Mini Search Engine Using Binary Search Tree

Data Structures – Final Project Report

Student: 四電子四甲 B11102034 童彥智

Contents

Contents 1
1. Problem Definition 2
2. Methodology & System Design 2
    2.1 Overview 2
    2.2 BST Structure 3
    2.3 Major Functions 3
        2.3.1 insertNode() 3
        2.3.2 searchNode() 3
        2.3.3 findClosest() 4
        2.3.4 deleteNode() 4
        2.3.5 inorder() 4
        2.3.6 loadDictionary() 5
3. BST Algorithms Explanation 5
    3.1 Case-Insensitive Comparison 5
    3.2 Insertion Algorithm 5
    3.3 Search Algorithm 6
    3.4 Closest-Word Suggestion Algorithm 6
    3.5 Deletion Algorithm 6
4. Code 6
    4.1 bst.h 6
    4.2 bst.c 9
    4.3 dict_loader.h 22
    4.4 dict_loader.c 23
    4.5 main.c 28
5. Results 38
    5.1 Program main menu 38
    5.2 Load dictionary 38
    5.3 Show all words 39
    5.4 Search result (found) 39
    5.5 Search result (not found) + closest suggestions 39
    5.6 Insert a new word 40
    5.7 Delete a word 40
6. Discussion 40
7. Conclusion 41

1. Problem Definition

Large collections of word–meaning pairs require an efficient data structure to support insertion, deletion, lookup, and sorted traversal.
Traditional linear search becomes inefficient when the dictionary grows large.

Therefore, this project aims to build a Mini Search Engine that:

1. Loads dictionary data from .txt files
2. Stores all entries using a case-insensitive Binary Search Tree (BST)
3. Supports
    - Searching
    - Insertion
    - Deletion
    - Sorted listing
    - Path tracking
    - Closest-word suggestion (predecessor & successor)
4. Provides a simple and interactive text-based user interface

The objective is to demonstrate practical mastery of key Data Structures concepts, specifically BST operations and string handling in C.

2. Methodology & System Design

2.1 Overview

The entire system is composed of:

- bst.c / bst.h — BST implementation
- dict_loader.c / dict_loader.h — dictionary file parsing
- main.c — user interface and program flow

The BST uses case-insensitive string comparison for ordering, ensuring words

such as *Bible*, *bible*, and *BIBLE* map to the same logical position.

2.2 BST Structure

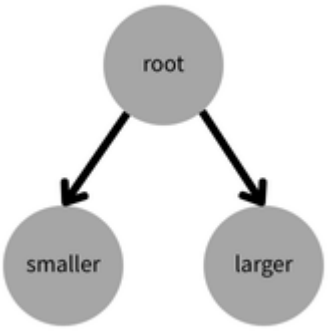


Figure 1 BST Structure

Each node contains:

```
typedef struct TreeNode {
    Entry data;         // word + meaning
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;
```

The dictionary is stored completely in memory.

2.3 Major Functions

2.3.1 insertNode()

- Inserts a new word or updates existing meaning
- Maintains BST property
- Uses case-insensitive comparison

2.3.2 searchNode()

- Searches for a word

- Tracks search path: nodes visited along the way
- Returns comparison count

Output example:

```
Search path: accord -> action -> apple
Comparisons: 3
```

2.3.3 findClosest()

When a word is not found:

- predecessor → largest word smaller than query
- successor → smallest word larger than query

Example:

```
Not found.
Closest suggestions:
  Previous: action
  Next: actress
```

2.3.4 deleteNode()

Standard BST delete implementation with three cases:

1. Node has no children → remove directly
2. Node has one child → replace with child
3. Node has two children → replace with inorder successor

2.3.5 inorder()

Prints dictionary in alphabetical order.

2.3.6 loadDictionary()

Reads .txt file line by line:

```
word<TAB>meaning
```

Example:

```
accord    一致、符合
Bible     聖經
X-ray     X光
```

3. BST Algorithms Explanation

This section describes the logic behind each core BST operation.

3.1 Case-Insensitive Comparison

```
while (*a && *b) {
    ca = tolower(*a);
    cb = tolower(*b);
```

avoids ASCII ordering problems (e.g., uppercase before lowercase).

3.2 Insertion Algorithm

1. Compare new word with current node (case-insensitive)
2. Traverse left or right
3. Insert when reaching a NULL pointer

Time complexity: O(h)

3.3 Search Algorithm

Same traversal logic as insertion.

Additionally:

- Each visited node is recorded in path[][]
- Each comparison increments the counter
- Useful for illustrating search efficiency

3.4 Closest-Word Suggestion Algorithm

During search:

- If moving left → update successor
- If moving right → update predecessor

Works even if the word does not exist.

