**Project report :**

Made by :

-MEZHOUD Hiba B4.

-LAIFAOUI Marya Tesnim B4.

*first part :*

## 2.1. Module 1: Linked Linear Lists (linkedLists.c/linkedLists.h) :

**1. create\_node**

Create a new singly linked list node with specified data.

**Inputs:**

* int id
* int severity
* char timestemp[]
* char describtion[]

**Output:**

* node\* – A pointer to the newly created node.

**Steps:**

* Allocate memory for the new node.
* Assign values to its fields.
* Set next to NULL.

**Strengths:**

* Simple structure.
* Prepares node cleanly for insertion.

**Limitations:**

* No error handling for failed malloc.

**2. insert\_at\_beg**

Insert a new node at the beginning of the list.

**Inputs:**

* node\* head
* node\* newnode

**Output:**

* Updated node\* head

**Steps:**

* Set newnode->next = head.
* Update head = newnode.

**Strengths:**

* No list traversal needed.

**Limitations:**

* Requires pre-allocated node.

**3. insert\_at\_position**

Insert a node at a specific position.

**Inputs:**

* node\* head
* node\* newnode
* int position

**Output:** Updated head.

**Steps:**

* Traverse to (position - 1).
* Link newnode accordingly.

**Strengths:**

* Flexibility to insert anywhere.

**Limitations:**

* No bounds checking.

**4. delet\_by\_id**

Delete node by id.

**Inputs:**

* node\* head
* int id

**Output:** Updated head.

**Steps:**

* Handle empty list and head match.
* Traverse to node before match.
* Unlink and free node.

**Strengths:**

* Works for all cases: head, middle, tail.

**Limitations:**

* Linear time complexity.

**5. delet\_by\_timestemp**

Delete node by timestamp.

**Inputs:**

* node\* head
* char timestemp[]

**Output:** Updated head.

**Steps:**

* Same logic as delet\_by\_id, but compares strings.

**Strengths:**

* Allows removal by time-based identifier.

**Limitations:**

* Relies on exact match.
* Case-sensitive comparison.

**6. search\_by\_id, search\_by\_timestemp, search\_by\_keyword**

Locate a node based on a field.

**Inputs:**

* node\* head
* int id / char timestemp[] / char keyword[]

**Output:** Pointer to matching node or NULL.

**Steps:**

* Traverse and compare each node.

**Strengths:**

* Easy to use.
* Helps with filtering.

**Limitations:**

* Returns only first match.
* No multiple match handling.

**7. sort\_by\_severity**

Sort nodes in ascending order by severity.

**Inputs:** node\* head

**Output:** Sorted head.

**Steps:**

* Bubble sort logic using data swap (not pointer change).

**Strengths:**

* Simple, effective for small lists.

**8. reverse**

Reverse the list in-place.

**Inputs:** node\* head

**Output:** New head after reversal.

**Steps:**

* Iteratively reverse pointers using prev, current, next.

**Strengths:**

* In-place operation.

**Limitations:**

* Original head becomes tail (expected, but must be noted).

**9. count**

Count number of nodes.

**Input:** node\* head  
**Output:** Integer count.

**Strengths:**

* Simple and effective.

**Limitations:**

* Requires full traversal.

## ***2.2. Module 2: Bidirectional Linked Lists :***

**1. insertend**

Insert a node at the end.

**Inputs:** bnode\* head, int id, int severity, char\* time, char\* desc

**Output:** Updated head.

**Steps:**

* Traverse to last node.
* Link new node to the end.

**Strengths:**

* Standard tail insertion.

**2. insertbegin**

Insert node at the beginning.

**Steps:**

* Update prev and next pointers.

**Strengths:**

* Quick head insertion.

**3. deletebyiD**

Delete a node by id.

**Input:** bnode\* head, int id  
**Output:** Updated head.

**Steps:**

* Locate node, relink previous and next nodes.
* Free target node.

**Strengths:**

* Works for all positions.

**4. forward, backward**

Print list forward or backward.

**Input:** bnode\* head / bnode\* tail

**Strengths:**

* Useful for verification.

**Limitations:**

* Assumes tail known for reverse.

