Optional Part: GUI

Overview

This document provides an in-depth description of the Graphical User Interface (GUI) for a dictionary management system, implemented in C using the Windows API (Win32) in `main.c`. The GUI enables users to interact with dictionary-related functions across three categories: Trees, Linked Lists, Stacks, and Queues, and Recursion. It integrates with wrapper functions in `functions_wrapper.c`, leveraging utilities from `code.c`, `code_utils.c`, `tree_utils.c`, `trees.c`, `recursion_utils.c`, and `Recursion_2.c` to manage data structures like Binary Search Trees (BSTs), linked lists, stacks, and queues. The system operates on a dictionary file (`dictinoary.txt`) with a specific format for words, synonyms, and antonyms.

This updated documentation incorporates detailed insights from the provided source files, offering:

- Comprehensive function breakdowns with algorithms and edge cases.
- Clarification of inter-file dependencies and data flow.
- Notes on memory management, error handling, and performance.
- Enhanced build instructions and troubleshooting tips.
- Suggestions for addressing limitations (e.g., typos, redundant code).

The GUI provides a user-friendly interface for loading dictionaries, manipulating data structures, and viewing results in a read-only text area, with robust error handling and status feedback.

Key Features

- Category-Based Navigation: Select Trees, Linked Lists, Stacks, and Queues, or Recursion via a dropdown menu.
- Dynamic Function Selection: Functions update based on category, with argument fields generated dynamically.
- File Input: A Browse button allows loading `dictinoary.txt` for file-based operations.
- Output Display: Captures `stdout` output in a read-only text area using temporary files.
- Status Bar: Displays states like "Ready" or error messages.
- Responsive Design: Adapts to window resizing for usability.
- Input Validation: Ensures valid inputs (e.g., loaded tree/list) and shows errors.

Dictionary File Format

The dictionary file, named `dictinoary.txt` (note the consistent typo across all source files), stores words with their synonyms and antonyms in a structured format. Example:

```
happy=joyful=content#sad=unhappy
sad=unhappy=gloomy#happy=cheerful
```

- Format: `word=synonym1=synonym2#antonym1=antonym2`
- Components:
  - Word: The primary word (e.g., `happy`).
  - Synonyms: Zero or more synonyms separated by `=` (e.g., `joyful=content`).
  - Antonyms: Zero or more antonyms after `#`, separated by `=` (e.g., `sad=unhappy`).
- Parsing:
  - `code.c`: `getSynWords` and `getAntoWords` use `sscanf` to extract word-synonym or word-antonym pairs.
  - `tree_utils.c` / `trees.c`: `fillTree` uses `strtok` to tokenize lines, handling missing `#` by assuming synonyms only.
- Notes:
  - The typo `dictinoary.txt` is consistent in `code.c` and other files. Users must name the file `dictinoary.txt` or update the code to expect `dictionary.txt`.
  - Lines without `#` are treated as synonym-only entries in `fillTree`.
  - Buffer sizes (e.g., 170–175 bytes in `code.c`) assume reasonable word lengths but may overflow for very long lines.

Code Structure

The GUI is built around `main.c`, supported by source and header files defining data structures, utilities, and wrapper functions. Below is a detailed breakdown of the components and their interactions:

1. Data Structures (`unified_data_structures.h`)

Defined in `unified_data_structures.h`, these structures support dictionary operations:

- TTree2: BST node with:
  - `char word[50]`: Primary word.
  - `WordNode *synonym`: Linked list of synonyms.
  - `WordNode *antonym`: Linked list of antonyms.
  - `TTree2 *left, *right`: Child pointers.
- TList: Singly linked list with:
  - `Node *head`: Points to a `Node` with `char word[50]`, `char relatedWord[50]`, `int charCount`, `int vowelCount`, and `Node *next`.
- TList2: Bidirectional linked list with:
  - `DNode *head, *tail`: `DNode` has `char word[50]`, `char synonym[50]`, `char antonym[50]`, `int charCount`, `int vowelCount`, `DNode *next`, `DNode *prev`.
- TList3: Circular linked list with:

    - `CNode *head, *tail`: `CNode` has `char word[50]`, `char synonym[50]`, `char antonym[50]`, `int charCount`, `int vowelCount`, `CNode *next`.
- TStack: Stack with:
    - `CNode *head`: Same `CNode` as `TList3`.
- TStack2: Character stack with:
    - `TNode *head`: `TNode` has `char c` and `TNode *next`.
- TQueue: Queue with:
    - `QNode *front, *rear`: `QNode` has `char word[50]` (or `TTree2 *treeNode` in `trees.c`) and `QNode *next`.
- WordNode: Linked list node for synonyms/antonyms in `TTree2`:
    - `char word[50]`, `WordNode *next`.