3.5 Deletion Algorithm

Case 1: leaf → delete

Case 2: one child → replace

Case 3: two children → replace with successor

4. Code

4.1 bst.h

```
#ifndef BST_H
#define BST_H
#include <stddef.h>

/* ------------------------------------------------------------
```

```
 * Constants for maximum string length.
 * These values define how long a word or meaning can be stored.
 * ------------------------------------------------------------*/
#define MAX_WORD_LEN     128
#define MAX_MEANING_LEN  512
#define MAX_PATH_LEN     256

/* ------------------------------------------------------------
 * Structure: Entry
 * Purpose  : Stores a dictionary entry (word + meaning)
 * ------------------------------------------------------------*/
typedef struct Entry {
    char word[MAX_WORD_LEN];        // the vocabulary word
    char meaning[MAX_MEANING_LEN];  // the explanation / definition
} Entry;

/* ------------------------------------------------------------
 * Structure: TreeNode
 * Purpose  : Node for Binary Search Tree (BST)
 * Notes    :
 *   - left subtree  contains words alphabetically smaller (case-insensitive)
 *   - right subtree contains words alphabetically larger
 * ------------------------------------------------------------*/
typedef struct TreeNode {
    Entry data;              // the stored dictionary entry
    struct TreeNode* left;   // pointer to left child
    struct TreeNode* right;  // pointer to right child
} TreeNode;

/* ------------------------------------------------------------
```

/

```
36. * Function: insertNode
37. * Purpose : Insert a word into BST (or update meaning if key already
    exists)
38. * Inputs  : root   - pointer to the BST root pointer
39. *          e      - entry containing the new word + meaning
40. * ------------------------------------------------------------*/
41. void insertNode(TreeNode** root, const Entry* e);
42.
43. /* ------------------------------------------------------------
44. * Function: inorder
45. * Purpose : Print all words in alphabetically sorted order
46. * Inputs  : root   - pointer to BST root
47. * Notes   : Uses inorder traversal to guarantee sorted order
48. * ------------------------------------------------------------*/
49. void inorder(TreeNode* root);
50.
51. /* ------------------------------------------------------------
52. * Function: searchNode
53. * Purpose : Search for a word in BST (case-insensitive)
54. * Inputs  : root     - BST root
55. *          key      - word to search
56. *          path[][] - array storing each visited node's word
57. *          path_len - length of search path
58. *          comparisons - number of comparisons performed
59. * Returns : pointer to found TreeNode, or NULL if not found
60. * ------------------------------------------------------------*/
61. TreeNode* searchNode(TreeNode* root,
62.                      const char* key,
63.                      char path[][MAX_WORD_LEN],
64.                      int* path_len,
65.                      int* comparisons);
```

```
66.
67./* ------------------------------------------------------------
68. * Function: findClosest
69. * Purpose : When search fails, suggest:
70. *           - predecessor (largest word smaller than key)
71. *           - successor   (smallest word larger than key)
72. * ------------------------------------------------------------*/
73.void findClosest(TreeNode* root,
74.                const char* key,
75.                TreeNode** pred,
76.                TreeNode** succ);
77.
78./* ------------------------------------------------------------
79. * Function: deleteNode
80. * Purpose : Remove a word from BST (handles all 3 delete cases)
81. * ------------------------------------------------------------*/
82.void deleteNode(TreeNode** root, const char* key);
83.
84./* ------------------------------------------------------------
85. * Function: freeTree
86. * Purpose : Free all nodes in the BST (postorder deletion)
87. * ------------------------------------------------------------*/
88.void freeTree(TreeNode* root);
89.
90.#endif
```

