SIDE PROJECT

應試者:江恩榮

附上 (1)Source code (2)說明 (3)自己的 testing results

注重每個解法的執行效率以及程式可維護性(「可擴展性」和「可靠性」)

是否獨立完成:是

是否有用 AI 工具:有

題目二 ：Tree walk

配置:

C++版本:17/20/23 編譯器: g++ 13.1.0

假設輸入 “A,B,C,,D,E,F,,,,,,,G” 能建立下方的 Binary tree，保證 Tree 的深度不會超過 30 層，

前進方向只有 “上” 、” 左” 、” 右” 。

上：代表向上走到 parent

左：代表走到 left child

右：代表走到 right child

輸入一個 “起點” 跟 “終點” ，例如 B 跟 F，能找出從 B 到達 F 的路徑。以下圖為例，找出 “上右右”

最快速度 + 最少 memory usage 的解法

一、 題目理解

本題目想得知兩節點間的深度差異及廣度差異，答案須滿足以下要求：

1. 只能向上、右、左移動
2. 要處理空樹、單節點、節點不存在、確保例外狀況有回應
3. 最快速度 + 最少 memory usage

二、 解題思路與程式碼說明

1. 樹節點定義

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_map>
```

```
using namespace std;


struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode* parent;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr),
parent(nullptr) {}
};
```

說明：

宣告節點結構且初始化參數，雖本算法未保留 parent 用於擴展

2. 程式碼說明

非遞迴路徑查找 (findDirectionPathNonRecursive):

- 思路：使用深度優先搜索 (DFS) 搭配容器 stack 來追蹤路徑方向。
- Stack 儲存：當前節點 + 從根節點到該節點的路徑方向 (0=上,1=左, 2=右)。
- 路徑記錄：每次向下探索時，複製當前路徑並追加新方向。

```
// 非遞迴查找從根到目標節點的路徑方向
bool findDirectionPathNonRecursive(TreeNode* root, int target,
vector<int>& path) {
    if (!root) return false;

    stack<pair<TreeNode*, vector<int>>> s; // stack 容器堆疊路線

    s.push({root, {}});

    while (!s.empty()) {
        auto [node, currentPath] = s.top();
        s.pop();
```

```
        if (node->val == target) {
            path = currentPath;
            return true;
        }

        if (node->right) {
            vector<int> newPath = currentPath;
            newPath.push_back(2); // 向右
            s.push({node->right, newPath});
        }

        if (node->left) {
            vector<int> newPath = currentPath;
            newPath.push_back(1); // 向左
            s.push({node->left, newPath});
        }
    }

    return false;
}
```

實現代碼說明：

1. 避免遞迴風險：迭代法不像遞迴可能遭遇 Stack memory 溢出問題，適合大樹，減少例外情況。
2. 明確方向記錄：直接儲存「左右」步驟，省去後續轉換。
3. 空間效率：僅需 O(h) 空間（h 為樹高），且路徑共享部分可被複用。

路徑合併和錯誤處理((findPathBetweenNodesWithDirections):

- 思路 ：分層驗證 + 明確錯誤訊息。先檢查空樹，再驗證節點是否存在。
- LCA (最低共同祖先) 路徑合併 ：分別找到兩節點到根的路徑（方向序列）。
- 錯誤訊息 ：指出缺失的節點（如 "node 5 miss"）。

```
// 查找兩個節點之間的路徑（帶詳細錯誤信息）
pair<vector<int>, string> findPathBetweenNodesWithDirections(TreeNode*
root, int node1, int node2) {
    vector<int> path1, path2;
```

```
    string errorMsg;

    // 檢查空樹
    if (!root) {
        return {{}, "empty no node in tree"};
    }

    // 查找路徑
    bool found1 = findDirectionPathNonRecursive(root, node1, path1);
    bool found2 = findDirectionPathNonRecursive(root, node2, path2);