2. File Interactions

- `main.c`: GUI logic, calls `fw_*` functions from `functions_wrapper.c`, redirects `stdout` to temporary files.
- `functions_wrapper.c`: Wraps functions from `code.c`, `tree_utils.c`, and `recursion_utils.c`, handling file operations and data structure management.
- `code.c`: Implements linked list, stack, and queue operations, reading/writing `dictinoary.txt`.
- `code_utils.c`: Provides utility functions (e.g., `createNode`, `countVowels`, `getVowelType`) used by `code.c` and others.
- `tree_utils.c` / `trees.c`: Implement BST operations, with `fillTree` parsing `dictinoary.txt`. `trees.c` adds `BTSMerge`.
- `recursion_utils.c` / `Recursion_2.c`: Implement recursive algorithms and file utilities (e.g., `countWordOccurrence`). `Recursion_2.c` adds `reverseFile`.

Data Flow:

1. User selects a function in the GUI (e.g., `fw_fillTree`).
2. `main.c` calls the wrapper in `functions_wrapper.c`.
3. The wrapper opens `dictinoary.txt` (if needed) and calls the core function (e.g., `fillTree` in `tree_utils.c` or `getSynWords` in `code.c`).
4. Core functions use utilities from `code_utils.c` (e.g., `createNode`) or `recursion_utils.c` (e.g., `isPalindromWord`).
5. Output is captured via `stdout` redirection and displayed in the GUI.

3. GUI Components (`main.c`)

- `WndProc`: Handles messages like `WM_CREATE` (initializes controls), `WM_COMMAND` (dropdown/button events), and `WM_SIZE` (resizes layout).
- `custom_printf`: Redirects `stdout` to a buffer for GUI display.
- `OpenFileBrowser`: Uses `OPENFILENAME` to select `dictinoary.txt`.
- `ExecuteFunction`: Runs the selected `fw_*` function, capturing output in a temporary file (e.g., `tree_output.txt`).
- `UpdateArgsContainer`: Dynamically creates `Edit` controls for function arguments based on `FunctionDefinition` in `main.c`.
- Controls:
    - Category ComboBox: Lists `Trees`, `Code`, `Recursion`.
    - Function ComboBox: Populates with `fw_*` functions.
    - Browse Button: For file selection.
    - Run Button: Triggers `ExecuteFunction`.
    - Result Text Area: Read-only `Edit` control for output.
    - Status Bar: Displays state/error messages.

4. Function Categories

Defined via `FunctionCategory` enum in `main.c`:

- CATEGORY_TREES: BST operations (e.g., `fw_fillTree`, `fw_deleteWordBST`).
- CATEGORY_CODE: Linked lists, stacks, queues (e.g., `fw_getSynWords`, `fw_syllable`).
- CATEGORY_RECURSION: Recursive algorithms (e.g., `fw_wordPermutation`, `fw_longestSubseqWord`).

Detailed Implementation

Below is an in-depth look at key functions, their algorithms, and implementation details, grouped by category. Pseudo-code or flow descriptions are included for clarity, focusing on critical functions from `code.c`, `tree_utils.c`, `trees.c`, `recursion_utils.c`, and `Recursion_2.c`.

1. Trees

`TTree2* fw_fillTree(const char *filename)` (Wraps `fillTree` in `tree_utils.c`)

- Purpose: Builds a BST (`TTree2`) from `dictinoary.txt`.
- Implementation:
    - Opens `filename` (`dictinoary.txt`) in read mode.
    - Reads lines (up to 500 bytes) using `fgets`.
    - For each line:
        - Removes trailing newline.
        - Checks for `#` delimiter using `strchr`.
        - If `#` is absent, assumes synonyms only; uses `strtok` with `=` to extract word and synonyms.
        - If `#` is present, splits at `#`, then uses `strtok` with `=` for synonyms and antonyms.
        - Creates `WordNode` lists for synonyms and antonyms using `insertAtEnd`.
        - Inserts into BST using `insertBST2`.
    - Prints insertion details (word, synonym count, antonym count) to `stdout`.
    - Returns the BST root or `NULL` on file error.