4.2 bst.c

```
1. #include "bst.h"
2. #include <stdio.h>      // for printf (in inorder)
3. #include <stdlib.h>     // for malloc, free
```

```c
#include <string.h>     // for strlen, etc.
#include <ctype.h>      // for tolower

/* ============================================================
 * Helper Function: ciCompare
 * ------------------------------------------------------------
 * Purpose : Case-insensitive string comparison for dictionary order.
 *           Similar to strcmp(), but ignores letter case.
 * Inputs  : a, b - C strings to compare
 * Returns : < 0  if a < b  (alphabetically smaller)
 *           = 0  if a == b
 *           > 0  if a > b
 *
 * Note    : We convert both characters to lowercase before comparison.
 *           This avoids the ASCII issue where 'A' < 'a' etc.
 * ============================================================*/
static int ciCompare(const char* a, const char* b) {
    unsigned char ca, cb;

    /* Compare character-by-character */
    while (*a && *b) {
        /* Convert both to lowercase */
        ca = (unsigned char)tolower((unsigned char)*a);
        cb = (unsigned char)tolower((unsigned char)*b);

        if (ca < cb) return -1;   // a is smaller
        if (ca > cb) return 1;    // a is larger

        ++a;
        ++b;
```

```
34.     }
35.
36.     /* If we reach here, at least one string ended */
37.     if (*a == *b) {
38.         return 0;                // both ended at same time → equal
39.     }
40.     return *a ? 1 : -1;          // if a still has chars → a > b; else a < b
41. }
42.
43. /* ============================================================
44.  * Function: insertNode
45.  * ------------------------------------------------------------
46.  * Purpose : Insert a new Entry into the BST, or update meaning if the word
47.  *           already exists (case-insensitive).
48.  * Inputs  : root  - address of the BST root pointer
49.  *           e     - pointer to the Entry to insert
50.  * Behavior:
51.  *   - If the tree is empty → create new root node.
52.  *   - Otherwise, traverse down the tree:
53.  *       - If e->word < current word → go to left child.
54.  *       - If e->word > current word → go to right child.
55.  *       - If equal (ignoring case)  → update the existing node's Entry.
56.  * ============================================================*/
57. void insertNode(TreeNode** root, const Entry* e) {
58.     TreeNode* cur;
59.     int cmp;
60.
61.     if (root == NULL || e == NULL) return;
```

```
62.
63.     /* If tree is empty, allocate a new root node */
64.     if (*root == NULL) {
65.         TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
66.         if (!node) {
67.             /* Allocation failed */
68.             return;
69.         }
70.         node->data = *e;          // copy entry data (word + meaning)
71.         node->left = node->right = NULL;
72.         *root = node;             // update root pointer
73.         return;
74.     }
75.
76.     /* Otherwise, find the correct position to insert or update */
77.     cur = *root;
78.     while (1) {
79.         /* Compare new word e->word with current node's word (case-ins
    ensitive) */
80.         cmp = ciCompare(e->word, cur->data.word);
81.
82.         if (cmp < 0) {
83.             /* New word is alphabetically smaller → go left */
84.             if (cur->left == NULL) {
85.                 /* Left child is empty → insert here */
86.                 TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
87.                 if (!node) return;
88.                 node->data = *e;
89.                 node->left = node->right = NULL;
90.                 cur->left = node;
91.                 break;
```

```
92.             }
93.             cur = cur->left;      // continue traversing left
94.         }
95.         else if (cmp > 0) {
96.             /* New word is alphabetically larger → go right */
97.             if (cur->right == NULL) {
98.                 /* Right child is empty → insert here */
99.                 TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
100.                  if (!node) return;
101.                  node->data = *e;
102.                  node->left = node->right = NULL;
103.                  cur->right = node;
104.                  break;
105.              }
106.              cur = cur->right;     // continue traversing right
107.          }
108.          else {
109.              /* cmp == 0 → word already exists (ignoring case).
110.               * Overwrite the meaning with the new entry.
111.               */
112.              cur->data = *e;
113.              break;
114.          }
115.      }
116. }
117.
118. /* ================================================================
119.  * Function: inorder
120.  * ----------------------------------------------------------------
121.  * Purpose : Print all dictionary entries in sorted order (A → Z).
```

```
122.  * Inputs  : root - pointer to BST root
123.  * Behavior:
124.  *   - Inorder traversal:
125.  *        1. traverse left subtree
126.  *        2. visit current node
127.  *        3. traverse right subtree
128.  *   - Because BST is ordered, inorder prints all entries in sorted
      order.
129.  * ============================================================*/
130. void inorder(TreeNode* root) {
131.     if (!root) return;
132.
133.     /* 1. Visit left subtree */
134.     inorder(root->left);
135.
136.     /* 2. Visit current node (print word + meaning) */
137.     printf("%s : %s\n", root->data.word, root->data.meaning);
138.
139.     /* 3. Visit right subtree */
140.     inorder(root->right);
141. }
142.
143. /* ============================================================
144.  * Function: searchNode
145.  * ------------------------------------------------------------
146.  * Purpose : Search for a word in the BST (case-insensitive) while r
      ecording
147.  *          the path and the number of comparisons.
148.  * Inputs  : root       - BST root
149.  *          key        - word to search
150.  *          path[][]    - 2D array to store visited node words
```

```
 *           path_len    - pointer to length of path
 *           comparisons - pointer to comparison count
 * Returns : pointer to found TreeNode, or NULL if not found.
 *
 * Example output usage (in main):
 *   Search path: accord -> action -> apple
 *   Comparisons: 3
 * ============================================================*/
TreeNode* searchNode(TreeNode* root,
                     const char* key,
                     char path[][MAX_WORD_LEN],
                     int* path_len,
                     int* comparisons) {
    TreeNode* cur;

    /* Initialize path length and comparison count */
    if (path_len) *path_len = 0;
    if (comparisons) *comparisons = 0;

    if (!key) return NULL;

    cur = root;
    while (cur) {
        /* Record current node's word into path (if buffer availabl
e) */
        if (path && path_len && *path_len < MAX_PATH_LEN) {
            /* Copy current word into path[*path_len] */
            strncpy(path[*path_len], cur->data.word, MAX_WORD_LEN -
1);
            path[*path_len][MAX_WORD_LEN - 1] = '\0';  // ensure null
-termination
