資料結構期末

程式執行流程

1. 初始化階段

1. 程式啟動時創建 Trie 樹的根節點 (`create_node()`)
2. 嘗試打開並讀取 `article.txt` 文件 (`load_article()`)
3. 為文本內容分配記憶體 (`malloc()` in `load_article()`)
4. 創建文本副本用於分詞處理 (`strdup()` in `main()`)

2. 文本處理階段

1. 使用分隔符將文本分割成單字 (`insert_words()` 使用 `strtok()`)
2. 對每個單字進行處理 (`insert()`) :
    - 過濾無效字符 (使用 `isalpha()` , `isdigit()`)
    - 轉換大寫為小寫 (`tolower()`)
    - 保留數字和撇號

3. Trie 樹建構階段

1. 對每個處理後的單字 (`insert()`) :
    - 從根節點開始遍歷
    - 為每個字符創建或訪問對應節點 (`create_node()`)
    - 到達單字末尾時增加計數器 (`count++`)

4. 統計收集階段

1. 深度優先遍歷 Trie 樹 (`collect_words()`)
2. 收集所有完整單字及其頻率 (存入 `WordInfo` 結構)
3. 計算相關統計數據：
    - 總單字數 (`word_count`)
    - 總節點數 (`word_count`)
    - 樹的高度 (`max_depth`)

5. 結果輸出階段 (`main()`)

1. 按順序輸出每個單字及其頻率 (`printf()`)
2. 輸出總體統計信息 (`printf()`)

6. 清理階段 (`main()`)

1. 釋放文章內容的記憶體 (`free(article)`)
2. 釋放文章副本的記憶體 (`free(article_copy)`)
3. 遞歸釋放 Trie 樹的所有節點 (`free_trie()`)

技術實現

數據結構

1. **Trie 節點**

```
typedef struct Node {
    struct Node *children[37];  // 10個數字 + 26個字母 + 1個撇號
    int count;                  // 計數器
} Node;
```

2. **單字信息**

```
typedef struct {
    char word[100];
    int count;
} WordInfo;
```

字符索引對應

程式使用 37 個槽位的陣列來存儲每個節點：

- 數字 (0-9)：索引 0-9
- 字母 (a-z)：索引 10-35
- 撇號 (')：索引 36

核心功能

1. **文件操作**

- `load_article()` ：讀取輸入文件
- `insert_words()` ：將文本分割成單字

2. **Trie 樹操作**
   - `create_node()` ：創建新節點
   - `insert()` ：插入單字
   - `collect_words()` ：收集單字統計信息
   - `free_trie()` ：釋放記憶體

核心算法說明

collect_words 深度優先搜索算法

1. **函數參數**

```
void collect_words(
    Node *node,           // 當前節點
    char *current_word,   // 目前累積的單字
    int depth,            // 當前深度
    WordInfo *words,      // 儲存結果的陣列
    int *word_count,      // 已收集的單字數量
    int *max_depth        // 記錄最大深度
)
```

2. **執行步驟**
   i. **基本判斷**
      - 檢查當前節點是否為單字結尾 (`node->count > 0`)
      - 如果是，將累積的單字和頻率存入 `words` 陣列
      - 更新單字計數 (`word_count`) 和最大深度 (`max_depth`)

   ii. **遍歷子節點**
      - 依序檢查所有可能的字符 (0-9, a-z, ')
      - 對每個存在的子節點進行處理

   iii. **字符轉換**
      - 數字 (0-9)： `'0' + i`
      - 字母 (a-z)： `'a' + (i - 10)`
      - 撇號： `'\''`

   iv. **遞迴處理**
      - 將當前字符加入累積的單字
      - 遞迴訪問子節點
      - 返回時自動回溯 (不需要顯式移除字符)

3. **時間複雜度**

- 最壞情況：O(N * L)
    - N: 不同單字的數量
    - L: 最長單字的長度

4. **空間複雜度**
    - 遞迴調用棧：O(H)，H 為樹高
    - 儲存結果：O(N * L)
5. **範例執行流程**

    初始狀態：`current_word = ""`

    遞迴過程示例（以 `"cat"` 為例）：
    第一層：`current_word = "c"`
    第二層：`current_word = "ca"`
    第三層：`current_word = "cat"`
    → 發現完整單字，儲存 `"cat"` 及其頻率

    回溯並繼續搜索其他分支...

insert 函數算法

1. **函數參數**

    ```
    void insert(
        Node *root,          // Trie 樹的根節點
        const char *word     // 要插入的單字
    )
    ```

2. **執行步驟**
    i. **字符處理**
        - 遍歷輸入單字的每個字符
        - 對每個字符進行分類處理：

            ```
            if (isdigit(c)) {
                index = c - '0';           // 數字映射到 0-9
            }
            else if (isalpha(c)) {
                index = 10 + tolower(c) - 'a';  // 字母映射到 10-35
            }
            else if (c == '\'') {
                index = APOSTROPHE_INDEX;  // 撇號映射到 36
            }
            ```

ii. **節點創建與遍歷**

- 檢查當前字符的子節點是否存在
- 如果不存在，創建新節點：

```
if (current->children[index] == NULL) {
    current->children[index] = create_node();
}
```

- 移動到子節點繼續處理

iii. **計數更新**

- 到達單字末尾時，增加該節點的計數器
- 表示這是一個完整單字的結尾

3. **特點說明**
   - 自動忽略無效字符
   - 大小寫不敏感
   - 支持數字和撇號
4. **時間複雜度**
   - O(L)，其中 L 為單字長度
   - 每個字符只需處理一次
5. **空間複雜度**
   - 最壞情況：O(L)
   - 需要為新的字符路徑創建節點
6. **範例執行流程**

插入 "Cat's" 的過程：

1. 處理 'C'：
   - 轉小寫 'c'
   - 計算索引：10 + (c - 'a') = 12
   - 創建或使用索引 12 的子節點

2. 處理 'a'：
   - 計算索引：10 + (a - 'a') = 10
   - 創建或使用索引 10 的子節點

3. 處理 't'：
   - 計算索引：10 + (t - 'a') = 29
   - 創建或使用索引 29 的子節點

4. 處理 '''：
   - 使用索引 36
   - 創建或使用索引 36 的子節點

5. 處理 's'：
   - 計算索引：10 + (s - 'a') = 28
   - 創建或使用索引 28 的子節點
   - 增加該節點的計數器