  - ADJUST ADJACENT NODE POINTERS
  - FREE NODE MEMORY (STRINGS AND STRUCT)
- RETURN MODIFIED STACK

Purpose:

REMOVES A NODE WITH MATCHING WORD FROM STACK

updateWordStack

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

Purpose:

UPDATES SYNONYM AND ANTONYM FOR A WORD IN STACK

Steps:

- SEARCH FOR WORD IN STACK
- IF FOUND:
  - FREE OLD STRINGS
  - DUPLICATE NEW STRINGS (IF PROVIDED)
- RETURN STACK

stackToQueue

TQueue *stackToQueue(TStack *stk)

Purpose:

CONVERTS STACK TO QUEUE (FIFO ORDER)

Steps:

- **SORT STACK ALPHABETICALLY**
- **CREATE NEW QUEUE NODES FOR EACH ELEMENT**
- **MAINTAIN FRONT AND REAR POINTERS**
- **RETURN QUEUE HEAD**

StacktoList

TList *StacktoList(TStack *stk)

Steps:

- **SORT STACK ALPHABETICALLY**
- **CREATE NEW LIST NODES**
- **MAINTAIN PREV/NEXT POINTERS**
- **RETURN LIST HEAD**

Purpose:

CONVERTS STACK TO SORTED DOUBLY LINKED LIST

addWordStack

```
TStack *addWordStack(TStack *stk, char *word, char *syne, char *anton)
```

Steps:

- **CREATE NEW NODE**
- **FIND CORRECT POSITION (ALPHABETICAL ORDER)**
- **INSERT NODE**
- **RETURN (POTENTIALLY NEW) STACK TOP**

Purpose:

INSERTS NEW WORD INTO SORTED STACK

syllableStack

TStack *syllableStack(TStack *stk)

Steps:

- **COUNT SYLLABLES IN EACH WORD**
- **CREATE ARRAY OF NODE COPIES**
- **BUBBLE SORT BY SYLLABLE COUNT**
- **REBUILD STACK**
- **RETURN NEW HEAD**

Purpose:

SORTS STACK BY SYLLABLE COUNT

prounounciationStack

TPronunciationStacks prounounciationStack(TStack *stk)

Purpose:

CATEGORIZES WORDS BY PRONUNCIATION TYPE

Steps:

- **CHECK EACH WORD FOR:**
  - **DIPHTHONGS (VOWEL COMBINATIONS)**
  - **LONG VOWELS**
  - **SHORT VOWELS**
- **ADD TO APPROPRIATE STACK**
- **RETURN STRUCT WITH THREE STACKS**

getSmallest

```
char *getSmallest(TStack *stk)
```

Purpose:

FINDS LEXICOGRAPHICALLY SMALLEST WORD

Steps:

- **INITIALIZE WITH FIRST WORD**
- **COMPARE WITH ALL OTHER WORDS**
- **RETURN SMALLEST FOUND**

cycleSearch

void cycleSearch(TStack *stk)

Steps:

- **FOR EACH WORD:**
- **FOLLOW SYNONYM/ANTONYM CHAIN**
- **TRACK VISITED WORDS**
- **DETECT CYCLES**
- **PRINT RESULTS**

Purpose:

DETECTS CIRCULAR REFERENCES IN SYNONYMS/ANTONYMS

isPalindromeStack

bool isPalindromeStack(char *word)

Purpose:

CHECKS IF WORD IS PALINDROME

Steps:

- **PUSH ALL CHARACTERS ONTO STACK**
- **COMPARE WITH ORIGINAL STRING**
- **MERGE THE TWO SORTED ARRAYS.**
- **RETURNS TRUE IF PALINDROME, FALSE OTHERWISE .**

StackRev

TStack* StackRev(TStack *stk)

Purpose:

REVERSES STACK USING RECURSION

Steps:

- **BASE CASE: EMPTY/SINGLE-NODE STACK**
- **REMOVE TOP NODE**
- **RECURSIVELY REVERSE REMAINDER**
- **INSERT OLD TOP AT BOTTOM**
- **RETURN NEW HEAD**

Second Part

Modules based on Binary Search Tree

- toTree
- fillTree
- getInfWordTree
- AddWordBST
- deleteWordBST
- UpdateWordBST
- TraversalBSTinOrder
- TraversalBSTpreOrder
- TraversalBSTpostOrder

- HighSizeBST
- LowestCommonAncestor
- CountNodesRanges
- inOrderSuccesor
- BSTMirror
- isBalencedBST
- BTSMerge

toTree

TTree *toTree(TStack *stk)

Steps:

- **INITIALIZE BST ROOT AS NULL.**
- **COMPARE THE TARGET WORD WITH THE CURRENT NODE'S WORD:**

1. **POP THE TOP ELEMENT FROM THE STACK.**
2. **INSERT THE POPPED ELEMENT'S DATA INTO THE BST USING YOUR BST INSERTION FUNCTION.**
3. **FREE THE POPPED STACK NODE.**

- **RETURN THE ROOT OF THE CONSTRUCTED BST.**

Purpose:

Converts a stack of word data nodes into a Binary Search Tree (BST)

fillTree

TTree *fillTree(FILE *f)

Steps:

- **INITIALIZE BST ROOT AS NULL.**
- **WHILE THE FILE HAS LINES TO READ:**

1. **Read a word, its synonym, and antonym.**
2. **Create a WordData struct and fill it (compute vowels and length).**
3. **Insert this data into the BST.**

- **RETURN THE ROOT OF THE FILLED BST.**

Purpose:

Reads words, synonyms, and antonyms from a file and inserts them into a BST.

getInfWordTree

```
TTree *getInfWordTree(TTree *tr, char *word)
```

Steps:

- **IF THE TREE IS EMPTY, RETURN NULL.**
- **COMPARE THE TARGET WORD WITH THE CURRENT NODE'S WORD:**

1. **- IF EQUAL, RETURN THE CURRENT NODE.**
2. **- IF TARGET WORD IS LESS, RECURSE INTO THE LEFT SUBTREE.**
3. **- ELSE RECURSE INTO THE RIGHT SUBTREE.**

Purpose:

Searches the BST for a node matching the given word and returns its detailed data.

AddWordBST

TStack *AddWordBST(TTree *tr, char *word, char *syne, char *anton)

Steps:

- **IF THE BST TR IS EMPTY, CREATE A NEW NODE WITH GIVEN DATA AND RETURN IT.**
- **COMPARE THE NEW WORD WITH THE CURRENT NODE'S WORD:**

1. **IF SMALLER, RECURSIVELY INSERT INTO LEFT SUBTREE.**
2. **- IF GREATER, RECURSIVELY INSERT INTO RIGHT SUBTREE.**

- **IF THE BST TR IS EMPTY, CREATE A NEW NODE WITH GIVEN DATA AND RETURN IT.**

Purpose:

Adds a word with its synonym and antonym into a sorted BST.

deleteWordBST

```
TTree *deleteWordBST(TTree *tr, char *word)
```

Steps:

- **SEARCH RECURSIVELY FOR THE NODE CONTAINING THE WORD.**
- **WHEN FOUND, HANDLE THREE CASES:**

1. **NO CHILDREN: FREE NODE AND RETURN NULL.**
2. **ONE CHILD: FREE NODE AND RETURN THE CHILD.**
3. **TWO CHILDREN: FIND THE IN-ORDER SUCCESSOR (MINIMUM NODE IN RIGHT SUBTREE), REPLACE CURRENT NODE'S DATA WITH SUCCESSOR'S, THEN DELETE SUCCESSOR RECURSIVELY.**

- **RETURN THE UPDATED TREE ROOT.**

Purpose:

Deletes a word from the BST.

UpdateWordBST

```
TTree *UpdateWordBST(TTree *tr, char *word, char *syne, char *anton)
```

Steps:

- SEARCH FOR THE NODE WITH THE GIVEN WORD RECURSIVELY.
- IF FOUND, COPY NEW SYNONYM AND ANTONYM INTO NODE.
- RETURN THE UPDATED TREE ROOT.

Purpose:

Updates synonym and antonym for a word in the BST.

TraversalBSTinOrder

`TTree *TraversalBSTinOrder(TTree *tr)`

Steps:

- **RECURSIVELY TRAVERSE THE LEFT SUBTREE.**
- **PROCESS THE CURRENT NODE (E.G., PRINT ITS WORD AND INFO).**
- **RECURSIVELY TRAVERSE THE RIGHT SUBTREE.**

Purpose:

Performs an in-order traversal of the BST (left, node, right) which visits nodes in lex order.

TraversalBSTpreOrder

`TTree *TraversalBSTpreOrder(TTree *tr)`

Steps:

- **PROCESS CURRENT NODE**
- **RECURSIVELY TRAVERSE LEFT SUBTREE.**
- **RECURSIVELY TRAVERSE RIGHT SUBTREE.**

Purpose:

Performs a pre-order traversal (node, left, right).

TraversalBSTpostOrder

```
TTree *TraversalBSTpostOrder(TTree *tr)
```

Purpose:

Performs a post-order traversal (left, right, node).