```

```
179.             (*path_len)++;
180.         }
181.
182.         /* Count this comparison */
183.         if (comparisons) (*comparisons)++;
184.
185.         /* Compare search key with current node's word (case-insensi
    tive) */
186.         int cmp = ciCompare(key, cur->data.word);
187.
188.         if (cmp == 0) {
189.             /* Found the key */
190.             return cur;
191.         } else if (cmp < 0) {
192.             /* key < current word → go left */
193.             cur = cur->left;
194.         } else {
195.             /* key > current word → go right */
196.             cur = cur->right;
197.         }
198.     }
199.
200.     /* Reached NULL → key not found in BST */
201.     return NULL;
202. }
203.
204. /* ============================================================
205.  * Helper Function: findMin
206.  * ------------------------------------------------------------
207.  * Purpose : Find the node with the minimum word in a given subtree.
208.  * Inputs  : node - root of the subtree
```

```
209.  * Returns : pointer to the node with the smallest key in this subtr
      ee.
210.  *
211.  * Note    : We keep going to the left child until there is no more
      left.
212.  * ============================================================*/
213. static TreeNode* findMin(TreeNode* node) {
214.     if (!node) return NULL;
215.     while (node->left) {
216.         node = node->left;
217.     }
218.     return node;
219. }
220.
221. /* ============================================================
222.  * Function: findClosest
223.  * ------------------------------------------------------------
224.  * Purpose : When a search fails (word not found), this function sug
      gests:
225.  *            - pred : predecessor  (largest word smaller than key)
226.  *            - succ : successor    (smallest word larger than key)
227.  * Inputs  : root - BST root
228.  *           key  - word we tried to search
229.  * Outputs : *pred, *succ - pointers to closest nodes (may be NULL)
230.  *
231.  * Algorithm idea:
232.  *   - Traverse the tree similarly to search:
233.  *       * If key < current word:
234.  *           - current node is a candidate successor
235.  *           - move to left child
236.  *       * If key > current word:
```

```
237.  *          - current node is a candidate predecessor
238.  *          - move to right child
239.  * ============================================================*/
240. void findClosest(TreeNode* root,
241.                  const char* key,
242.                  TreeNode** pred,
243.                  TreeNode** succ) {
244.     TreeNode* cur;
245. 
246.     /* Initialize output pointers */
247.     if (pred) *pred = NULL;
248.     if (succ) *succ = NULL;
249.     if (!key) return;
250. 
251.     cur = root;
252.     while (cur) {
253.         int cmp = ciCompare(key, cur->data.word);
254. 
255.         if (cmp < 0) {
256.             /* key < current word:
257.              *   - current node could be the successor
258.              *   - we move left to find a possibly smaller one
259.              */
260.             if (succ) *succ = cur;
261.             cur = cur->left;
262.         }
263.         else if (cmp > 0) {
264.             /* key > current word:
265.              *   - current node could be the predecessor
266.              *   - we move right to find a possibly larger one
```

```
267.              */
268.             if (pred) *pred = cur;
269.             cur = cur->right;
270.         }
271.         else {
272.             /* cmp == 0 (exact match) - usually we call this function
273.              * only when NOT found, but we keep it for completeness.
274.              */
275.             break;
276.         }
277.     }
278. }
279.
280. /* ============================================================
281.  * Function: deleteNode
282.  * ------------------------------------------------------------
283.  * Purpose : Remove a word from the BST, handling all three standard
    cases:
284.  *           Case 1: Node has no children
285.  *           Case 2: Node has one child
286.  *           Case 3: Node has two children
287.  * Inputs  : root - address of root pointer of the BST / subtree
288.  *           key  - word to delete (case-insensitive)
289.  *
290.  * Details:
291.  *   - We locate the node using case-insensitive comparison.
292.  *   - Then handle based on how many children the node has.
293.  * ============================================================*/
294. void deleteNode(TreeNode** root, const char* key) {
295.     int cmp;
296.
```

```
297.     if (!root || !*root || !key) return;
298. 
299.     /* Compare target key with current node's word */
300.     cmp = ciCompare(key, (*root)->data.word);
301. 
302.     if (cmp < 0) {
303.         /* key < current word → go left */
304.         deleteNode(&((*root)->left), key);
305.     }
306.     else if (cmp > 0) {
307.         /* key > current word → go right */
308.         deleteNode(&((*root)->right), key);
309.     }
310.     else {
311.         /* ----------------------------------------------------
312.          * We found the node to delete: *root
313.          * Now handle the 3 deletion cases.
314.          * ----------------------------------------------------*/
315. 
316.         /* Case 1: No children (leaf node) */
317.         if (!(*root)->left && !(*root)->right) {
318.             free(*root);          // free the node
319.             *root = NULL;         // set pointer to NULL
320.         }
321.         /* Case 2: Only right child */
322.         else if (!(*root)->left) {
323.             TreeNode* tmp = *root;
324.             *root = (*root)->right;  // promote right child
325.             free(tmp);
326.         }
```

```
327.         /* Case 2: Only left child */
328.         else if (!(*root)->right) {
329.             TreeNode* tmp = *root;
330.             *root = (*root)->left;   // promote left child
331.             free(tmp);
332.         }
333.         /* Case 3: Two children */
334.         else {
335.             /* Find inorder successor (smallest node in right subtre
    e) */
336.             TreeNode* succ = findMin((*root)->right);
337. 
338.             /* Copy successor's data into current node */
339.             (*root)->data = succ->data;
340. 
341.             /* Delete the successor node in right subtree */
342.             deleteNode(&((*root)->right), succ->data.word);
343.         }
344.     }
345. }
346. 
347. /* ============================================================
348.  * Function: freeTree
349.  * ------------------------------------------------------------
350.  * Purpose : Free all nodes in the BST to avoid memory leaks.
351.  * Inputs  : root - BST root
352.  * Behavior:
353.  *   - Uses postorder traversal:
354.  *       1. free left subtree
355.  *       2. free right subtree
356.  *       3. free current node
```

```
 * ============================================================*/
void freeTree(TreeNode* root) {
    if (!root) return;

    /* First free children */
    freeTree(root->left);
    freeTree(root->right);

    /* Then free current node */
    free(root);
}
```

