EC-200 Data Structures

LAB MANUAL # 06

Course Instructor: Dr. Anum Abdul Salam

Lab Engineer: Engineer Hira Irshad

Student Name: ____

Degree/ Syndicate: ____

	Trait	Obtained Marks	Maximum Marks
R1	**Application Functionality 20%**		20
R2	**Specification & Data structure implementation 30%**		30
R3	**Reusability 10%**		10
R4	**Input Validation 10%**		10
R5	**Efficiency 20%**		20
R6	**Delivery 10%**		10
R7	**Plagiarism above 60%**		0
Total			100

Total Marks = $\boldsymbol{Obtained\ Marks}\ (\sum_{1}^{6} \boldsymbol{R_i} * \boldsymbol{R_7})$

Lab # 07: Recursion

Lab Objective:

To Practice recursion/ think recursively.

Lab Description:

A Recursive task is a task that is defined in the terms of itself. Recursive function calls itself. With each invocation, the problem is reduced to a smaller task (reducing case) until the task arrives at some terminal case. A recursive function has two parts:

1. Terminal/base case. a stopping conditions.
2. Reducing case/recursive step an expression of the computation or definition in terms of itself.

Example
if (terminal_condition) terminal_case else reducing_case

The factorial function:

n! = n * (n-1) * (n -2) * … * 2 * 1. The same function can be defined recursively as follows:

1. 0! = 1 – terminal case
2. n! = n * (n - 1)! - the reducing case

Example
int fact (int n) if (n==0) // base case return 1; else // reducing case return n*fact(n-1)

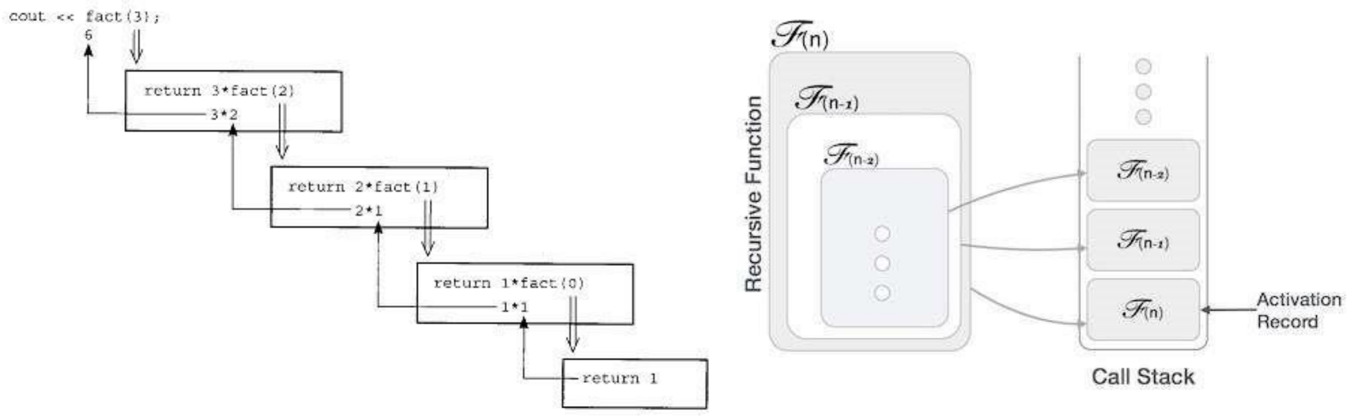


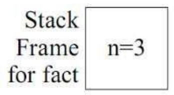
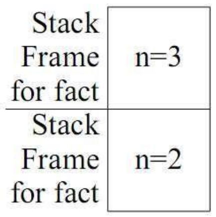
Figure 8.1: Activation Record Management during function call

Activation Records:

Programming languages typically have local variables. That is created upon entry to function and Destroyed when function returns. Each invocation of a function has its own instantiation of local variables. Recursive calls to a function require several instantiations to exist simultaneously. Functions return only after all functions it calls have returned last-in-first-out (LIFO) behavior. A LIFO structure called a stack is used to hold each instantiation. The portion of the stack used for an invocation of a function is called the function's stack frame or activation record.

Example:

Let us consider the above factorial example to understand activation records.

Step	Function call	Action	Stack state
1.	Fact(3)	Call generated from main	
2.	n*Fact(2)	Call generated from fact(3). Fact(3) pushed on stack.	
3	n*Fact(1)	Call generated from Fact (2). Fact (2) pushed on stack.	

4	n*Fact(0)	Call generated from Fact (1). Fact (1) pushed on stack.	Stack Frame for fact | n=3 Stack Frame for fact | n=2 Stack Frame for fact | n=1
5	1*1	Return 1 from base case. Fact(1) is popped from stack	Stack Frame for fact | n=3 Stack Frame for fact | n=2
6	2*1	Fact(2) popped from stack	Stack Frame for fact | n=3
7	3*2	Fact(3) popped from stack	Stack empty

Recursion elimination using stacks:

Recursion is implemented in via the system stack. Each recursive call to a method requires that the following information be pushed onto the system stack:

- Formal Parameters
- Local Variables
- Return Address

This information, collectively, is referred to as a stack frame. The stack frame for each method call will be different. For example, the stack frame for a parameter less/no local variable method will contain just a return address. All this is transparent to us as users because this happens magically due to the compiler. When we make a call to a recursive method, the compiler has taken care of the entire above required stack maintenance. To aid in our understanding of the process that takes place during the execution of a typical recursive routine, it is instructional to simulate it by managing this stack frame ourselves (not let C++/Java do it) with our own Stack. The following strategy can be used to the remove the recursion from a recursive routine, although not elegantly. There might be a far more pleasing method for a particular routine, but this technique is very instructional. It allows you simulate the system stack by declaring your own stack structure and manage the recursion. It is accomplished as follows:

1. Each time a recursive call is made in the algorithm, push the necessary information onto your stack.
2. When you complete processing at this deeper level, pop the simulated stack frame and continue processing in the higher level at the point dictated by the return address popped from the stack.
3. Use an iterative control structure that loops until there are no return addresses left to pop from the stack.