- Pseudo-Code:

```
FUNCTION fillTree(filename):
    OPEN file as f
    IF f is NULL: PRINT error; RETURN NULL
    root = NULL
    WHILE fgets(line):
        REMOVE newline from line
        hashpos = FIND '#' in line
        IF hashpos is NULL:
            mainWord = strtok(line, '=')
            synonyms = NULL
            WHILE token = strtok(NULL, '='): insertAtEnd(&synonyms, token)
            root = insertBST2(root, mainWord, synonyms, NULL)
        ELSE:
            SET hashpos to '\0'
            mainWord = strtok(line, '=')
            synonyms = NULL
            WHILE token = strtok(NULL, '='): insertAtEnd(&synonyms, token)
            antonyms = NULL
            ant = hashpos + 1
            WHILE token = strtok(ant, '='): insertAtEnd(&antonyms, token)
            root = insertBST2(root, mainWord, synonyms, antonyms)
    CLOSE f
    RETURN root
```

- Edge Cases:
  - Empty file: Returns `NULL` and prints "Tree is empty".
  - Missing main word: Skips line and prints error.
  - Buffer overflow: Line buffer (500 bytes) may truncate long lines.
- Memory Management: Allocates `TTree2` and `WordNode` nodes; user must free the tree (not implemented in `fillTree`).
- Performance: O(n log n) average case for BST insertion, where n is the number of words.

`TTree2* fw_deleteWordBST(TTree2 *tr, char *word)` (Wraps `deleteWordBST` in `tree_utils.c`)

- Purpose: Removes a word from the BST.
- Implementation:
  - Recursively searches for the word using `strcmp`.
  - Handles three cases:
    1. No children: Frees the node, returns `NULL`.
    2. One child: Frees the node, returns the child.
    3. Two children: Finds the in-order successor (minimum node in right subtree via `findMinNode2`), copies its data (`word`, `synonym`, `antonym`), and deletes the successor.
  - Uses `copyNodeList` to duplicate synonym/antonym lists.
- Edge Cases:
  - Word not found: Returns unchanged tree.
  - Empty tree: Returns `NULL`.
- Memory Management: Frees the deleted node; `copyNodeList` allocates new `WordNode` lists.
- Performance: O(h) where h is tree height (O(log n) for balanced, O(n) for skewed).

`TTree2* BTSMerge(TTree2 *tr1, TTree2 *tr2)` (`trees.c`)

- Purpose: Merges two BSTs into a balanced BST.
- Implementation:
  - Stores in-order traversals of both trees in queues (`q1`, `q2`) using `StoreBSTinOrder`.
  - Merges queues into a sorted queue (`merged`) by comparing words.
  - Converts the merged queue to an array.
  - Selects the middle element as the root and builds left/right subtrees using `insertBST2`.
- Pseudo-Code:

```
FUNCTION BTSMerge(tr1, tr2):
    IF tr1 is NULL: RETURN tr2
    IF tr2 is NULL: RETURN tr1
    q1 = StoreBSTinOrder(tr1)
    q2 = StoreBSTinOrder(tr2)
    merged = createQueue()
    WHILE q1 and q2 not empty:
        IF q1.front.word ≤ q2.front.word:
            enqueue(merged, dequeue(q1))
        ELSE:
            enqueue(merged, dequeue(q2))
    WHILE q1 not empty: enqueue(merged, dequeue(q1))
    WHILE q2 not empty: enqueue(merged, dequeue(q2))
    count = lenQueue(merged)
    IF count = 0: RETURN NULL
    array = CONVERT merged to array
    mid = count / 2
    root = createTreeNode2(array[mid].word, array[mid].synonym, array[mid].antonym)
    FOR i from 0 to mid-1: insertBST2(root, array[i].word, array[i].synonym, array[i].antonym)
    FOR i from mid+1 to count-1: insertBST2(root, array[i].word, array[i].synonym, array[i].antonym)
    FREE array
    RETURN root
```

- Edge Cases:
  - One tree empty: Returns the other tree.
  - Both empty: Returns `NULL`.
  - Duplicate words: `insertBST2` skips duplicates.
- Memory Management: Allocates queue nodes, array, and new BST nodes; frees array but not queue nodes (potential leak).
- Performance: O(n log n) for traversal and insertion, where n is total nodes.

