Assignments week 3

Son Cao

2025-09-28

Assignment 1 - Use recursion to reverse a singly linked list

My code:

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

sax::linked_list_node<std::string>* reverse_recursive(sax::linked_list_node<std::string>* he
    //if empty
    if(head == nullptr || head->next == nullptr){
        return head;
    }

    sax::linked_list_node<std::string>* new_head = reverse_recursive(head->next);

    //put current head at the end of the reversed list
    head->next->next = head;
    head->next = nullptr;

    return new_head;
}

int main() {
    /* TODO:
        Write a program that reads a singly linked list of strings from standard input,
        reverses the list using recursion, and prints the reversed list to standard output.

        Example input: [apple -> banana -> cherry -> blueberry]
        Example output: [blueberry -> cherry -> banana -> apple]
    */
    sax::linked_list_node<std::string>* head = nullptr;
    std::cin >> &head;
```

```
    head = reverse_recursive(head);
    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 the function is called once per node in the list, and each call performs only constant-time operations.

Assignment 2 - Evaluating expressions

My code:

```
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>

//recursive evaluator
int eval(std::istringstream& in){
    while(in.peek() == ' ') in.get();

    if(std::isdigit(in.peek())){
        int value;
        in >> value;
        return value;
    }

    if(in.peek() == '('){
        in.get();

        int left = eval(in);
        while(in.peek() == ' ') in.get();

        char op;
        in >> op;

        int right = eval(in);
        while(in.peek() == ' ') in.get();
        if(in.get() != ')') throw std::runtime_error("Missing closing parenthesis");

        switch(op){
            case '+': return left + right;
            case '*': return left * right;
```

```
            case '%':
                if (right == 0) throw std::runtime_error("Modulo by zero");
                return left % right;
            default:
                throw std::runtime_error("Invalid operator");
        }
    }
    throw std::runtime_error("Invalid expression");
}
int main() {
    /* TODO:
        Write a program that reads an expression that consists of positive integers, parenth
        the operators +, * and % (modulo) from standard input, and evaluates the expression.

        On each side of an operator you will either find an integer or a parenthesized expre
        so there is no need to consider operator precedence.

        Example input: ((1 + 2) * (3 + 4)) % 11
        Example output: 10

        Hint: Create a stringstream from the input line, and pass it to a recursive functio
        that evaluates the expression and returns the result.
    */
    std::string line;
    while(std::getline(std::cin, line)){
        if(line.empty()) continue;  //continue: skip this iteration, start the loop again
        try{
            std::istringstream in(line);
            int result = eval(in);
            std::cout << result << std::endl;
        } catch (const std::exception& e){
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because each character of the input expression is read and processed exactly once, and every recursive call does only constant work besides reading its subexpression.

Assignment 3 - Quick sort

My code:

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <iomanip> //std::quoted
#include <fstream>
#include "counter.h"

std::vector<std::string>::iterator partition(std::vector<std::string>::iterator begin, std:
    auto mid = begin + (std::distance(begin, end) / 2);
    auto pivot_value = *mid;

    //move pivot to the end temporarily
    std::iter_swap(mid, end - 1);
    sax::counter::instance().inc_swaps();

    std::vector<std::string>::iterator store = begin;  //store keeps track of the boundary 
    for (std::vector<std::string>::iterator it = begin; it != end; it++){
        sax::counter::instance().inc_comparisons();
        if(*it < pivot_value){
            std::iter_swap(it, store);
            sax::counter::instance().inc_swaps();
            ++store;
        }
    }
    std::iter_swap(store, end - 1);
    sax::counter::instance().inc_swaps();
    return store;
}

    //recursive quick sort
    void quick_sort(std::vector<std::string>::iterator begin, std::vector<std::string>::iter
        if(std::distance(begin, end) > 1){
            auto pivot_pos = partition(begin, end);  //get pivot position
            quick_sort(begin, pivot_pos);       //sort elements b4 pivot
            quick_sort(pivot_pos + 1, end);   //sort elements after pivot
        }
    }

int main() {
    /* TODO:
        Write a program that reads the names of two files from the standard input,
        reads the strings from the first file into a vector, sorts the vector using quick s
        and then writes the sorted strings to the second file.

        When sorting the strings, the program should count the number of comparisons and sw
        and print these counts to standard output.
```

```
        Use the sax::counter class for this (see counter.h and counter.cpp).
    */
    std::string input_name, output_name;
    std::getline(std::cin, input_name);
    std::getline(std::cin, output_name);

    std::ifstream fin(input_name);
    if(!fin){
        std::cerr << "Error opening input file " << std::quoted(input_name) << "\n";
        return 1;
    }

    std::vector<std::string> data;
    std::string word;
    while(fin >> word){
        data.push_back(word);
    }
    fin.close();

    //sort with quicksort
    quick_sort(data.begin(), data.end());

    std::ofstream fout(output_name);
    if(!fout){
        std::cerr << "Error opening output file " << std::quoted(output_name) << "\n";  //s
        return 1;
    }

    for(const auto& s : data){
        fout << s << "\n";
    }

    fout.close();

    std::cout << "Data size: " << data.size() << "\n";
    std::cout << "Comparisons: " << sax::counter::instance().comparisons() << "\n";
    std::cout << "Swaps: " << sax::counter::instance().swaps() << "\n";

    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n log n), because each partitioning step scans the subarray once in O(n), and the recursion depth is O(log n) when partitions are reasonably balanced.

Assignment 4 - Merge sort

My code:

```
#include <iostream>
#include <string>
#include <fstream>
#include "counter.h"
#include "linked_list.h"

//split to 2 lists
sax::linked_list_node<std::string>* split(sax::linked_list_node<std::string>* head){
    if(!head || !head->next) return nullptr;
                                        // apple -> pear -> banana -> lemon -> nullptr
    auto slow = head;                  //slow => apple
    auto fast = head->next;            //fast => pear
    while (fast && fast->next){
        slow = slow->next;              //slow moves 1 step
        fast = fast->next->next;        //fast moves 2 steps
    }
                                                                        //fast reach nullpt
    sax::linked_list_node<std::string>* second_half = slow->next;        //apple -> pear(sl
    slow->next = nullptr;                                          //break => apple -> pear -> n
    return second_half;                                            //      => banana -> lemon ->
}

//just merge, can be unsorted
sax::linked_list_node<std::string>* merge(sax::linked_list_node<std::string>* l1, sax::linke
    sax::linked_list_node<std::string> dummy;  //fake head
    auto tail = &dummy;

    while(l1 && l2){
        sax::counter::instance().inc_comparisons();
        if(l1->data <= l2->data){
            tail->next = l1;
            l1 = l1->next;
        } else {
            tail->next = l2;
            l2 = l2->next;
        }
        tail = tail->next;
    }                                       //dummy(tail) -> apple -> banana -> grape -> o
    tail->next = (l1) ? l1 : l2;        //for longer list, merge it in
    return dummy.next;            //skip dummy => apple -> banana -> grape -> orange -> peac
}
```

```cpp
//recursively sort any unsorted list and then merge sublists
sax::linked_list_node<std::string>* merge_sort(sax::linked_list_node<std::string>* head){
    if(!head || !head->next) return head;

    sax::linked_list_node<std::string>* second = split(head);
    head = merge_sort(head);            //each half keep splitting until every sublist has
    second = merge_sort(second);

    return merge(head, second);     //then merge each list and overall 2 lists  (check agai
}

int main() {
    /* TODO:
        Write a program that reads the names of two files from the standard input,
        reads the strings from the first file into a linked list, sorts the linked list usi
        and then writes the sorted strings to the second file.

        When sorting the strings, the program should count the number of comparisons perfor
        and print this count to standard output.

        Use the sax::counter class for this (see counter.h and counter.cpp).
    */
    std::string input_file, output_file;
    std::getline(std::cin, input_file);
    std::getline(std::cin, output_file);

    std::ifstream fin(input_file);
    if(!fin){
        std::cerr << "Error opening input file " << input_file << "\n";
        return 1;
    }

    sax::linked_list_node<std::string>* head = nullptr;
    sax::linked_list_node<std::string>* tail = nullptr;
    std::string word;      //cannot use fin >> head; cuz the text is like "apple banna oran
    while (fin >> word){
        sax::linked_list_node<std::string>* new_node = new sax::linked_list_node<std::string
        new_node->data = word;
        new_node->next = nullptr;

        if(!head){
            head = new_node;            //first node
            tail = head;
        } else {
            tail->next = new_node;      //append new one
            tail = new_node;
```

```
        }
    }

    fin.close();

    head = merge_sort(head);

    std::ofstream fout(output_file);
    if(!fout){
        std::cerr << "Error opening output file " << output_file << "\n";
        sax::linked_list_node<std::string>::cleanup(head);
        return 1;
    }

    //write 1 word per line
    auto* curr = head;
    while(curr){
        fout << curr-> data << "\n";
        curr = curr->next;
    }
    fout.close();

    std::cout << "Comparisons: " << sax::counter::instance().comparisons() << "\n";

    sax::linked_list_node<std::string>::cleanup(head);
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n log n), because at each level of recursion the list is split and merged in linear time O(n), and there are O(log n) levels of recursion.

Assignment 5 - Compare sorting algorithms

Part 1: run algorithms on non-sorted data

Algorithm	Comparisons	Swaps
Selection sort	348387606	26375
Quick sort	3542168	243667
Merge sort	354664	

Part 2: run algorithms on sorted data

Algorithm	Comparisons	Swaps
Selection sort	348387606	0

```
| Quick sort     |   4448219   | 159700  |
| Merge sort     |    200188   |         |
```

- **Selection sort** always does the same number of comparisons (`n*(n-1)/2`), always has to check every pair of elements to find the smallest one. That's why the number of comparisons doesn't change, no matter if the list is already sorted or completely random. But if the list is already sorted, it ends up doing far fewer swaps because most elements are already in the right place.
- **Quick sort** usually works very fast on unsorted lists, roughly cutting the problem in half each time `O(n log n)`. But if you always pick the last element as the pivot, and the list is already sorted, it ends up doing the worst possible thing: splitting off only one element each time. This makes it very slow `O(n²)` and can even crash your program if the list is big. Picking the middle element or a random pivot fixes this problem.
- **Merge sort** consistently performs about `O(n log n)` comparisons regardless of input order. Always splits the list and merges it back together. This gives it a predictable number of comparisons, no matter whether the list is sorted or not. It doesn't swap elements in place—it builds new merged lists so it's stable and consistent. The downside is it uses more memory because of all the merging and recursion.