    // 處理節點不存在的情況
    if (!found1 && !found2) {
        errorMsg = "node " + to_string(node1) + " and " +
to_string(node2) + " miss";
        return {{}, errorMsg};
    } else if (!found1) {
        errorMsg = "node " + to_string(node1) + " miss";
        return {{}, errorMsg};
    } else if (!found2) {
        errorMsg = "node " + to_string(node2) + " miss";
        return {{}, errorMsg};
    }

    // 尋找共同前綴長度
    int commonLength = 0;
    while (commonLength < path1.size() &&
           commonLength < path2.size() &&
           path1[commonLength] == path2[commonLength]) {
        commonLength++;
    }

    // 構建最終路徑
    vector<int> result;

    // 從 node1 到 LCA 的路徑（需要往父節點走，用 0 表示）
    for (int i = path1.size() - 1; i >= commonLength; --i) {
        result.push_back(0); // 往父節點
```

```
    }

    // 從 LCA 到 node2 的路徑（使用原來的方向）
    for (int i = commonLength; i < path2.size(); ++i) {
        result.push_back(path2[i]);
    }

    return {result, ""};
}
```

從 node1 到 LCA：反向遍歷路徑，用 0（↑）表示「回父節點」。
從 LCA 到 node2：直接追加剩餘方向。

實現程式碼說明：

1. 基於樹的性質:兩節點間路徑必通過 LCA(lowest common ancestor)，退化為鏈表的樹例外。
2. 時間效率：O(h) 時間完成路徑查找與合成（h 為樹高）。
3. 清晰的方向語義：用 0/1/2 直觀表示「上/左/右」，易於理解與調整。

3. 輔助函式

測試樹的建構 (buildTreeWithParent):

本案例未特別要求樹的類型因此假設基礎測試為 Full binary tree，並以遞迴建立平衡樹，同時設置 `parent` 指針。選擇中點作為根，確保樹平衡。遞迴時傳遞父節點，建立雙向連結。測試案例能涵蓋典型與邊界情況。雖然本解法未直接使用 `parent`，但保留彈性供擴展。

```
// 創建樹並設置父節點指針
TreeNode* buildTreeWithParent(const vector<int>& values, int start, int
end, TreeNode* parent = nullptr) {
    if (start > end) return nullptr;

    int mid = start + (end - start) / 2;
    TreeNode* node = new TreeNode(values[mid]);
    node->parent = parent;
```

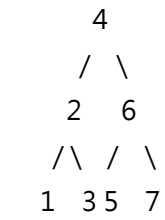
```
    node->left = buildTreeWithParent(values, start, mid - 1, node);
    node->right = buildTreeWithParent(values, mid + 1, end, node);