Steps:

- RECURSIVELY TRAVERSE LEFT SUBTREE.
- RECURSIVELY TRAVERSE RIGHT SUBTREE.
- PROCESS CURRENT NODE.

void HighSizeBST

```
void HighSizeBST(TTree *tr)
```

Purpose:

Prints the height and size (total nodes) of the BST.

Steps:

- **CALCULATE THE HEIGHT RECURSIVELY (MAX DEPTH).**
- **CALCULATE THE SIZE RECURSIVELY (NODE COUNT).**
- **PRINT THE RESULTS**

LowestCommonAncestor

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

Steps:

- **IF BOTH WORDS ARE LEX LESS THAN CURRENT NODE, RECURSE LEFT.**
- **IF BOTH ARE LEX GREATER, RECURSE RIGHT.**
- **OTHERWISE, CURRENT NODE IS THE LCA.**

Purpose:

Finds the lowest common ancestor (LCA) of two nodes in the BST.

CountNodesRanges

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

Steps:

- IF NODE IS NULL, RETURN 0.
- CHECK IF CURRENT NODE'S CHAR COUNT LIES WITHIN RANGE; COUNT 1 IF YES.
- RECURSIVELY COUNT IN LEFT AND RIGHT SUBTREES.
- RETURN SUM.

Purpose:

Counts nodes whose word length (or another int property) lies within [l, h].

inOrderSuccessor

`TTree* inOrderSuccessor(TTree *tr , char *word)`

Purpose:

Finds the in-order successor node of a given word in BST
(next node in lex order).

Steps:

- **START AT ROOT, TRACK POTENTIAL SUCCESSOR.**
- **WHILE TRAVERSING, IF WORD IS LESS THAN CURRENT NODE'S WORD, UPDATE SUCCESSOR AND GO LEFT.**
- **IF WORD IS GREATER, GO RIGHT.**
- **WHEN NODE IS FOUND, IF IT HAS RIGHT CHILD, SUCCESSOR IS MINIMUM IN RIGHT SUBTREE.**
- **RETURN SUCCESSOR.**

BSTMirror

```
TTree* BSTMirror(TTree *tr)
```

Purpose:

RETURNS MIRROR IMAGE OF BST BY SWAPPING LEFT AND RIGHT CHILDREN RECURSIVELY.

Steps:

- **IF NODE IS NULL, RETURN NULL.**
- **ALLOCATE NEW NODE AND COPY DATA.**
- **RECURSIVELY SET LEFT SUBTREE OF NEW NODE AS MIRROR OF ORIGINAL RIGHT SUBTREE.**
- **RECURSIVELY SET RIGHT SUBTREE OF NEW NODE AS MIRROR OF ORIGINAL LEFT SUBTREE.**
- **RETURN NEW MIRRORED NODE.**

isBalancedBST

```
bool isBalancedBST(TTree *tr)
```

Steps:

- **RECURSIVELY CALCULATE HEIGHT OF LEFT AND RIGHT SUBTREES.**
- **IF DIFFERENCE EXCEEDS 1 AT ANY NODE, MARK TREE AS UNBALANCED.**
- **RETURN BALANCE STATUS.**

Purpose:

CHECKS WHETHER BST IS HEIGHT-BALANCED (DIFFERENCE IN HEIGHTS OF LEFT/RIGHT SUBTREE ≤ 1 EVERYWHERE).

BTSMerge

TTree *BTSMerge(TTree *tr1, TTree *tr2)

Purpose:

MERGES TWO BSTS INTO A SINGLE BALANCED BST.

Steps:

- **CALCULATE SIZES OF BOTH BSTS.**
- **CONVERT EACH BST TO A SORTED ARRAY VIA IN-ORDER TRAVERSAL.**
- **MERGE THE TWO SORTED ARRAYS.**
- **CONSTRUCT BALANCED BST FROM THE MERGED SORTED ARRAY RECURSIVELY.**
- **FREE TEMPORARY ARRAYS.**
- **RETURN BALANCED BST ROOT.**

Second Part

Modules based on Recursion

- countWordOccurence
- removeWordOccurence
- replaceWordOccurence
- swap
- permute

- wordPermutation
- subseqWord
- longestSubseqWord
- isPalindromWord

countWordOccurence

countWordOccurence(File *f, char *word)

Purpose:

COUNTS HOW MANY TIMES A SPECIFIC WORD APPEARS IN A LINKED LIST OF FILE NODES.