2. Linked Lists, Stacks, Queues

`TList* fw_getSynWords(const char *filename)` (Wraps `getSynWords` in `code.c`)

- Purpose: Loads synonyms from `dictinoary.txt` into a `TList`.
- Implementation:
  - Opens `filename` in read mode.
  - Allocates a `TList` with `head = NULL`.
  - Reads lines (170 bytes) using `fgets`.
  - Uses `sscanf` with format `%49[^=] = %49[^#]` to extract word and synonym.
  - Calls `addNodeAtEnd` to append each pair to the list.
  - Closes the file and returns the list.
- Edge Cases:
  - File not found: Prints error via `perror` and returns `NULL`.
  - Invalid line format: Skips lines where `sscanf` fails.
- Memory Management: Allocates `TList` and `Node` structures; user must free the list.
- Performance: O(n) for reading n lines, plus O(1) per node insertion.

`TList* fw_sortWord(TList *syn)` (Wraps `sortWord` in `code.c`)

- Purpose: Sorts a `TList` alphabetically by word.
- Implementation:
  - Uses selection sort:
    - Iterates through the list (`current`).
    - For each `current`, finds the minimum node (`minnode`) by `strcmp`.
    - Swaps data (`word`, `relatedWord`, `charCount`, `vowelCount`) if needed using `swapData`.
  - Returns the sorted list.
- Pseudo-Code:

```
FUNCTION sortWord(syn):
    IF syn.head is NULL or syn.head.next is NULL: RETURN syn
    current = syn.head
    WHILE current.next:
        minnode = current
        temp = current
        WHILE temp:
            IF strcmp(temp.word, minnode.word) < 0: minnode = temp
            temp = temp.next
        IF minnode ≠ current: swapData(minnode, current)
        current = current.next
    RETURN syn
```

- Edge Cases:
  - Empty or single-node list: Returns unchanged.
  - Equal words: Stable sort (maintains order).

- Memory Management: No additional allocation; swaps data in-place.
- Performance: O(n²) due to selection sort, where n is list length.

`TQueue* fw_syllable(TList *syn)` (Wraps `syllable` in `code.c`)

- Purpose: Creates a queue of words sorted by syllable count.
- Implementation:
  - Counts list length by traversing `syn->head`.
  - Allocates an array of `Node*` pointers.
  - Populates the array with list nodes.
  - Sorts the array using `qsort` with `compareNodes`, which compares syllable counts (via `count_syllables`) and breaks ties with `strcmp`.
  - Creates a `TQueue` and enqueues words from the sorted array.
  - Frees the array and returns the queue.
- Pseudo-Code:

```
FUNCTION syllable(syn):
    length = 0
    FOR temp = syn.head: length++
    arr = ALLOCATE Node* array of size length
    FOR i = 0 to length-1: arr[i] = temp; temp = temp.next
    qsort(arr, length, sizeof(Node*), compareNodes)
    queue = createQueue()
    FOR i = 0 to length-1: enqueue(queue, arr[i].word)
    FREE arr
    RETURN queue
```

- Edge Cases:
  - Empty list: Returns empty queue.
  - Words with same syllable count: Sorted alphabetically.
- Memory Management: Allocates array and queue nodes; frees array but not queue nodes.
- Performance: O(n log n) due to `qsort`, where n is list length.

3. Recursion

`void fw_wordPermutation(char *word)` (Wraps `wordPermutation` in `recursion_utils.c`)

- Purpose: Prints all permutations of a word.
- Implementation:
  - Uses recursive `permute` with a helper `loop`:
    - `permute(word, index)`: If `index` equals word length, prints the word; else calls `loop`.
    - `loop(i, end, index, word)`: Swaps characters at `index` and `i`, recurses, and swaps back.
  - Generates all possible permutations by swapping characters.
- Pseudo-Code:

```
FUNCTION wordPermutation(word):
    permute(word, 0)

FUNCTION permute(word, index):
    IF index = len(word): PRINT word; RETURN
    loop(index, len(word), index, word)

FUNCTION loop(i, end, index, word):
    IF i ≥ end: RETURN
    SWAP word[index], word[i]
    permute(word, index + 1)
    SWAP word[index], word[i]
    loop(i + 1, end, index, word)
```

- Edge Cases:
  - Empty word: No output.
  - Single character: Prints the character.
- Memory Management: Modifies word in-place; no additional allocation.
- Performance: O(n!) for n-character word due to generating all permutations.