    return node;
}
```

4. 測試案例說明

測試案例

1. 完美平衡樹測試



測試編號	節點 A →節點 B	預期路徑	測試目的	測試結果
test1	1 → 7	↑ ↑ → →	最遠距離測試：從最左下角到最右下角	通過
test2	3 → 5	↑ ↑ → ←	跨子樹測試：需回溯到根節點再下行	通過
test3	4 → 4	(空路徑)	相同節點測試	通過
test4	1 → 9	錯誤提示	無效節點測試	通過
test_sside	1 → 3	↑ →	同側子樹測試：需回溯再下行	通過

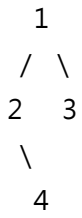
2. 單節點樹測試

測試編號	節點 A →節點 B	預期路徑	測試目的	測試結果
test5	1 → 1	(空路徑)	根為節點測試	通過
test6	1 → 2	錯誤提示	單節點樹的無效節點測試	通過

3. 空樹測試

測試編號	節點 A →節點 B	預期路徑	測試目的	測試結果
test7	1 → 2	錯誤提示	空樹的基本錯誤處理	通過
test6	1 → 2	錯誤提示	單節點樹的無效節點測試	通過

4. 其他樹狀結構測試



測試編號	節點 A →節點 B	預期路徑	測試目的	測試結果
Test8	4 → 3	↑ ↑ →	非對稱結構測試	通過
test9	2 → 4	→	單邊下行測試	通過

三、總結

透過 LCA (最低共同祖先)和分治法的概念將複雜問題拆解為可處理的子問題。測試案例涵蓋空樹、單節點、無效節點等情境，使用 stack 模擬遞迴，避免了潛在的堆疊溢出風險。在搜尋速度及記憶體的使用上應該還能再改進，例如路徑的存處，也許可以使用轉發或是更有效的利用容器特性。

參考資料

1. tree - 演算法筆記
2. binary tree - 演算法筆記
3. 二叉树的中序遍历

題目三 ：Big File Reverse

大型文字檔案倒轉處理與效能比較

配置:

Intel(R) Core(TM) i7-10700 CPU
RAM 16.0 GB
Python 3.9.7

一、題目理解：

- **檔案分割與讀寫操作**：由於檔案過大，無法一次載入至記憶體，需進行分段處理，一般的讀寫操作不可用。
- **文字內容倒轉**：將整體檔案內容倒轉，例如原始為 abcde，倒轉後為 edcba。
- **倒轉邏輯正確性**：每個區塊需先倒轉，再依「原始檔案結尾至開頭」的順序組合成新檔案。
- **效能最佳化**：透過不同方法比較效能（執行時間、記憶體使用、可維護性等）。

二、 解題策略與技術選擇

1. 分割策略採用固定大小的區塊（如每 10MB 一塊）進行讀取與處理。為避免文字斷裂，需處理編碼邊界（特別是 UTF-8 多位元字元）。
2. 倒轉流程設計 Step 1：從檔案尾端開始，分塊讀取內容。Step 2：將每個區塊中的文字倒轉（如 chunk[::-1]）。Step 3：將倒轉後的區塊，依倒序順序寫入新檔案。Step 4：重複直到處理整個檔案。
3. 效能實作比較方法 A：多執行緒讀取 + 排隊寫入方法 B：加上使用 mmap 記憶體映射加速存取

三、比較方案評估

方法	優點	缺點	適用場景
多執行緒分區處理	平行化加速處理	同步與順序控制較複雜	多核環境
mmap 操作	記憶體效率高、處理大型檔案快	跨平台需測試	檔案非常大、效能敏感的系統

四、注意事項與挑戰

1. 編碼問題處理：避免將多位元字元切割，導致無法解析或亂碼。
2. 區塊邊界調整：必要時往前或後找換行或字元邊界來分塊。磁碟 I/O 瓶頸：需控制 buffer 大小以降低讀寫耗時。
3. 錯誤處理與記憶體控制：確保程式長時間運行不會漏出記憶體或產生錯誤

五、程式碼說明(python)

1. 生成測試文件(file_init.py)

可藉由 gb 參數調整生成文件大小，重複"This is a sample text line for generating a large file. " 字串，將其作為生成文件內容，字串長度為 55 個字節* 20000 其記憶體量約等於 1MB 方便填入計算。

```
if __name__ == "__main__":
# 測試文件大小（10GB）

    gb="0.01" # 調整生成文件大小係數
    target_size = float(gb) * 1024 * 1024 * 1024  # 10GB in bytes
    output_file = "D:/trading_code/py_trade/1_100gb_text_file.txt"

    # 寫入重複的數據
    chunk_size = 1024 * 1024  # 1MB per chunk
    chunk_data = "This is a sample text line for generating a large
file. " * 20000  # ~1MB

    with open(output_file, "w") as f:
```

```
        written = 0
        while written < target_size:
            f.write(chunk_data)
            written += len(chunk_data)
            print(f"Written: {written / (1024*1024*1024):.2f} GB",
end="\r")
```

2. 輔助函式

detect_file_encoding_from_mmap( )識別編碼為 ANSI Big5 or UTF-8:

如果是 UTF-8：一個字可能 1~4 byte，要反轉字元（而不是 byte），如果是 Big5（雙位元）：直接反轉 bytes 就行。

```
def detect_file_encoding_from_mmap(mm):
    header = mm[:3]
    if header == b'\xef\xbb\xbf':
        return 'utf-8'
    return 'big5'
```

reverse_chunk( )反轉文字檔案的內容

