MECHTRON 2MD3
Data Structures and Algorithms for Mechatronics
Winter 2022

Tutorial 06

Published: Feb 25, 2022

I recommend you try to solve the problems by yourself before the tutorial. Problems will be solved during the tutorial hours. I have uploaded the companion code in the same folder as this document.

1 Question 1

In the tutorial no. 4, we discussed about adding extra functionalities to the **Singly Linked List**. These functions were added: *printList*, *size*, *addTail*, *concatenate* and *reverse*.

There was a specific pattern of coding in all of those implementations; we traversed through the linked list using a pointer, starting from the head and hopping the pointer at each **iteration of the loop**, until we arrived at the last node.

You may remember from our discussions on recursion that, with recursion, you can substitute loops. Also, any algorithm that is implemented with loops can be implemented with recursion. Designing algorithms using loops for some data structures (such as linked structures and trees) requires thinking extensively about how to manipulate pointers to perform the task. One thing that recursion can do that loops can't is to make such tasks very easy and straightforward to implement. Similar to solving linked structure-related problems with loops, we have algorithmic patterns in solving those problems with recursion that we will discuss next.

We bring the class declaration of **SNode** and **SLinkedList** in the following which are suitable for storing integer data. The related code is provided in the same directory of this tutorial's content in Teams under: "General > Class Materials > Tutorials > Tutorial 06 > singly-linked-list-recursive.cpp"

```
typedef int Elem;
class SNode {
    private:
        Elem elem;
        SNode* next;

    friend class SLinkedList;
};
```

```
class SLinkedList {
    public:
        SLinkedList();
        ~SLinkedList();
        bool empty() const;
        const Elem& front() const;
        void addFront(const Elem& e);
        void removeFront();

    private:
        SNode* head;
};
```

1.1 Implementation Notes

When designing algorithms for linked list operations using recursion you have to think about the fact that a linked list can also be **defined recursively**. This means:

- A *null* pointer is a linked list that is empty.
- Every node object's *next* pointer refers to a linked list. That linked list can be either *null* or a non-empty linked list.

In the example shown in Figure 1, I will call nodes with their elements. You can think of node 12 pointing to a linked list with three elements 16, 7, and 4. Also, node 16 is pointing to a linked list with two elements 7 and 4. Node 7 is a node with pointer to another linked list that has only node 4. Finally, node 4 points to a linked list, which is a null pointer (as defined above).

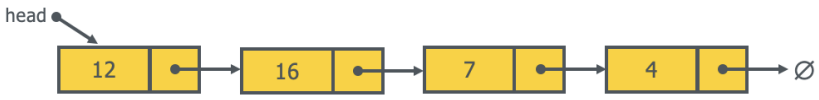

Figure 1: a sample SLinkedList

We will use above definition to specify the base case and recursive step of the recursive algorithms.

In the following, I provide the recursive algorithm for obtaining the size of the linked list using the *_sizeRec* function.

```
int SLinkedList::_sizeRec(SNode* node){
    if (node == NULL)
        return 0;
    else
        return 1 + _sizeRec(node->next);
}
```

The function receives a pointer to a node as the parameter. If that node is *Null*, which means the linked list is *Null* (base case), the size of the list is zero. Otherwise, we can obtain the size by calling the function recursively on the rest of the linked list (the list that starts from the current node's next node) and adding one to the result. Adding one means that we counted the current node.

Figure 2 shows the recursion trace for the algorithm running on the SLinkedList of Figure 1.