`int fw_longestSubseqWord(char *word1, char *word2)` (Wraps `longestSubseqWord` in `recursion_utils.c`)

- Purpose: Returns the length of the longest common subsequence (LCS).
- Implementation:
  - Recursive approach:
    - If either word is empty, returns 0.
    - If first characters match, adds 1 and recurses on remaining strings.
    - Else, takes the maximum of LCS excluding first character of `word1` or `word2`.
- Pseudo-Code:

```
FUNCTION longestSubseqWord(word1, word2):
    IF word1[0] = '\0' OR word2[0] = '\0': RETURN 0
    IF word1[0] = word2[0]:
        RETURN 1 + longestSubseqWord(word1 + 1, word2 + 1)
    ELSE:
        RETURN max(longestSubseqWord(word1, word2 + 1), longestSubseqWord(word1 + 1, word2))
```

- Edge Cases:
  - Empty strings: Returns 0.
  - No common characters: Returns 0.
- Memory Management: No allocation; recursive stack space.
- Performance: O(2^(m+n)) for strings of length m and n (exponential; could be optimized with dynamic programming).

`void reverseFile(const char *inputFilename, const char *outputFilename)` (`Recursion_2.c`)

- Purpose: Reverses the lines of a file.
- Implementation:
  - Opens input and output files.
  - Counts lines by reading with `fgets`.
  - Creates a stack (`Stack`) with capacity equal to line count.
  - Pushes each line onto the stack.
  - Pops lines and writes to the output file, reversing order.
  - Frees stack memory.
- Edge Cases:
  - File not found: Prints error via `perror`.
  - Empty file: Creates empty output file.
- Memory Management: Allocates stack and duplicates lines with `strdup`; frees all memory.
- Performance: O(n) for n lines, plus O(n) space for stack.

Build Instructions

To compile the GUI, use the provided `build.bat` with GCC. Here's a detailed guide:

1. Prerequisites:
   - GCC Compiler: Install MinGW (e.g., via MSYS2 or standalone). Ensure `gcc` is in the system PATH.
   - Windows SDK: Required for Win32 API headers (`windows.h`, `commctrl.h`).
   - Files: Ensure all source files (`main.c`, `functions_wrapper.c`, `code.c`, `code_utils.c`, `tree_utils.c`, `trees.c`, `recursion_utils.c`, `Recursion_2.c`) and headers (`functions_wrapper.h`, `unified_data_structures.h`, `code.h`, `code_utils.h`, `tree_utils.h`, `trees.h`, `recursion_utils.h`, `Recursion_2.h`) are in the project directory.
   - Dictionary File: Create `dictinoary.txt` with the correct format.
2. Update `build.bat`:
   - The original `build.bat` may exclude `trees.c` and `Recursion_2.c`. Modify it to include all files:

     ```
     @echo off
     gcc -Wall -o GUI main.c functions_wrapper.c code.c code_utils.c tree_utils.c trees.c recursion_utils.c Recursion_2.c -mw
     if %ERRORLEVEL% == 0 (
         echo Build complete. Run GUI.exe to start the application.
     ) else (
         echo Build failed.
     )
     pause
     ```

   - Flags:
     - `-Wall`: Enables warnings.
     - `-mwindows`: Links Win32 API for GUI.
     - `-lcomctl32`: Links common controls library.
3. Run the Build:
   - Open a command prompt in the project directory.
   - Execute: `build.bat`
   - Output: `GUI.exe` if successful.
4. Verify:
   - Run `GUI.exe` to launch the GUI.
   - Ensure `dictinoary.txt` is in the working directory or a path accessible via the Browse button.