4.3 dict_loader.h

```
#ifndef DICT_LOADER_H
#define DICT_LOADER_H

#include "bst.h"   /* We need Entry / TreeNode definitions */

/* ============================================================
 * Function: loadDictionary
 * ------------------------------------------------------------
 * Purpose : Read a dictionary text file and insert all entries into the BST.
 *
 * File format (one entry per line):
 *     word<TAB or spaces>meaning
 *
 * Example:
 *     accord  [n] 一致、符合
 *     Bible   [n]《聖經》
 *
```

```
 * Inputs  : filename - C string of the text file name to open
 *           root     - address of pointer to BST root
 *
 * Returns : 1  if file is opened and loaded successfully
 *           0  if file cannot be opened or error occurs
 *
 * Behavior:
 *   - Each non-empty line is split into "word" and "meaning".
 *   - A new Entry is created and inserted into the BST using insertNode().
 * ============================================================*/
int loadDictionary(const char* filename, TreeNode** root);

#endif
```

4.4 dict_loader.c

```
#include "dict_loader.h"
#include <stdio.h>    /* for FILE, fopen, fgets, etc. */
#include <string.h>   /* for strlen, strncpy, etc. */
#include <ctype.h>    /* for isspace */

/* ============================================
 * Helper Function: rtrim
 * ------------------------------------------------------------------------
 * Purpose : Remove trailing newline and whitespace characters from a string.
 * Inputs  : s - C string to modify in-place
 * Behavior:
 *   - Starting from the end of the string, remove:
 *       '\n', '\r', and any isspace() characters.
```

```
 *   - Useful for cleaning lines read by fgets().
 * ===============================================*/
static void rtrim(char* s) {
    int len;

    if (!s) return;

    len = (int)strlen(s);

    /* Move backward while last char is newline, carriage return or whitespace */
    while (len > 0 &&
           (s[len - 1] == '\n' ||
            s[len - 1] == '\r' ||
            isspace((unsigned char)s[len - 1]))) {
        s[len - 1] = '\0';  /* Replace with string terminator */
        --len;
    }
}

/* ================================================
 * Function: loadDictionary
 * ----------------------------------------------------------------------
 * Purpose : Load dictionary data from a text file and insert into BST.
 *
 * Inputs  : filename - name of dictionary file (e.g., "dictionary_data_1.txt")
 *           root     - address of BST root pointer
 * Returns : 1 on success, 0 on failure
 *
```

```
42. * Steps:
43. *   1. Open the file for reading.
44. *   2. For each line:
45. *       - Trim trailing whitespace.
46. *       - Skip empty lines.
47. *       - Split into "word" and "meaning" based on whitespace after
    word.
48. *       - Create Entry and call insertNode() to put it into BST.
49. * ============================================*/
50. int loadDictionary(const char* filename, TreeNode** root) {
51.     FILE* fp;
52.     char line[1024];          /* buffer for each line in the file */
53. 
54.     if (!filename || !root) {
55.         return 0;             /* invalid arguments */
56.     }
57. 
58.     /* ---------------------------------------------------------------
    -------
59.      * Open the file.
60.      * On MSVC, fopen_s is safer; on other compilers, use fopen.
61.      * ---------------------------------------------------------------
    ---- */
62. #ifdef _MSC_VER
63.     if (fopen_s(&fp, filename, "r") != 0) {
64.         fp = NULL;
65.     }
66. #else
67.     fp = fopen(filename, "r");
68. #endif
69. 
```

```
70.    if (!fp) {
71.        printf("Failed to open dictionary file: %s\n", filename);
72.        return 0;
73.    }
74.
75.    /* ------------------------------------------------------------
   -------
76.     * Read the file line by line.
77.     * Each line is expected to contain:  word  meaning...
78.     * ------------------------------------------------------------
   ---- */
79.    while (fgets(line, sizeof(line), fp)) {
80.        char* p;
81.        char word[MAX_WORD_LEN];
82.        char meaning[MAX_MEANING_LEN];
83.        int i;
84.
85.        /* Remove trailing newline / spaces */
86.        rtrim(line);
87.
88.        /* Skip empty line */
89.        if (line[0] == '\0') {
90.            continue;
91.        }
92.
93.        /* p will walk through the line */
94.        p = line;
95.
96.        /* Skip any leading spaces */
97.        while (*p && isspace((unsigned char)*p)) {
98.            p++;
```

```
99.         }
100.
101.         /* If after skipping spaces the line is empty, skip it */
102.         if (*p == '\0') {
103.             continue;
104.         }
105.
106.         /* ----------------------------
107.          * Extract the word part
108.          * ----------------------------
109.          * Read characters until we hit whitespace or end of line.
110.          */
111.         i = 0;
112.         while (*p && !isspace((unsigned char)*p) && i < (MAX_WORD_LE
    N - 1)) {
113.             word[i++] = *p++;
114.         }
115.         word[i] = '\0';  /* null-terminate the word */
116.
117.         /* If no word was read, skip this line */
118.         if (word[0] == '\0') {
119.             continue;
120.         }
121.
122.         /* ----------------------------
123.          * Skip spaces between word and meaning
124.          * ---------------------------- */
125.         while (*p && isspace((unsigned char)*p)) {
126.             p++;
127.         }
128.
```

```
129.         /* The rest of the line is the meaning */
130.         strncpy(meaning, p, MAX_MEANING_LEN - 1);
131.         meaning[MAX_MEANING_LEN - 1] = '\0';  /* ensure null-termina
   tion */
132.
133.         /* ----------------------------
134.          * Build Entry structure
135.          * --------------------------- */
136.         {
137.             Entry e;
138.
139.             /* Copy word into e.word */
140.             strncpy(e.word, word, MAX_WORD_LEN - 1);
141.             e.word[MAX_WORD_LEN - 1] = '\0';
142.
143.             /* Copy meaning into e.meaning */
144.             strncpy(e.meaning, meaning, MAX_MEANING_LEN - 1);
145.             e.meaning[MAX_MEANING_LEN - 1] = '\0';
146.
147.             /* Insert into BST */
148.             insertNode(root, &e);
149.         }
150.     }
151.
152.     fclose(fp);
153.     return 1;
154. }
```

