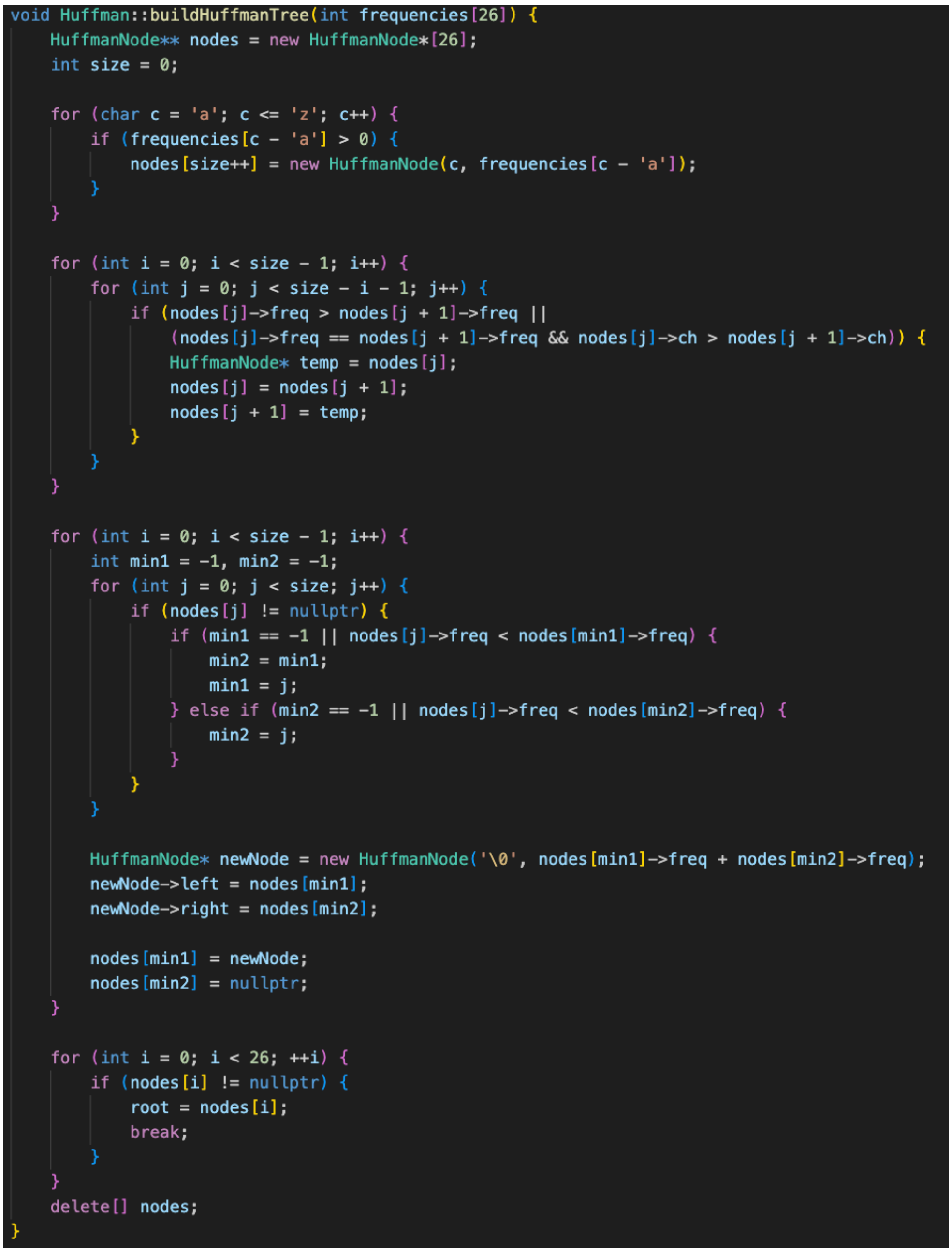
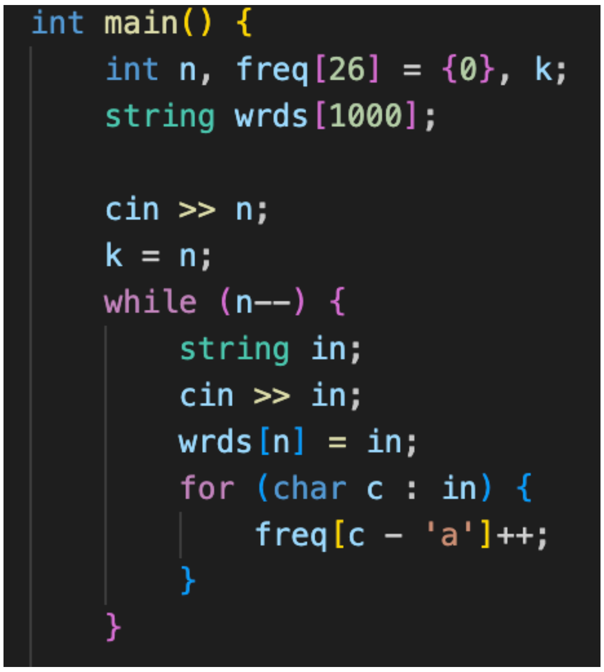
A1125531-HW5



```cpp
void Huffman::buildHuffmanTree(int frequencies[26]) {
    HuffmanNode** nodes = new HuffmanNode*[26];
    int size = 0;

    for (char c = 'a'; c <= 'z'; c++) {
        if (frequencies[c - 'a'] > 0) {
            nodes[size++] = new HuffmanNode(c, frequencies[c - 'a']);
        }
    }

    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (nodes[j]->freq > nodes[j + 1]->freq ||
                (nodes[j]->freq == nodes[j + 1]->freq && nodes[j]->ch > nodes[j + 1]->ch)) {
                HuffmanNode* temp = nodes[j];
                nodes[j] = nodes[j + 1];
                nodes[j + 1] = temp;
            }
        }
    }

    for (int i = 0; i < size - 1; i++) {
        int min1 = -1, min2 = -1;
        for (int j = 0; j < size; j++) {
            if (nodes[j] != nullptr) {
                if (min1 == -1 || nodes[j]->freq < nodes[min1]->freq) {
                    min2 = min1;
                    min1 = j;
                } else if (min2 == -1 || nodes[j]->freq < nodes[min2]->freq) {
                    min2 = j;
                }
            }
        }

        HuffmanNode* newNode = new HuffmanNode('\0', nodes[min1]->freq + nodes[min2]->freq);
        newNode->left = nodes[min1];
        newNode->right = nodes[min2];

        nodes[min1] = newNode;
        nodes[min2] = nullptr;
    }

    for (int i = 0; i < 26; ++i) {
        if (nodes[i] != nullptr) {
            root = nodes[i];
            break;
        }
    }
    delete[] nodes;
}
```