The use of a switch statement might help with the return address location.

LAB TASKS

1. Given an array of integers, construct a binary tree by inserting the elements of array using Recursion. The insertion operation must be implemented using a recursive function.

2. Write a recursive function to search for a value in BST. The function should return an appropriate indicator of found or not found.

3. Write a recursive function to calculate height of BST.

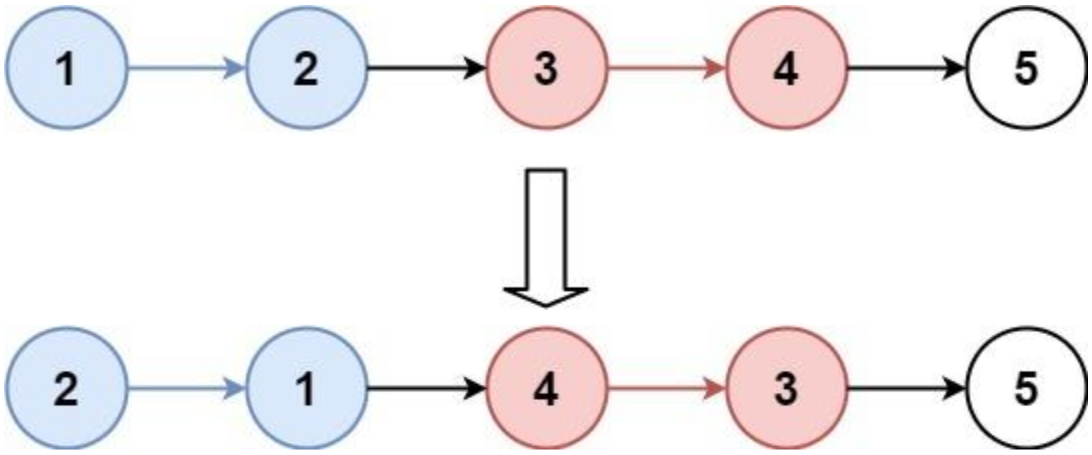
4. Find smallest and largest element in BST recursively.

5. Given the head of a linked list, reverse the nodes of the list k at a time, and return *the modified list*.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

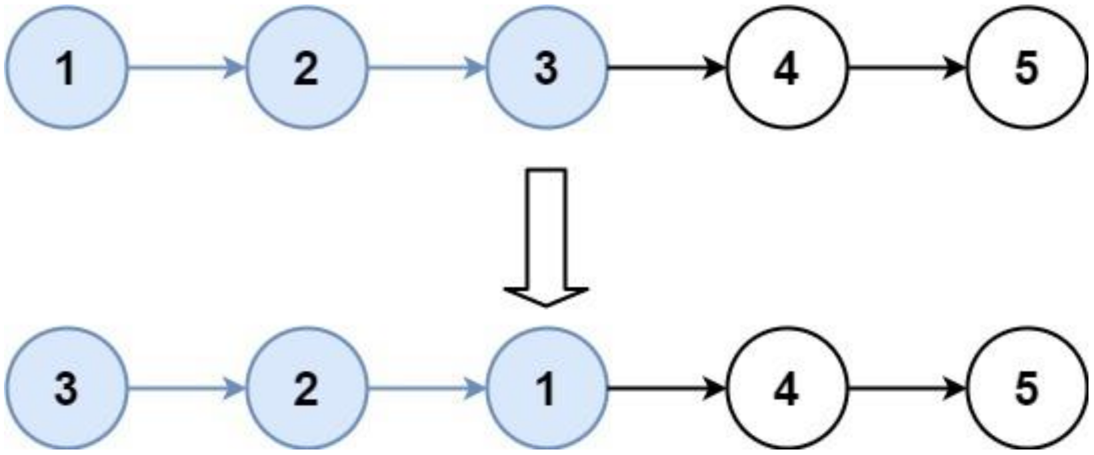
Example 1:



Input: head = [1,2,3,4,5], k = 2

Output: [2,1,4,3,5]

Example 2:



Input: head = [1,2,3,4,5], k = 3

Output: [3,2,1,4,5]

Constraints:

- The number of nodes in the list is n.
- 1 <= k <= n <= 5000
- 0 <= Node.val <= 1000

Link: https://leetcode.com/problems/reverse-nodes-in-k-group/description/?envType=problem-list-v2&envId=recursion

You need to make a profile on Leet Code and submit this task on Leet Code. Attach a screenshot of your submission on Leet Code in your lab report.

6. Compute time complexity of each recursive function implemented in Tasks 1–5.