```
def reverse_chunk(chunk, encoding):
    if encoding == 'utf-8':
        reversed_bytes = bytearray()
        i = len(chunk) - 1
        while i >= 0:
            if (chunk[i] & 0x80) == 0:
                reversed_bytes.append(chunk[i])
                i -= 1
            else:
                start = i
                while start >= 0 and (chunk[start] & 0xC0) == 0x80:
                    start -= 1
                reversed_bytes.extend(chunk[start:i+1])
                i = start - 1
        return bytes(reversed_bytes)
    else:
        return chunk[::-1]
```

reverse_chunk( )反轉分割後的文字塊並進行寫入

分為 thread 版本可使用 mmap 共享映射內容，以及每個 process 自行開啟檔案，讀取區塊。

```
def reverse_chunk_range(range_info, mm, encoding):
    #thread 版本
    start, end = range_info
    chunk = mm[start:end]
    reversed_data = reverse_chunk(chunk, encoding)
    temp_file = tempfile.NamedTemporaryFile(delete=False)
    temp_file.write(reversed_data)
    temp_file.close()
    return temp_file.name


def reverse_chunk_range_from_file(range_info, mmap_file, encoding,
chunk_size): #process 版本
    start, end = range_info
    with open(mmap_file, 'rb') as f:
        f.seek(start)
        chunk = f.read(end - start)
    reversed_data = reverse_chunk(chunk, encoding)
    temp_file = tempfile.NamedTemporaryFile(delete=False)
    temp_file.write(reversed_data)
    temp_file.close()
    return temp_file.name
```

3. 主要函式

reverse_large_file( )多執行緒：

直接用 mmap 讀 chunk → 反轉 → 寫入臨時檔

```
def reverse_large_file(input_file, output_file, buffer_size=1024*1024,
num_threads=4):
    start_time = time.time()
    with open(input_file, 'rb') as f:
        with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm:
            encoding = detect_file_encoding_from_mmap(mm)
            total_size = len(mm)
            chunk_ranges = [(i, min(i + buffer_size, total_size))
                            for i in range(0, total_size, buffer_size)]

            with ThreadPoolExecutor(max_workers=num_threads) as
executor:
                reverse_func = partial(reverse_chunk_range, mm=mm,
encoding=encoding)
                temp_files = list(executor.map(reverse_func,
chunk_ranges))

            with open(output_file, 'wb') as out_f:
                for temp_file in reversed(temp_files):
                    with open(temp_file, 'rb') as in_f:
                        while True:
                            data = in_f.read(buffer_size)
                            if not data:
                                break
                            out_f.write(data)
                    os.remove(temp_file)
    elapsed_time = time.time() - start_time
    return elapsed_time
```

reverse_large_file_mp( )多進程：

每個進程重新開檔、seek 到對的位置 → 處理 → 寫入臨時檔

```
def reverse_large_file_mp(input_file, output_file,
buffer_size=1024*1024, num_processes=4):
    start_time = time.time()
    with open(input_file, 'rb') as f:
            ...與 thread 版本相同
            with multiprocessing.Pool(processes=num_processes) as pool:
                reverse_func = partial(reverse_chunk_range_from_file,
                                       mmap_file=input_file,
encoding=encoding, chunk_size=buffer_size)
                temp_files = pool.map(reverse_func, chunk_ranges)

            with open(output_file, 'wb') as out_f:
              ...與 thread 版本相同
    elapsed_time = time.time() - start_time
    return elapsed_time
```

4. 結果測試

試多組 buffer size(1/2/4/8MB)/ 執行緒數（1/2/4/8），每組跑一次 threading + multiprocessing，印出時間(second)比較表格。

```
def benchmark_all(input_file, base_output_file):
    threads_or_procs = [1, 2, 4, 8]
    buffer_sizes = [1*1024*1024,2*1024*1024, 4*1024*1024, 8*1024*1024]