4.5 main.c

```
1. #include <stdio.h>      /* for printf, scanf, fgets */
2. #include <string.h>     /* for strlen, strcspn */
```

```c
#include "bst.h"        /* for TreeNode, Entry, BST functions */
#include "dict_loader.h"/* for loadDictionary */

/* ========================================================================
 * Helper Function: clearLineBuffer
 * ------------------------------------------------------------------------
 * Purpose : Clear remaining characters in the input buffer until '\n' or EOF.
 *           This is useful after using scanf(), to remove leftover newline.
 * Inputs  : none
 * Returns : none
 * ======================================================================== */
static void clearLineBuffer(void) {
    int c;
    /* Read and discard characters until newline or EOF */
    while ((c = getchar()) != '\n' && c != EOF) {
        /* do nothing (just drop the chars) */
    }
}

/* ========================================================================
 * Function: main
 * ------------------------------------------------------------------------
 * Purpose : Entry point of the Mini Search Engine program.
 *
 * Behavior:
 *   - Maintain a Binary Search Tree (BST) as the dictionary.
```

```
29. *   - Provide a text-based menu:
30. *
31. *       1. Load dictionary from file
32. *       2. Show all words (sorted)
33. *       3. Search a word (with path & comparisons)
34. *       4. Add a new word
35. *       5. Delete a word
36. *       6. Exit
37. *
38. *   - Each menu option calls the appropriate BST or dictionary loader
    function.
39. * ================================================================
    ======= */
40. int main(void) {
41.     TreeNode* root = NULL;  /* BST root pointer, initially empty */
42.     int choice;             /* user menu choice */
43.
44.     while (1) {
45.         /* ------------------------------------------------------------
    ----
46.          * Display main menu
47.          * ------------------------------------------------------------
    - */
48.         printf("\n===== Mini Search Engine =====\n");
49.         printf("1. Load dictionary\n");
50.         printf("2. Show all words\n");
51.         printf("3. Search a word\n");
52.         printf("4. Add a new word\n");
53.         printf("5. Delete a word\n");
54.         printf("6. Exit\n");
55.         printf("Enter choice: ");
56.
```

```
57.         /* Read user choice using scanf */
58.         if (scanf("%d", &choice) != 1) {
59.             /* Input not an integer → invalid input */
60.             printf("Invalid input (not a number).\n");
61.             clearLineBuffer(); /* clear invalid input */
62.             continue;          /* go back to show menu again */
63.         }
64.
65.         /* ------------------------------------------------------------
    ----
66.          * Option 1: Load dictionary from file
67.          * ------------------------------------------------------------
    - */
68.         if (choice == 1) {
69.             char filename[256];
70.
71.             printf("Enter dictionary file (e.g., dictionary_data_1.tx
    t): ");
72.             /* %255s ensures we do not overflow filename buffer */
73.             if (scanf("%255s", filename) != 1) {
74.                 printf("Invalid filename.\n");
75.                 clearLineBuffer();
76.                 continue;
77.             }
78.
79.             /* Call loader to read file and insert entries into BST */
80.             if (loadDictionary(filename, &root)) {
81.                 printf("Dictionary loaded successfully.\n");
82.             } else {
83.                 printf("Failed to load dictionary.\n");
84.             }
```

```
85.         }
86.         /* ------------------------------------------------------------
    ----
87.          * Option 2: Show all words (in sorted order)
88.          * ------------------------------------------------------------
    - */
89.         else if (choice == 2) {
90.             if (!root) {
91.                 /* Tree is still empty (no dictionary loaded yet) */
92.                 printf("Please load dictionary first.\n");
93.             } else {
94.                 /* inorder traversal prints words in sorted order */
95.                 inorder(root);
96.             }
97.         }
98.         /* ------------------------------------------------------------
    ----
99.          * Option 3: Search for a word
100.           * ------------------------------------------------------------
    -- */
101.         else if (choice == 3) {
102.             if (!root) {
103.                 printf("Please load dictionary first.\n");
104.                 continue;
105.             }
106.
107.             {
108.                 char query[MAX_WORD_LEN];
109.                 char path[MAX_PATH_LEN][MAX_WORD_LEN];
110.                 int path_len = 0;
111.                 int comparisons = 0;
112.                 TreeNode* result;
```

```
113.
114.                 printf("Enter word to search: ");
115.                 if (scanf("%127s", query) != 1) {
116.                     printf("Invalid input.\n");
117.                     clearLineBuffer();
118.                     continue;
119.                 }
120.
121.                 /* Call searchNode to search in BST and record path *
    /
122.                 result = searchNode(root, query, path, &path_len, &co
    mparisons);
123.
124.                 /* Print search path: all visited nodes in order */
125.                 printf("Search path: ");
126.                 if (path_len == 0) {
127.                     printf("(empty)");
128.                 } else {
129.                     int i;
130.                     for (i = 0; i < path_len; ++i) {
131.                         if (i > 0) {
132.                             printf(" -> ");   /* arrow between nodes *
    /
133.                         }
134.                         printf("%s", path[i]);
135.                     }
136.                 }
137.                 printf("\n");
138.
139.                 /* Print comparison count */
140.                 printf("Comparisons: %d\n", comparisons);
```

```
141.
142.                 /* If found, show word and meaning; otherwise suggest
    closest */
143.                 if (result) {
144.                     printf("Found: %s : %s\n",
145.                             result->data.word,
146.                             result->data.meaning);
147.                 } else {
148.                     TreeNode* pred = NULL;
149.                     TreeNode* succ = NULL;
150.
151.                     printf("Not found.\n");
152.
153.                     /* Find closest words (predecessor & successor) *
    /
154.                     findClosest(root, query, &pred, &succ);
155.
156.                     printf("Closest suggestions:\n");
157.                     if (pred) {
158.                         printf("  Previous (smaller): %s : %s\n",
159.                                 pred->data.word, pred->data.meaning);
160.                     }
161.                     if (succ) {
162.                         printf("  Next (larger):     %s : %s\n",
163.                                 succ->data.word, succ->data.meaning);
164.                     }
165.                     if (!pred && !succ) {
166.                         printf("  (No close words found in dictionary)
    \n");
167.                     }
168.                 }
```

```
169.             }
170.         }
171.         /* ------------------------------------------------------------
     -----
172.          * Option 4: Add a new word to dictionary
173.          * ------------------------------------------------------------
     -- */
174.         else if (choice == 4) {
175.             Entry e;
176.
177.             /* Clear leftover newline from previous scanf */
178.             clearLineBuffer();
179.
180.             /* Read new word (full line) */
181.             printf("Enter new word: ");
182.             if (!fgets(e.word, sizeof(e.word), stdin)) {
183.                 printf("Input error.\n");
184.                 continue;
185.             }
186.             /* Remove trailing newline character */
187.             e.word[strcspn(e.word, "\r\n")] = '\0';
188.
189.             /* Read meaning (can contain spaces) */
190.             printf("Enter meaning: ");
191.             if (!fgets(e.meaning, sizeof(e.meaning), stdin)) {
192.                 printf("Input error.\n");
193.                 continue;
194.             }
195.             /* Remove trailing newline character */
196.             e.meaning[strcspn(e.meaning, "\r\n")] = '\0';
197.
```