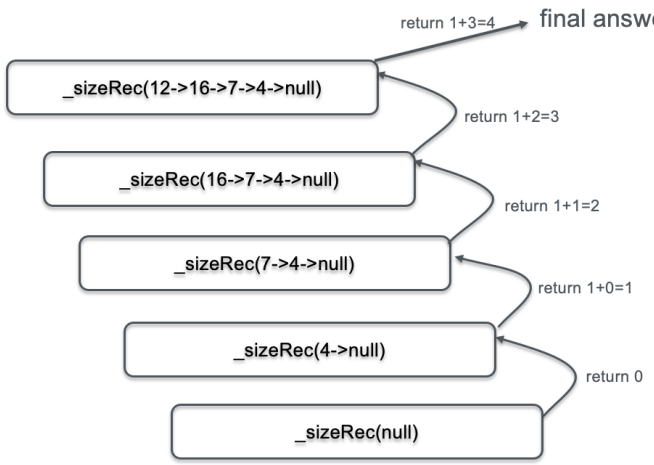

Figure 2: Recursion Trace for _sizeRec

Later, when we talk about the correctness of programs we will see how easy it is to prove that a given recursive algorithm works correctly. Roughly speaking, the correctness of a recursive algorithm can be proved mathematically.

The proof of correctness for this function is easy. The method returns zero for the base case (empty list) which is definitely true and return the correct size. In the recursive step, the function calls itself on a linked list with smaller size (using node's next) which contains one fewer node. So, we can ensure that the algorithm terminates. Assuming that `_sizeRec(node->next)` correctly computes the size of the given list, then returning a value one bigger than its returned value will be the correct length of the entire list.

You may have noticed that the user of this function must provide a pointer to node as the parameter of this function. A natural choice for this parameter would be the **head** node. It would be much better if we could somehow lift the burden of providing such a node pointer from the user. We can do this using the combination of public and utility functions. We can define *_sizeRec* to be a utility function and define another public member function in our class that calls utility function with appropriate parameter.

```
int SLinkedList::sizeRec(){
    return _sizeRec(head);
```

```
}
```

Note that it is a good programming practice to name utility functions with a leading underscore.

And this will be the class definition:

```
class SLinkedList {
    public:
        // methods here
        int sizeRec();
    private:
        SNode* head;
        int _sizeRec(SNode* node);
};
```

Now, the user of the class can easily call the *sizeRec()* on a *SLinkedList* object and *sizeRec()* will take care of recursive calls and returning the result.

1.2 Adding Recursive Functions

In this exercise, you are asked to add the following new functionalities to our *SLinkedList* class, all using **recursion**:

- Add a member function *sumRec* and its corresponding utility function that obtain the sum of values stored in the list. Pay attention to the return type of this function.
- Add a member function *printRec* and its corresponding utility function that print the elements of the list. Hint: You should visit a node recursively and print the visited node. Then you should make the recursive call.
- Add a member function *printReverseRec* and its corresponding utility function that print the elements of the list in the reverse order. Hint: Again think recursively. If you implement *printRec*, then you really need to make a very small change to the order of statements. This time, you should make a recursive call and then print the node. Try on the paper.
- Add a member function *containsRec* and its corresponding utility function that identify whether an element is in the list. You have already seen this question in your first mid-term.
- Add a member function *addTailRec* and its corresponding utility function that receive an element as the parameter and adds it to the end of the list. Hint: The utility function call is a bit different here. Note that you are going to manipulate the list. Thus, you have to manipulate the *head*. The base case just creates the node and return its pointer.

- Add a member function *reverseRec* and its corresponding utility function that reverses the list. Hint: We have two base cases here! Think about them. Other than that, you have to think about reversing the rest of the list and assuming that the reverse works correctly on the rest, you have to manipulate head's pointer appropriately.

2 Question 2

Write an iterative C++ program that computes the greatest common divisor (GCD) of two integer values. Try using Euclid's Algorithm for implementing the recursive version of the program. Try using Dijkstra's Algorithm for implementing the recursive version of the same program. Compare the recursive algorithms using the recursion trace.

Hint: The recursive relations of the two methods are given below:

Euclid's algorithm:

$$gcd(m,n) = \begin{cases} n, & \text{if n divides m without remainder} \\ gcd(n,\ \text{remainder of } \frac{m}{n}), & \text{otherwise} \end{cases}$$

Dijkstra's algorithm:

$$gcd(m,n) = \begin{cases} m, & \text{if } m = n \\ gcd(m-n,n), & \text{if } m > n \\ gcd(m,n-m), & \text{if } m < n \end{cases}$$