**5. mergelists**

Merge two doubly linked lists.

**Inputs:** bnode\* head1, bnode\* head2  
**Output:** Merged list head.

**Strengths:**

* Concatenates two lists in-place.

**Limitations:**

* No memory duplication.
* No sorting or validation on merge.

## ***2.3. Module 3: Circular Linked Lists (circularLists.c/circularLists.h) :***

**1. insert\_at\_beg\_CLL**

**Purpose:** Insert node at the beginning of a circular list.

**Input:** node\* head, node\* newnode  
**Output:** New head.

**Steps:**

* Link new node to head and last node.
* Update head.

**Strengths:**

* Maintains circular structure.

**Limitations:**

* Must traverse to last node

**2. insert\_at\_position\_CLL**

Insert node at a specific position.

**Strengths:**

* Flexible position-based insertion.

**3. insert\_at\_end\_CLL**

Insert node at the end.

**Strengths:**

* Handles both empty and non-empty lists.

**4. delete\_first\_CLL**

Delete first node in CLL.

**Strengths:**

* Handles single-node edge case.

**Limitations:**

* Requires full traversal to last node.

**5. delete\_last\_CLL**

Delete last node in CLL.

**Strengths:**

* Clean memory management.

**6. delete\_by\_id\_CLL / delete\_by\_timestemp\_CLL**

Delete a node by id or timestamp.

**Strengths:**

* Supports removal from any position.

**Limitations:**

* No indication when element not found (except a message).

**7. search\_by\_id\_CLL / search\_by\_timestemp\_CLL / search\_by\_keyword\_CLL**

Find nodes based on specific fields.

**Strengths:**

* Full-cycle searching.

**8. sort\_by\_date\_CLL / sort\_by\_severity\_CLL**

Sort CLL based on timestamp or severity.

**Steps:**

* Bubble sort by comparing values.
* Swap node content.

**Strengths:**

* Functional sorting in CLL.

**Limitations:**

* Inefficient (O(n²)).

**9. reverse\_CLL**

Reverse the CLL.

**Strengths:**

* Uses iterative pointer reversal.

**Limitations:**

* Complexity due to circular nature.

**10. count\_CLL**

Count number of nodes in CLL.

**Output:** Integer count.

## ***2.4. Module 4: Queues (queues.c/queues.h) :***

**1. createqueue**

Create an empty queue.

**Output:** Pointer to a new queue struct.

**Steps:**

* Allocate memory.
* Set front and rear to NULL.

**Strengths:**

* Initializes a proper empty queue structure.

**Limitations:**

* No error checking on malloc.

**2. enqueue**

Add a node to the rear of the queue.

**Inputs:**

* queue\* q
* node\* newnode

**Steps:**

* If queue is empty, both front and rear point to newnode.
* Else, link current rear’s next to newnode and update rear.

**Strengths:**

* Maintains FIFO structure.
* O(1) time complexity.

**Limitations:**

* Assumes newnode is valid.

**3. dequeue**

Remove a node from the front of the queue.

**Inputs:**

* queue\* q

**Output:** Updated q->front

**Steps:**

* If empty, return NULL.
* Else, remove front and free it.

**Strengths:**

* O(1) removal.
* Shows queue underflow case.

**Limitations:**

* Returns pointer to new front, not the dequeued node (could be enhanced).

**4. peek\_queue**

View the front element.

**Input:** queue\* q  
**Output:** Pointer to front node.

**Strengths:**

* Quick lookup.

**Limitations:**

* No indication of invalid access if queue is empty.

**5. is\_empty\_queue**

Check if queue is empty.

**Input:** queue\* q  
**Output:** Boolean result.

**Strengths:**

* Utility function for safety checks.

***2.5. Module 5: Stacks (stacks.c/stacks.h) :***

**1. is\_empty\_stack**

Check if stack is empty.

**Input:** stack s  
**Output:** Boolean result.

**Strengths:**

* Simple check for stack underflow.

**2. push**

Push a new node onto the stack.

**Inputs:**

* stack\* s
* node\* newnode

**Steps:**

* Set newnode’s next to current top.
* Update top.

**Strengths:**

* Fast O(1) insertion.