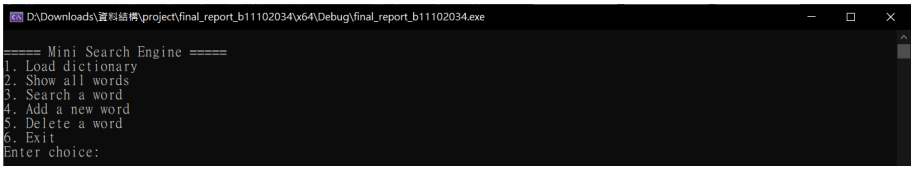
```
198.             if (e.word[0] == '\0') {
199.                 /* Empty word is not allowed */
200.                 printf("Word cannot be empty.\n");
201.             } else {
202.                 /* Insert or update this entry into BST */
203.                 insertNode(&root, &e);
204.                 printf("Word inserted/updated successfully.\n");
205.             }
206.         }
207.         /* ------------------------------------------------------------
    -----
208.          * Option 5: Delete a word from dictionary
209.          * ------------------------------------------------------------
    -- */
210.         else if (choice == 5) {
211.             if (!root) {
212.                 printf("Dictionary is empty. Please load or add words
     first.\n");
213.                 continue;
214.             }
215. 
216.             {
217.                 char word[MAX_WORD_LEN];
218.                 char dummyPath[MAX_PATH_LEN][MAX_WORD_LEN];
219.                 int dummyLen = 0;
220.                 int dummyCmp = 0;
221.                 TreeNode* found;
222. 
223.                 printf("Enter word to delete: ");
224.                 if (scanf("%127s", word) != 1) {
225.                     printf("Invalid input.\n");
```

```
226.                     clearLineBuffer();
227.                     continue;
228.                 }
229.
230.                 /* First, check if the word actually exists in BST */
231.                 found = searchNode(root, word, dummyPath, &dummyLen,
    &dummyCmp);
232.
233.                 if (!found) {
234.                     printf("Word not found. Nothing deleted.\n");
235.                 } else {
236.                     deleteNode(&root, word);
237.                     printf("Word deleted successfully.\n");
238.                 }
239.             }
240.         }
241.         /* ------------------------------------------------------------
    -----
242.          * Option 6: Exit program
243.          * ------------------------------------------------------------
    -- */
244.         else if (choice == 6) {
245.             printf("Exiting program...\n");
246.             break;   /* leave the while loop */
247.         }
248.         /* ------------------------------------------------------------
    -----
249.          * Invalid menu option
250.          * ------------------------------------------------------------
    -- */
251.         else {
252.             printf("Invalid choice. Please enter 1-6.\n");
```

```
253.         }
254.     }
255.
256.     /* Before exiting, free the entire BST to avoid memory leaks */
257.     freeTree(root);
258.     return 0;
259. }
```

