Assignments week 2

Rickquan Nelson

2025-09-20

Assignment 1 - list_insertions

My code:

```
int main() {
    string input;
    getline(cin, input);

    size_t bracket_pos = input.find(']');

    if (bracket_pos == string::npos) {
        string new_word = input.substr(input.find_first_not_of(" "));
        new_word = new_word.substr(0, new_word.find_last_not_of(" ") + 1);

        if (new_word.empty()) {
            cout << "[]" << endl;
        } else {
            cout << "[" << new_word << "]" << endl;
        }
        return 0;
    }

    string list_part = input.substr(1, bracket_pos - 1);
    string new_word = input.substr(bracket_pos + 1);

    new_word = new_word.substr(new_word.find_first_not_of(" "));
    new_word = new_word.substr(0, new_word.find_last_not_of(" ") + 1);

    vector<string> words;
    size_t position = 0;

    while ((position = list_part.find_first_not_of(" ->", position)) != string::npos) {
        size_t word_end = list_part.find_first_of(" ->]", position);
        string word = list_part.substr(position, word_end - position);
        if (!word.empty()) {
            words.push_back(word);
        }
        position = word_end;
    }

    if (!new_word.empty()) {
        bool word_exists = find(words.begin(), words.end(), new_word) != words.end();
        if (!word_exists) {
            words.push_back(new_word);
            sort(words.begin(), words.end());
        }
    }

    cout << "[";
    for (size_t i = 0; i < words.size(); i++) {
        cout << words[i];
        if (i < words.size() - 1) {
            cout << " -> ";
        }
    }
    cout << "]" << endl;

    return 0;
}
```

Approach: I parsed the input string to extract the existing list of words and the new word to insert. I used string manipulation to handle the brackets and arrows. The algorithm checks if the new word already exists in the list, and if not adds it and sorts the list then formats the output with proper arrows.

Challenges: The main challenge was maintaining the correct list structure while ensuring no duplicates were added and maintaining alphabetical order.

Time complexity: this algorithm has a time complexity of O(n log n), because it sorts the list of words.

Assignment 4 - unsafe_buffer

My code:

```
int main() {
    int buffer_size;
    string input_string;
    cin >> buffer_size >> input_string;
    vector<char> buffer(buffer_size);
    int read = 0, write = 0, count = 0;
    for (char ch : input_string) {
        if (ch != '*') {
            if (count < buffer_size) {
                buffer[write] = ch;
                write = (write + 1) % buffer_size;
                count++;
            } else {
                buffer[write] = ch;
                write = (write + 1) % buffer_size;
            }
        } else {
            if (count > 0) {
                cout << buffer[read];
                read = (read + 1) % buffer_size;
            }
        }
    }
    return 0;
}
```

Approach: I implemented a circular buffer using a vector with read and write pointers. The algorithm processes each character, non-asterisk characters are enqueued and asterisk characters are dequeud and the oldest element is printed.

Challenges: Managing the circular buffer pointers with modulo was tricky. The main challenge was handling the buffer fullness correctly, when the buffer is full, new characters overwrite the oldest ones without changing the count, while asterisks only dequeue when the buffer is not empty.

Time complexity: this algorithm has a time complexity of O(n), because it processes each character in the input string once with constant time operations.

Assignment 5 - anagrams

My code:

```
int main()
{
    string word, sentence;
    getline(cin, word);
    getline(cin, sentence);
    word.erase(remove(word.begin(), word.end(), ' '), word.end());
    sort(word.begin(), word.end());

    string no_spaces = sentence;
    no_spaces.erase(remove(no_spaces.begin(), no_spaces.end(), ' '), no_spaces.end());

    size_t n = word.size();

    for (size_t i = 0; i <= no_spaces.size() - n; i++)
    {
        string sub = no_spaces.substr(i, n);
        sort(sub.begin(), sub.end());

        if (sub == word)
        {
            size_t count = 0;
            int start = 0, end = 0;

            for (size_t j = 0; j < sentence.size() && count <= i; j++)
            {
                if (sentence[j] != ' ')
                    count++;
                if (count == i + 1)
                    start = j;
            }

            count = 0;
            for (size_t j = 0; j < sentence.size() && count < i + n; j++)
            {
                if (sentence[j] != ' ')
                    count++;
                if (count == i + n)
                    end = j;
            }

            cout << sentence.substr(start, end - start + 1) << endl;
            return 0;
        }
    }

    cout << "<not found>" << endl;
    return 0;
}
```

Approach: I first processed the target word by removing spaces and sorting it. Then I processed the sentence by removing spaces and used a sliding window approach to check each possible substring. For each window, I sorted the substring and compared it with the sorted target word. When a match was found, I mapped the positions back to the original sentence with spaces to output the correct substring.

Challenges: Handling spaces in the input while maintaining correct character positions and mapping between the space removed version and the original sentence was challenging.

Time complexity: this algorithm has a time complexity of O(m $\times$ n log n), because it sorts each substring of length n for each position in the sentence of length m.