Assignments week 2

Son Cao

2025-09-20

Assignment 1 - Singly Linked List insertions

My code:

```cpp
#include <iostream>
#include "linked_list.h"

int main() {
    /* TODO:
        Write a program that reads a singly linked list of strings from standard input,
        followed by a string to insert into the list. The program should insert the string
        into the list such that the list remains sorted in alphabetic order. The program
        should print the updated list to standard output.

        Example input: [apple -> banana -> cherry] blueberry
        Example output: [apple -> banana -> blueberry -> cherry]
    */

    sax::linked_list_node<std::string>* head = nullptr;
    std::cin >> head;
    if(!std::cin){
        std::cerr << "Failed to read list from input" << std::endl;
        sax::linked_list_node<std::string>::cleanup(head);
        return 1;
    }

    // string to insert
    std::string to_insert;
    std::cin >> to_insert;

    //if the list is empty, create the first node
    if(head == nullptr){
        head = new sax::linked_list_node<std::string>{to_insert};
        std::cout << *head << std::endl;
        sax::linked_list_node<std::string>::cleanup(head);
        return 0;
    }

    //sentinel node
    sax::linked_list_node<std::string> sentinel{""};  //buffer to insert node before head
    sentinel.next = head;

    //traverse the list to find insertion point
    sax::linked_list_node<std::string>* prev = &sentinel;
    sax::linked_list_node<std::string>* current = head;

    bool inserted = false;

    while(current != nullptr){
        if(current->data == to_insert){  //if existed
            inserted = true;
            break;
        } else if (current->data > to_insert){
            //insert new node before current node (in the middle)
            sax::linked_list_node<std::string>* new_node = new sax::linked_list_node<std::st
            prev->next = new_node;
            new_node->next = current;
            inserted = true;
            break;
        }
        prev = current;
        current = current->next;
    }

    //If we reached the end without inserting, insert at the end
    if(!inserted){
        prev->next = new sax::linked_list_node<std::string>{to_insert};
    }

    //print the updated list(Skip sentinel)
    std::cout << *sentinel.next << std::endl;

    //cleanup dynamically allocated memory
    sax::linked_list_node<std::string>::cleanup(sentinel.next);

    return 0;
}
```

Time complexity: This algorithm has a time complexity of O(n), because in the worst case it must traverse the entire linked list to find the correct insertion point, and also traverses it again for printing and cleanup.

Assignment 2 - Singly linked list partitioning

My code:

```
#include <iostream>
#include "linked_list.h"

int main() {
    /* TODO:
        Write a program that reads a singly linked list of strings from standard input,
        followed two antegers i and n. The program should move the first n nodes starting f
        to the front of the list. The program should print the updated list to standard outp

        Example input: [apple -> banana -> cherry] 1 2
        Example output: [banana -> cherry -> apple]
    */
   sax::linked_list_node<std::string>* head;
   size_t low, count;
   std::cin >> head >> low >> count;

    if(count == 0 || head == nullptr){
        std::cout << head << std::endl;
        sax::linked_list_node<std::string>::cleanup(head);
        return 0;
    }

    if(low == 0){
        //already at the front
        std::cout << head << std::endl;
        sax::linked_list_node<std::string>::cleanup(head);
        return 0;
    }

    //find the node b4 sublist
    sax::linked_list_node<std::string>* before_sublist = head;
    for(size_t i = 0; i < low - 1; i++){  //loop until seeing the first node before sublist
        before_sublist = before_sublist->next;
    }

    sax::linked_list_node<std::string>* sublist_start = before_sublist->next;
    sax::linked_list_node<std::string>* sublist_end = sublist_start;
    for(size_t i = 1; i < count; i++){  //loop until the last node of sublist
        sublist_end = sublist_end->next;
    }

    //detach sublist
    before_sublist->next = sublist_end->next;

    //move sublist to the front
    sublist_end->next = head;
    head = sublist_start;

    std::cout << head << std::endl;

    sax::linked_list_node<std::string>::cleanup(head);

    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because in the worst case we may traverse the entire list to find the sublist boundaries.

Assignment 3 - Linked list union

My code:

```
#include <iostream>
#include "linked_list.h"

void appendNode(sax::linked_list_node<int>** head, sax::linked_list_node<int>** tail, sax::l
    node->next = nullptr; //no new node, only relink existing ones
    if(*head == nullptr){
        *head = *tail = node;
    } else {
        (*tail)->next = node; //append new node after current last node
        *tail = node;        //move tail pointer to the new last node
    }
}

int main() {
    /* TODO:
        Write a program that reads two singly linked lists of integers from standard input.
        The program should build a new list that forms the union of the two input lists, by
        rearranging the nodes of the input lists (i.e., do not create new nodes).