  ... ...輸出執行秒數
```

首先以 1GB 的檔案為操作對象，使用 mmap 與不使用的比較 ，固定緩存大小為 1MB 的比較

公式:性能提升百分比=(多線程執行時間−多進程執行時間)×100%

Threads	BufferSize(MB)	Time(s)	m_Time(s)	性能提升
1	1	15.91	12.11	23.88%
2	1	9.71	9.93	-2.27%
4	1	9.98	8.60	13.83%
8	1	10.91	10.78	1.19%

整體提升約 10.94%（按總時間計算），有些許提升，但並無明顯變化。為了比較改變佔存大小是否會有更顯著的變化，將改變佔存(1/2/4/8MB)及執行緒/進程數（1/2/4/8）看看有無明顯提升。

Threads	BufferSize(MB)	Time(s)	m_Time(s)	性能提升
1	1	16.02	14.42	**9.99**
1	2	8.26	16.39	-98.43
1	4	8.55	12.27	-43.51
1	8	13.07	8.07	38.26
2	1	8.82	13.09	-48.41
2	2	9.38	7.40	21.11
2	4	7.34	9.25	-26.02
2	8	9.56	9.59	-0.31
4	1	10.35	10.35	-0.05
4	2	8.04	8.04	-7.76
4	4	9.85	9.85	-4.67
4	8	8.31	8.31	-34.54
8	1	12.13	12.13	9.04
8	2	10.32	10.32	16.47
8	4	10.01	10.01	-19.58
8	8	13.42	13.42	15.13

整體趨勢分析

提升的案例（6 組）

最大提升：**+38.26%**（行 4）

最小提升：**+9.07%**（行 13）

平均提升（僅計算提升的組）：**≈18.34%**

下降的案例（10 組）

最大下降：**-98.43%**（行 2，幾乎翻倍時間）

最小下降：**-0.31%**（行 8，幾乎持平）

- **右排時間在大多數情況下更差（10/16 組下降）**，尤其行 2、5、10 下降近 50% 或更高。

- **部分情況有提升（6/16 組）**，最佳優化達 **+38.26%**（行 4）。
- **整體趨勢偏向負面**，可能代表右排設定（多進程策略）在當前環境下效率較低。

六、總結

此次測試以 1GB 的文字檔為主，在此情況下使用 thread 搭配 mmap 在調整參數後有比較穩定的表現，在執行續數量為 2，緩衝區為 4MB 的情況下，10 秒內處理完畢，且一樣可處理>1GB 的大檔案，採用 mmap 的讀取方式，將整個檔案映射到記憶體中，不佔用大量 RAM 且減少 IO 操作。適合硬體配置較低的情況操作。

但在實際操作中應該根據檔案大小選擇多進程或多線程，檔案大小 ≥ 5~10 GB 選擇多進程可避免 mmap 映射太大，並行處理記憶體更穩定，硬體配置若是有高核心，也應該還是以多進程為主，可更好發揮其效能。

預估需要多少 memory？

公式:記憶體 ≈ 並行數 × buffer_size + 程式本體 overhead

例:num_threads=4　buffer_size=8MB

預估佔用：4 × 8MB = 32MB（+ 程式 overhead 約 20~50MB），小於 100MB，記憶體需求非常小，非常適合低 RAM 的環境

預估需要多少額外 disk space？

每個處理後的區塊都會被寫到一個暫存檔案，最終再倒序讀回組成反轉後的輸出檔案。

公式:Disk Space ≈ Input file size × 2

例:原始檔案：1GB ，最大額外磁碟需求：約 2GB（含 temp + 輸出）

認為這個程式效率如何？

不如預期，預計 10GB 的檔案想在 10 秒內完成，還有可更改處。

改進方向:

- **本次測試假設文件都以單一編碼為主，可擴充更多種字元的辨識以增加程式的泛用性。**
- **本次電腦為配置 NVDIA 顯示卡，根據過往經驗，若是使用 CUDA 應可大**

大提高其效率

- 程式語言選擇可改為 C++版本，但 C++處理文字檔案較為不便，應審慎評估。

參考資料

1. mmap 原理与应用 |yangjie2.github.io
2. python 基本文件读写 及 读取大文件而内存不溢出 4 种方式_python 读取大文件内存溢出-CSDN 博客
3. Performance of multi-process and multi-thread processing on multi-core SMT processors |
4. Python 如何優雅地限制執行緒數量