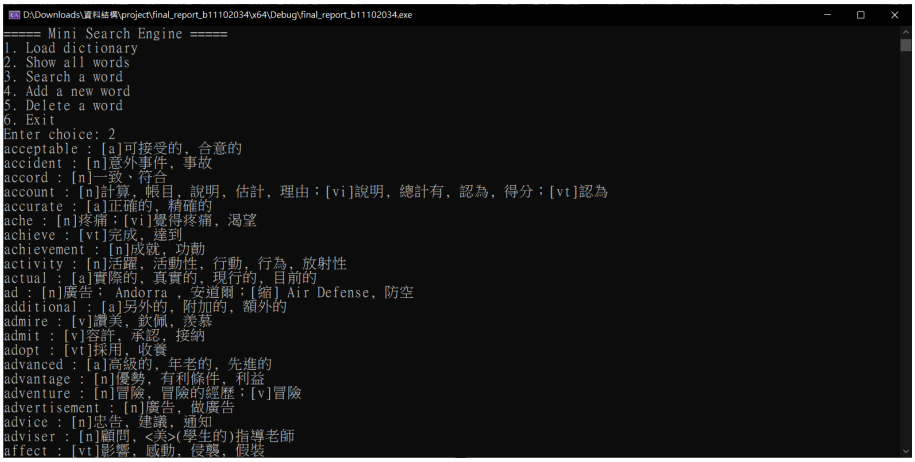
5. Results

5.1 Program main menu



5.2 Load dictionary

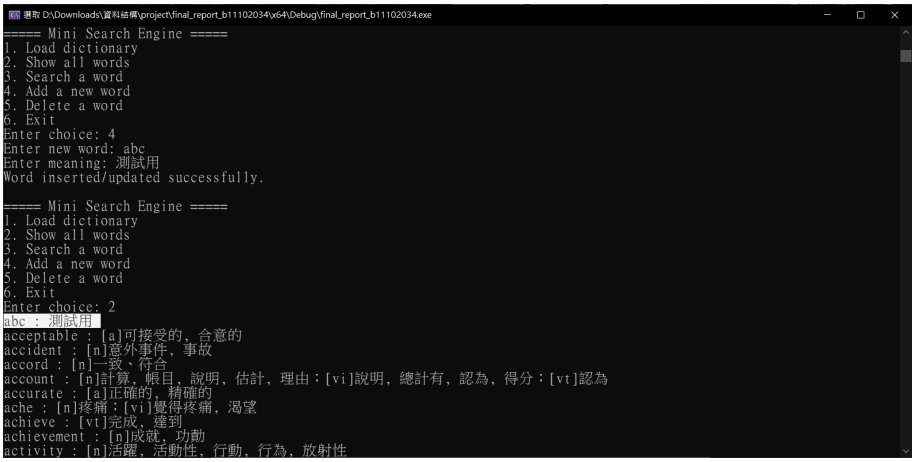
5.3 Show all words



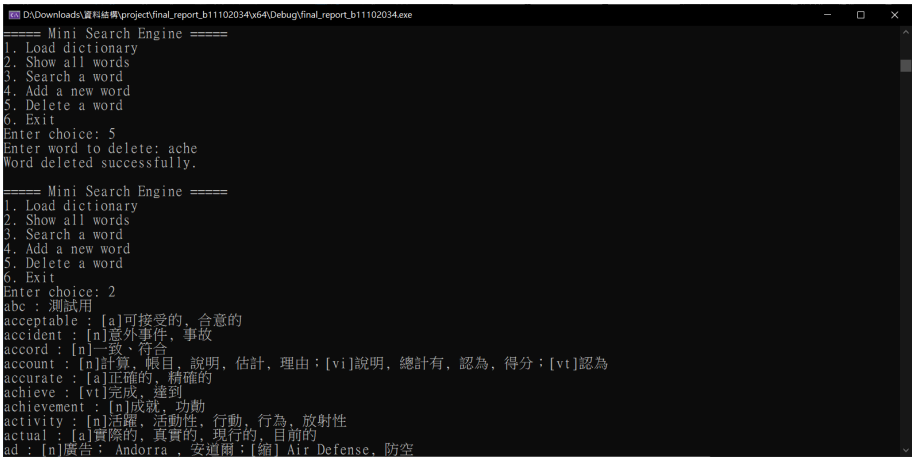
5.4 Search result (found)

5.5 Search result (not found) + closest suggestions

5.6 Insert a new word



5.7 Delete a word



6. Discussion

This project demonstrates how an efficient data structure improves search performance.

Compared with linear search, BST operations require only O(log n) comparisons on average, making it suitable for dictionary-like applications.

The project also highlights:

- Importance of case-insensitive storage
- Limitations of BST (unbalanced in worst-case)

- Need for user-friendly interface
- Correctness challenges of BST delete implementation

7. Conclusion

This Mini Search Engine successfully integrates key Data Structures concepts, including:

- Binary Search Trees
- String handling
- File parsing
- Algorithmic thinking

The system performs all required operations:

- Load
- Search
- Insert
- Delete
- Sorted listing
- Closest word suggestion

Through this project, I gained deeper understanding of dynamic data structures, recursion, memory management, and text processing in C.