        The input lists are sorted in ascending order and do not contain duplicates.
        The output list should also be sorted in ascending order and should not contain dup

        The program should print the resulting list to standard output.
        Example input: [1 -> 2 -> 3] [2 -> 3 -> 4]
        Example output: [1 -> 2 -> 3 -> 4]
    */
    sax::linked_list_node<int>* head1 = nullptr;
    sax::linked_list_node<int>* head2 = nullptr;

    std::cin >> head1 >> head2;

    sax::linked_list_node<int>* a = head1;
    sax::linked_list_node<int>* b = head2;

    sax::linked_list_node<int>* head = nullptr;  //head of union list
    sax::linked_list_node<int>* tail = nullptr;  //tail of union list

    while(a && b){
        if(a->data < b->data){
            sax::linked_list_node<int>* next = a->next;
            appendNode(&head, &tail, a);
            a = next;
        } else if (a->data > b->data){
            sax::linked_list_node<int>* next = b->next;
            appendNode(&head, &tail, b);
            b = next;
        } else {
            sax::linked_list_node<int>* nextA = a->next;
            sax::linked_list_node<int>* nextB = b->next;
            appendNode(&head, &tail, a);  // a == b -> only append 1
            sax::linked_list_node<int>* temp = b;
            b = nextB;
            delete temp;      //free memory for the duplicate node
            a = nextA;
        }
    }

    //if a longer
    while(a){
        sax::linked_list_node<int>* next = a->next;
        appendNode(&head, &tail, a);
        a = next;
    }
    //if b longer
    while(b){
        sax::linked_list_node<int>* next = b->next;
        appendNode(&head, &tail, b);
        b = next;
    }

    std::cout << head << std::endl;
    sax::linked_list_node<int>::cleanup(head);

    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(m + n), because we traverse each node of the two input lists (m = length of list1, n = length of list2) at most once while merging.

Assignment 4 - Unsafe circular buffer

My code:

```
#include <iostream>
#include <string>
#include <memory>

class Buffer{

    public:

        Buffer(int size) :_size(size),  _buffer(std::make_unique<char[]>(_size)) {}

        void queue(char c) {
            _buffer[_head] = c;
            _head = (_head + 1) % _size;   //abc*def*gh
        }

        char dequeue() {
            char c = _buffer[_tail];
            _tail = (_tail + 1) % _size;
            return c;
        }

    private:
        const int _size;
        int _tail = 0;
        int _head = 0;
        std::unique_ptr<char[]> _buffer;
};


int main() {
    /* TODO:
        Write a program that reads an integer that specifies the size of a buffer,
        followed by a string to process.
        Each character (unless it is '*') should be enqueued into the buffer, and if the bu
        the oldest character should be dequeued to make space for the new character.

        In case a '*' character is encountered, the program should dequeue a character from
        and print it to standard output.

        All characters must appear on a single line, without spaces.
    */

    int buffer_size;
    std::string sentence;

    std::cin >> buffer_size >> sentence;
    Buffer buffer(buffer_size);

    for (char c : sentence){

    if (c == '*') {
            std::cout << buffer.dequeue();
        }
        else {
            buffer.queue(c);
        }
    }
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because each character in the input string is processed exactly once, and both enqueue and dequeue operations take constant time O(1).

Assignment 5 - Sliding window anagram detection

My code:

```
#include <iostream>
#include <string>
#include <array>

int main() {
  /* TODO:
      Write a program that reads two lines of input: the first line contains a word,
      and the second line contains a sentence that may contain an anagram of the word (igno
      The program should find the first anagram of the word in the sentence and print it.
      If no anagram is found, the program should print "<not found>".

      Example input:
        conversation
        our voices rant on forever
      Example output: voices rant on
  */
  std::string target, text;
  std::getline (std::cin, target);
  std::getline(std::cin, text);

  //target frequency
  std::array<int, 26> target_freq{};
  int target_len = 0;
  for(char c : target){
    if(isalpha(c)){
      target_freq[c - 'a']++;
      target_len++;
    }
  }

  //sliding window
  std::array<int, 26> window_freq{};
  int left = 0, count = 0;

  for (int right = 0; right < (int)text.size(); right++){
    char rc = text[right];
    if(isalpha(rc)){
      window_freq[rc - 'a']++;
      count ++;
    }

    //if window too big -> shrink from left
    while(count > target_len){
      char lc = text[left++];
      if(isalpha(lc)){
        window_freq[lc - 'a']--;
        count --;
      }
    }

    //if window size matches target length -> anagram
    if(count == target_len && window_freq == target_freq){
      //skip non-alphabetic at the start (quotes, punctuation)
      while(left < (int)text.size() && !isalpha(text[left])){
        left++;
      }
      std::cout << text.substr(left, right - left + 1) << std::endl;
      return 0;
    }
  }

  //no match
  std::cout << "[]" << std::endl;
  return 0;
}
```

Time complexity: this algorithm has a time complexity of O(m + n), because each character in the target and the text is processed at most once, and frequency comparisons are constant time.

Assignment 6 - Subarray sum

My code:

```
#include <iostream>
#include <vector>
#include "utils.h"

int main() {
    /* TODO:
        Write a program that reads a vector of integers from standard input,
        followed by a number N. The program should find a contiguous (i.e. without gaps) su
        of the input vector that sums to N.
        If such a subvector is found, the program should print it.
        If no such subvector exists, the program should print an empty vector ("[]").

        Example input: [1, 2, 3, 4, 5] 9
        Example output: [2, 3, 4]
    */
    std::vector<int> arr;
    std::cin >> arr;

    if(!std::cin){
        std::cerr << "Failed to read input vector\n";
        return 1;
    }

    int target;
    std::cin >> target;
    size_t left = 0;
    int current_sum = 0;
    bool found = false;

    //sliding window
    for(size_t right = 0; right < arr.size(); right++){
        current_sum += arr[right];

        //shrink window if sum too largex
        while(current_sum > target && left <= right){
            current_sum -= arr[left];
            ++left;
        }

        //check for match
        if(current_sum == target){
            std::vector<int> sub(arr.begin() + left, arr.begin() + right + 1);
            std::cout << sub << std::endl;
            found = true;
            break;
        }
    }
    if(!found){
        std::cout << "[]" << std::endl;
    }
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because each element of the vector is added and removed from the sliding window at most once.