**Limitations:**

* No check on stack overflow or node validity.

**3. pop**

Pop the top node off the stack.

**Input:** stack\* s

**Steps:**

* Check for empty stack.
* Print node info.
* Update top and free the popped node.

**Strengths:**

* Shows informative pop result.

**Limitations:**

* Exits program on empty stack (could be handled more gracefully).

**4. peek\_stack**

View the top element.

**Input:** stack\* s  
**Output:** Pointer to top node.

**Strengths:**

* Quick access to current top.

***2.6. Module 6: Recursion (recursion.c/recursion.h) :***

**1. reverse\_ll**

Reverse a singly linked list recursively.

**Input:** node\* head  
**Output:** Reversed head pointer.

**Steps:**

* Base case: if head is NULL or has no next, return it.
* Recursive call on the rest of the list.
* Rearrange pointers.

**Strengths:**

* Elegant recursion.
* In-place reversal.

**Limitations:**

* Risk of stack overflow for long lists.
* Slightly harder to debug.

**2. rec\_factorial**

Calculate factorial of an integer n.

**Input:** int n  
**Output:** Integer factorial.

**Strengths:**

* Simple recursive pattern.

**Limitations:**

* No base case for n = 0.
* No error handling for invalid input.

**3. rec\_fibonacci**

Calculate the nth Fibonacci number recursively.

**Input:** int n  
**Output:** nth Fibonacci value.

**Strengths:**

* Conceptually clear.

**Limitations:**

* Inefficient (exponential time).
* Base cases are non-standard (returns 1 for n = 0 and n = 1).
* No memoization.

**4. max\_id**

**Purpose:** Find maximum id in a linked list recursively.

**Input:** node\* head  
**Output:** Maximum id value.

**Strengths:**

* Recursively traverses all nodes.

**Limitations:**

* Duplicate call to max\_id(current->next) can lead to inefficiency.
* Could be optimized.

***2.8. Module 8: Complexity Analysis (complexity.c/complexity.h) :***

**1. isBST**

Verify if a binary tree is a valid Binary Search Tree.

**Inputs:**

* treenode\* root
* int min, int max

**Output:** Boolean (true if valid BST).

**Steps:**

* Recursively ensure all left < root < right.

**Strengths:**

* Fully validates BST structure.

**Limitations:**

* Relies on root->value, which is undefined in your struct (bug). Should use root->id.

**2. insertlog**

Insert a node into BST based on id.

**Inputs:**

* treenode\* root, int id, char\* timestamp, char\* message, int severity

**Output:** New or updated root.

**Steps:**

* Recursively insert left or right by comparing id.

**Strengths:**

* Maintains BST ordering.

**Limitations:**

* No duplicate id handling.

**3. searchlog**

Search for a log by id.

**Input:** treenode\* root, int id  
**Output:** Pointer to matching node or NULL.

**Strengths:**

* Efficient O(log n) search (if balanced).

**4. findmin**

Find the node with the smallest id.

**Input:** treenode\* node  
**Output:** Minimum node.

**Strengths:**

* Needed for deletion.

**5. deletelog**

Delete a log node from BST by id.

**Input:** treenode\* root, int id  
**Output:** Updated root.

**Steps:**

* Handles all deletion cases (0, 1, 2 children).
* Uses findmin for successor replacement.

**Strengths:**

* Fully handles BST deletion scenarios.

**Limitations:**

* Calls deleteLog, but function is named deletelog (case mismatch – bug).

**6. inorder, preorder, postorder**

Traverse and print the BST in respective orders.

**Input:** treenode\* root

**Output:** Printed log info.

**Strengths:**

* Good for displaying tree contents.

**Limitations:**

* No return values (can’t reuse data directly).

*second part :*

***3.1. Modules based on Linked lists and Queues :***

**Linked List & Word Operations (TList & BList) :**

**1. addlist**

Add a new word node to the front of a list.

**Inputs:**

* TList\* list
* char\* word, char\* related

**Output:** Updated list with the new head.

**Strengths:**

* Quick insertion (O(1))
* Automatically calculates characters and vowels.

**Limitations:**

* Allocates memory without validation.