Usage Instructions

1. Launch: Run `GUI.exe` to open the "Tree Functions GUI".
2. Select Category: Choose Trees, Code, or Recursion from the dropdown.
3. Select Function: Pick a function (e.g., `fillTree`, `sortWord`).
4. Enter Arguments:

    - For file inputs, click Browse to select `dictinoary.txt`.
    - Enter text or numeric arguments in provided fields.
5. Run: Click Run to execute; results appear in the text area.
6. Check Status: View the status bar for feedback (e.g., "Ready").

Example:

- Load a BST: Select Trees > `fillTree`, browse for `dictinoary.txt`, click Run.
- Sort synonyms: Select Code > `sortWord`, ensure a list is loaded, click Run.
- Generate permutations: Select Recursion > `wordPermutation`, enter a word (e.g., "cat"), click Run.

Potential Issues & Limitations

- Typo in `dictinoary.txt`:
    - Consistent across `code.c`, `tree_utils.c`, etc., suggesting intentional use.
    - Recommendation: Rename to `dictionary.txt` and update all file references (e.g., in `getSynWords`, `fillTree`).
- Function Overlap:
    - `trees.c` vs. `tree_utils.c`: Nearly identical implementations (e.g., `fillTree`, `deleteWordBST`). `tree_utils.c` is used by `functions_wrapper.c`, but `trees.c` adds `BTSMerge`.
    - `Recursion_2.c` vs. `recursion_utils.c`: Duplicate functions except for `reverseFile`.
    - Impact: Code redundancy increases maintenance effort.
- Incomplete Implementations:
    - `fw_toStack` and `fw_stackToQueue` in `functions_wrapper.c` are simplified/dummy implementations, limiting functionality.
    - Example: `fw_toStack` creates an empty stack instead of converting a `TList`.
- Memory Management:
    - Functions like `getSynWords`, `fillTree`, and `syllable` allocate memory without cleanup functions, risking leaks.
    - Example: `BTSMerge` frees the array but not queue nodes.
- Buffer Sizes:
    - `code.c` uses fixed buffers (e.g., 170–175 bytes for lines, 50 bytes for words). Long lines/words may cause overflows.
    - Recommendation: Use dynamic buffers or validate input lengths.
- Performance:
    - `sortWord` uses O(n²) selection sort; could use merge sort for O(n log n).
    - `longestSubseqWord` is O(2^n); dynamic programming would reduce to O(mn).
- Error Handling:
    - File errors (e.g., `fopen` failure) use `perror` or `printf` but don't propagate to GUI consistently.
    - Input validation (e.g., empty strings) is minimal in some functions.

Future Improvements

- Fix Typo: Update all references to `dictinoary.txt` to `dictionary.txt`.
- Consolidate Code:
    - Merge `trees.c` into `tree_utils.c`, keeping `BTSMerge`.
    - Merge `Recursion_2.c` into `recursion_utils.c`, retaining `reverseFile`.
- Complete Wrappers:
    - Implement `fw_toStack` to convert `TList` to `TStack` by iterating and pushing nodes.
    - Enhance `fw_stackToQueue` to transfer stack elements correctly.
- Optimize Algorithms:
    - Replace selection sort in `sortWord` with merge sort.
    - Use dynamic programming for `longestSubseqWord`.
- Dynamic Buffers: Replace fixed buffers with dynamic allocation (e.g., `realloc`) for lines/words.
- Memory Cleanup: Add functions to free `TList`, `TTree2`, `TQueue`, etc., and call them after use.
- GUI Enhancements:
    - Add tooltips with function descriptions.
    - Display error dialogs for invalid inputs.
    - Support multiple file formats (e.g., JSON, CSV).
- Cross-Platform: Port to Qt or SDL for non-Windows support.

Troubleshooting

- Build Fails:
    - Cause: Missing files or incorrect GCC setup.
    - Fix: Verify all source/header files are present. Install MinGW and add to PATH.
- No Output:
    - Cause: `dictinoary.txt` missing or malformed.
    - Fix: Ensure file exists and follows format. Check GUI status bar for errors.
- Crashes:
    - Cause: Buffer overflow or memory leak.
    - Fix: Validate input lengths. Use a debugger (e.g., gdb) to trace crashes.
- Function Not Working:
    - Cause: Simplified wrappers (e.g., `fw_toStack`).
    - Fix: Check `functions_wrapper.c` for dummy implementations and replace with full logic.