In the first for loop I first created the nodes, in the second loop I sorted by frequency and ascii order, in the third loop I first find the smallest 2 nodes to later be created as a combined node, then I let root be the topmost of the Huffman tree.



```
int main() {
    int n, freq[26] = {0}, k;
    string wrds[1000];

    cin >> n;
    k = n;
    while (n--) {
        string in;
        cin >> in;
        wrds[n] = in;
        for (char c : in) {
            freq[c - 'a']++;
        }
    }
```

For the frequency count I just have a 26 element array indexed by a-z.

The encoding begins by analyzing the frequency of each character (a–z) in the input data. A Huffman Tree is then constructed. Characters with lower frequencies are placed deeper in the tree, resulting in shorter binary codes. To support autocomplete and fuzzy search, each encoded string (bit string) corresponding to a valid word is inserted into a Bit-level Trie (BitTrie). Each node in the BitTrie represents a binary digit (`0` or `1`), and complete bit strings are stored at the bottom nodes.

Decoding is done by traversing through the tree bit by bit.
If a bit other than `'0'` or `'1'` is encountered in the encoded input, the decoder returns an empty string (`""`) to indicate invalid data.

這次 Huffman Trie 的實作讓我深入體會了資料結構與字元編碼的實際應用，尤其是將 Huffman Tree 的壓縮特性與 Trie 結構相結合，達成有效率的自動補全功能。在過程中，我學會了如何從字元頻率建構 Huffman Tree，再利用編碼結果插入 Bit Trie，進行快速搜尋與自動補全。此外，為了實作模糊搜尋，我也重新複習並實作了 Edit Distance 的動態規劃演算法。這讓我理解到模糊匹配的實作並不只是逐字比對，還需要從使用者角度考慮輸入誤差。整體而言，這份作業不僅加深了我對樹狀結構的掌握，也強化了我解決實際問題的能力。

收穫方面，我學會了如何結合多種資料結構來解決複雜問題，並理解了 Huffman 編碼與 Trie 在實務上的整合方式。而困難之處在於模糊搜尋的邏輯：起初我錯誤地將整個字串與目標進行比對，導致預期中的模糊自動補全無法正常運作。後來我才意識到，應該只比對與輸入字串等長的字首，這樣才能正確模擬「打錯字」的情況。這部分的修正過程讓我更認識問題分析與除錯的重要性。

整體來說，這份作業的設計很完整，涵蓋了壓縮、搜尋、自動補全與模糊比對等應用，對資料結構與演算法的整合能力是一個很好的挑戰。不過建議可以在題目說明中多提供一些具體的測資範例，特別是模糊搜尋的預期輸出，這樣可以幫助學生更早察覺邏輯上的誤差。此外，若能搭配簡單的視覺化工具，展示 Huffman Tree 與 Trie 的建構與查找過程，會更有助於理解。