**2. addWord**

Add a word with synonym and antonym to both lists and file.

**Inputs:**

* TList\* syn, TList\* ant
* char\* word, char\* syne, char\* anton

**Output:** Updated synonym list.

**Strengths:**

* Syncs with file (file.txt)
* Adds to both in-memory lists and on-disk.

**3. countvowel**

Count vowels in a string.

**Input:** char\* word  
**Output:** Integer count.

**Strengths:**

* Works with both uppercase and lowercase.

**4. getSynWords, getAntoWords**

Parse file and return a linked list of words and their synonyms or antonyms.

**Input:** FILE\* f  
**Output:** TList\* head

**Strengths:**

* File parsing using delimiters (= and #).

**Limitations:**

* File format must be strict.

**5. getInfWord, getInfWord2**

Display full info (characters, vowels) of a word or related synonym/antonym.

**Strengths:**

* Searches both synonym and antonym lists.

**6. sortWord, sortWord2, sortWord3**

Sort words by:

* Alphabetical order (sortWord)
* Character count (sortWord2)
* Vowel count (sortWord3)

**Strengths:**

* Easy reordering of word lists.

**Limitations:**

* Uses bubble sort (O(n²)).

**7. countWord**

Find and return words containing a certain substring.

**Strengths:**

* Pattern-based filtering.

**8. deleteWord**

Remove a word from both lists and file.

**Strengths:**

* Handles both synonym and antonym removal.
* Updates the file correctly using a temp file.

**Limitations:**

* Deletes only the first match.
* Removes original file directly.

**9. updateWord**

Update synonym or antonym of a word in list and file.

**Strengths:**

* Real-time update in both memory and file.

**Limitations:**

* Creates temp file every time, even for minor changes.

**10. merge**

Merge synonym and antonym lists into a bidirectional list.

**Strengths:**

* Merges using word match

**String Similarity & Palindrome**

**11. callculate\_rate**

Calculate similarity rate (%) between two strings.

**Strengths:**

* Useful for fuzzy matching.

**Limitations:**

* lonstrlen is declared incorrectly (scope issue).

**12. similarWord**

Return list of words similar to the input word by given rate.

**Strengths:**

* Uses callculate\_rate.

**Limitations:**

* Depends on flawed rate calculation function.

**13. ispalindrome**

Check if a string is a palindrome.

**Output:** Returns 1 for true, 0 for false.

**Strengths:**

* Works correctly and efficiently.

**Queue Functions (TQueue)**

**14. enqueue**

Add a node to the rear of a queue.

**Input:** TQueue\* queue, TList\* node

**Strengths:**

* Deep copies node structure.

**Limitations:**

* Doesn't check for memory allocation failure.

**15. toQueue**

Convert BList to a queue of TList.

**Strengths:**

* Merges synonym/antonym into a single string.

**16. syllable**

Convert TList to queue.

**Strengths:**

* Preserves all node fields.

***3.2. Modules based on Stacks (stacks.h/stacks.c) :***

### Stack Operations (TStack).

### 1. ****printStack****

Print all elements in stack.

**Input:** TStack\* stk

**Strengths:**

* Shows all entries in readable format.

### 2. ****addWordStack****

Add a word with synonym/antonym to the stack.

**Inputs:**

* word, syne, anton

**Steps:**

* Create and push new node.

### 3. ****deleteWordStack****

Remove a node with a specific word.

**Input:** TStack\* stk, char\* word

**Method:**

* Pop all elements to temp stack.
* Push back those that don’t match.

### 4. ****updateWordStack****

### Update synonym/antonym for a word.

**Input:**

* word, syne, anton

**Steps:**

* Traverse and update if match is found.

### 5. ****getInfWordStack****

Print details of a specific word.

**Strengths:**

* Displays all info from the node.

### 6. ****getSmallest****

Find the smallest (lexicographically) word.

**Output:** char\*

**Strengths:**

* Simple linear scan.

### 7. ****isPalindromeStack****

Check if word is a palindrome using stack logic.

**Input:** char\* word  
**Output:** Boolean

**Strengths:**

* Demonstrates stack-based reversal comparison.