Steps:

- **IT RECURSIVELY TRAVERSES THE LINKED LIST.**
- **FOR EACH NODE, IT COMPARES THE NODE'S WORD TO THE TARGET WORD.**
- **IF THEY MATCH, IT ADDS 1 TO THE COUNT.**
- **IT RETURNS THE SUM OF MATCHES IN THE CURRENT NODE AND THE REST OF THE LIST.**

removeWordOccurence

removeWordOccurence(File *f, char *word)

Purpose:

REMOVES ALL NODES FROM THE LINKED LIST WHERE THE NODE'S WORD MATCHES THE GIVEN WORD.`1

Steps:

- **RECURSIVELY TRAVERSES THE LIST.**
- **FOR EACH NODE, PROCESSES THE NEXT NODES FIRST (F->NEXT = REMOVEWORDOCCURENCE(F->NEXT, WORD)).**
- **IF THE CURRENT NODE'S WORD MATCHES, FREES THE MEMORY FOR THE WORD AND THE NODE, THEN RETURNS THE NEXT NODE, EFFECTIVELY REMOVING THE CURRENT ONE.**
- **OTHERWISE, RETURNS THE CURRENT NODE UNCHANGED.**

replaceWordOccurence

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

Purpose:

REPLACES EVERY OCCURRENCE OF A TARGET WORD IN THE LINKED LIST WITH ANOTHER WORD (REP).

<u>Steps:</u>

- **RECURSIVELY TRAVERSES THE LINKED LIST.**
- **WHEN A NODE'S WORD MATCHES THE TARGET, IT FREES THE OLD WORD AND DUPLICATES THE REPLACEMENT WORD INTO THE NODE.**
- **CONTINUES RECURSIVELY UNTIL THE END OF THE LIST.**

swap

swap(char *x, char *y)

Purpose:

SWAPS TWO CHARACTERS IN MEMORY.

<u>Steps:</u>

- **TEMPORARILY STORES THE VALUE OF *X.**
- **ASSIGNS *Y TO *X.**
- **ASSIGNS THE TEMPORARY STORED VALUE TO *Y.**
- **THIS IS A UTILITY FUNCTION USED IN STRING PERMUTATIONS.**

permute

permute(char *word, int l, int r)

Purpose:

GENERATES AND PRINTS ALL PERMUTATIONS OF THE STRING WORD FROM INDEX L TO R.

Steps:

- **IF THE LEFT INDEX L EQUALS RIGHT INDEX R, THE CURRENT PERMUTATION IS PRINTED.**
- **OTHERWISE, FOR EACH POSITION FROM L TO R, IT:**
  - **SWAPS THE CURRENT CHARACTER AT L WITH TH**
  - **RECURSIVELY PERMUTES THE REST (L+1 TO R).**
  - **SWAPS BACK TO RESTORE THE ORIGINAL STRING (BACKTRACKING).**

wordPermutation

wordPermutation(char *word)

Steps:

- **CALLS PERMUTE WITH L = 0 AND R = STRLEN(WORD) - 1 TO GENERATE ALL PERMUTATIONS OF THE FULL STRING.**

Purpose:

WRAPPER FUNCTION THAT STARTS PERMUTATION GENERATION FOR THE ENTIRE STRING WORD.

subseqWord

subseqWord(char *word)

Purpose:

GENERATES AND PRINTS ALL NON-EMPTY SUBSEQUENCES OF WORD.

Steps:

- **ALLOCATES MEMORY FOR A BUFFER SUBSEQ.**
- **CALLS SUBSEQHELPER TO GENERATE SUBSEQUENCES STARTING AT INDICES 0.**
- **FREES ALLOCATED MEMORY AFTER COMPLETION.**

longestSubseqWord

longestSubseqWord(char *word1, char *word2)

Steps:

- **USES RECURSION TO COMPARE CHARACTERS OF BOTH WORDS:**
  - **IF EITHER STRING ENDS, RETURNS 0.**
  - **IF THE CURRENT CHARACTERS MATCH, ADDS 1 AND RECURSES WITH NEXT CHARACTERS.**
  - **OTHERWISE, TAKES THE MAX BETWEEN SKIPPING A CHARACTER IN WORD1 OR WORD2.**

Purpose:

CALCULATES THE LENGTH OF THE LONGEST COMMON SUBSEQUENCE (LCS) BETWEEN TWO STRINGS WORD1 AND WORD2.

isPalindromWord

isPalindromWord(char *word)

Purpose:

CHECKS IF THE ENTIRE STRING WORD IS A PALINDROME

Steps:

- **CALLS ISPALINDROMEHELPER WITH START = 0 AND END = LENGTH-1.**

THE END.

- This project took a lot of time and effort from us. So we hope that we get the mark that we dereserve
- We leaned a lot this year and we hope to see you next year too .