## Stack Sorting & Conversion

### 8. ****sortWordStack****

Sort words in stack alphabetically.

**Method:**

* Pop to array
* Sort with bubble sort
* Push back in order

**Strengths:**

* Effective sorting mechanism.

### 9. ****StacktoList****

Convert stack to a doubly linked list (BList).

**Steps:**

* Pop nodes from stack.
* Sort alphabetically.
* Insert into BList.

### 10. ****toStack****

Convert a BList to a stack.

**Strengths:**

* Stack reflects order from list.
* Used for data import or conversion.

## Queue Conversion

### 11. ****enqueue****

Add a TList node to the queue.

**Steps:**

* Standard linked list enqueue.

### 12. ****stackToQueue****

Convert a stack to a queue of TList.

**Steps:**

* Reverse stack into temp.
* Create TList from Stacknode.
* Add to queue.

**Strengths:**

* Combines synonym and antonym into related field.

***3.3. Modules based on Binary Search Tree (BST) :***

### 1. toTree Convert a stack into a BST.

**Inputs:**

* TStack \*stk

**Output:**

* TTree\* (BST root)

**Steps:**

* While stack is not empty:
  + Pop stack node.
  + Create tree node.
  + Insert into BST.
  + Free stack node.

**Strengths:**

* Efficient transformation from stack to BST.
* Reuses existing functions.

**Limitations:**

* Order of insertion affects final tree shape.

**2. getInfWordTree**  
Find a node by word in the BST.

**Inputs:**

* TTree \*tr
* char \*word

**Output:**

* TTree\* (matching node or NULL)

**Steps:**

* Compare word with current node.
* Traverse left or right accordingly.
* Return node if matched.

**Strengths:**

* Fast search in balanced BST.

**Limitations:**

* Performance depends on tree shape.

**3. findMin**  
Find the node with the minimum value in BST.

**Inputs:**

* TTree \*root

**Output:**

* TTree\* (minimum node)

**Steps:**

* Traverse to the leftmost node.
* Return node.

**Strengths:**

* Simple and reliable.

**Limitations:**

* Assumes tree is non-empty.

**4. deleteWordBST**  
Delete a node with a given word from the BST.

**Inputs:**

* TTree \*tr
* char \*word

**Output:**

* Updated TTree\*

**Steps:**

* Search for the node to delete.
* Handle three cases: no child, one child, two children.
* For two children, replace with in-order successor.

**Strengths:**

* Correctly maintains BST structure.

**Limitations:**

* Recursive memory usage.
* Unbalanced tree can affect performance.

**5. TraversalBSTinOrder**  
In-order traversal of BST (sorted output).

**Inputs:**

* TTree \*tr

**Output:**

* TTree\* (unchanged)

**Steps:**

* Recursively: left → root → right
* Print each node’s data.

**Strengths:**

* Prints words in lexicographical order.

**Limitations:**

* Read-only operation.

**6. TraversalBSTpostOrder**  
Post-order traversal of BST (children before parent).

**Inputs:**

* TTree \*tr

**Output:**

* TTree\* (unchanged)

**Steps:**

* Recursively: left → right → root
* Print each node’s data.

**Strengths:**

* Useful for deleting or freeing trees.

**Limitations:**

* Traversal order not sorted.

**7. LowestCommonAncestor**  
Find the lowest common ancestor (LCA) of two words in BST.

**Inputs:**

* TTree \*tr
* char \*word1, \*word2

**Output:**

* TTree\* (LCA node)

**Steps:**

* Compare both words to current node.
* If both less → go left.
* If both greater → go right.
* Else, current node is LCA.

**Strengths:**

* Efficient for BST structure.

**Limitations:**

* Assumes both words exist in the tree.

**7. inOrderSuccessor**  
Find the in-order successor of a given word.

**Inputs:**

* TTree \*tr
* char \*word

**Output:**

* TTree\* (successor node or NULL)

**Steps:**

* Search for node.
* If right child exists, find leftmost in right subtree.
* Else, track last ancestor with greater value.

**Strengths:**

* Useful for successor-based operations.

**Limitations:**

* Doesn’t return predecessor.
* Requires existing word.