NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
CS 201–DATA STRUCTURES LAB
Lab Session 04

Instructors: Ms Aqsa Zahid, Ms. Mubashra

Outline

- Functions
- Recursion
- Direct and Indirect Recursion
- Linked List
- Task

F

A function is a group of statements that together perform a particular task. Every C++ program has at least one function and that is a main function.

Based on the nature of task, we can divide up our program into several functions.

What are local variables?

These are the variables that are declared in the function. Their lifetime ends when the execution of the function finishes and are only known in the function in which they are declared.

Function returns a value

Value returning functions are used when only one result is returned and that result is used directly in an expression.

General Format of a function return a value

```
datatype nameOfFunction()
{
return variable;
}
```

Void Function

Void functions are used when function doesn't return a value.

General Format of a void function

```
void nameOfFunction()
{
Statement1; Statement2;
…
Statement n;
}
```

R

When a function repeatedly calls itself, it is called a recursive function and the process is called recursion.

It seems like a never ending loop, or more formally it seems like our function will never finish. In some cases, this might true, but in practice we can check if a certain condition becomes true then return from the function.

Base Case

The case/condition in which we end our recursion is called a base case.

Example of finite recursion

```
#include<iostream> using namespace std;
void myFunction( int counter)
{
if(counter == 0)
return;
else
{
cout <<counter<<endl; myFunction(--counter); return;
}
}

int main()
{
myFunction(10);
}
```

Characteristics of Recursion

Every recursion should have the following characteristics.

1. A simple base case which we have a solution for and a return value. Sometimes there are more than one base cases.
2. A way of getting our problem closer to the base case. i.e. a way to chop out part of the problem to get a somewhat simpler problem.
3. A recursive call which passes the simpler problem back into the function.

General Format

```
returntype recursive_func ([argument list])
{
    statements;
    recursive_func ([actual argument])
}
```

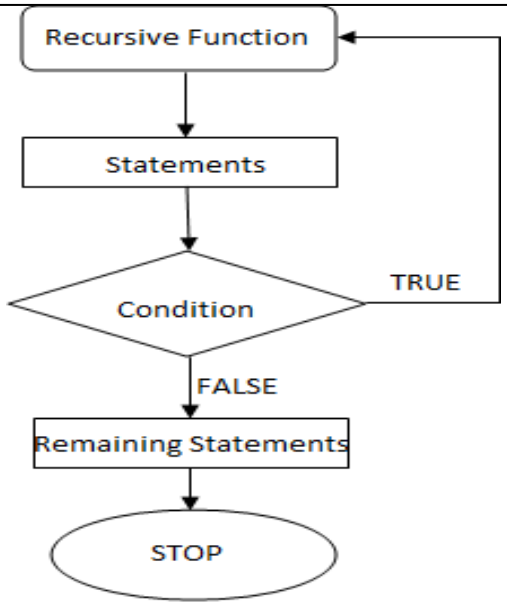


Fig: Flowchart showing recursion

DIRECT Vs INDIRECT RECURSION

There are two types of recursion, direct recursion and indirect recursion.

1. *Direct Recursion*

A function when it calls itself directly is known as Direct Recursion.

Example of Direct Recursion

```
#include<iostream> using
namespace std; int factorial
(int n)
 {
            if (n==1 || n==0)
        return 1;
         else
                return n*factorial(n-1);
 }

 int main()
 {
                  int f =
        factorial(5); cout << f;
 }
```

2. Indirect Recursion

A function is said to be indirect recursive if it calls another function and the new function calls the first calling function again.

Example of Indirect Recursion

```
 #include<iostream> using
namespace std; int func1(int);
 int func2(int); int
func1(int n)
 {
                if
         (n<=1)
         return 1;
          else
                    return func2(n);
 }
 int func2(int n)
 {
          return func1(n-1);
 }
 int main()
 {
                     int f =
         func1(5); cout << f;
 }
```

Here, recursion takes place in 2 steps, unlike direct recursion.

- First, func1 calls func2
- Then, func2 calls back the first calling function func1.

Disadvantages of Recursion

- Recursive programs are generally slower than non recursive programs. This is because, recursive function needs to store the previous function call addresses for the correct program jump to take place.
- Requires more memory to hold intermediate states. It is because, recursive program requires the allocation of a new stack frame and each state needs to be placed into the stack frame, unlike non-recursive(iterative) programs.

Linked List Basics

A linked list is a sequence of data structures, which are connected together via links.
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – each link of a linked list can store a data called an element.
- **Next** – each link of a linked list contains a link to the next link called Next.
- **Linked-List** – A Linked List contains the connection link to the first link called First.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

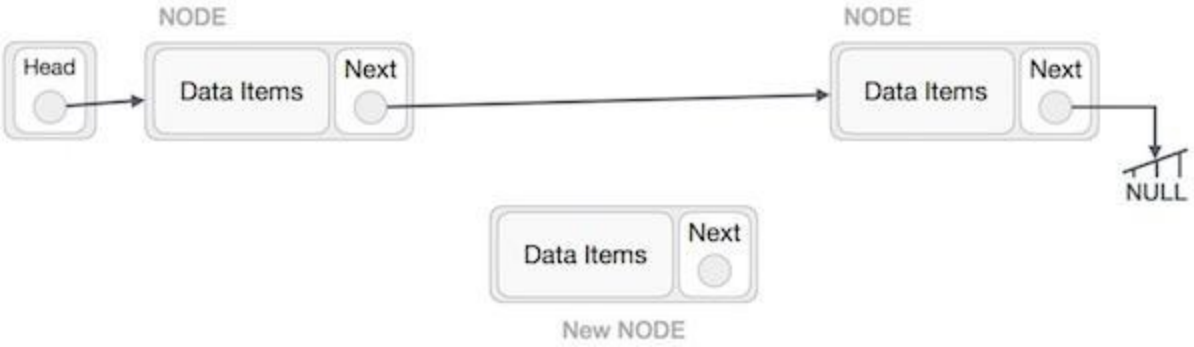
Single Linked List:

Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

Insertion Operation

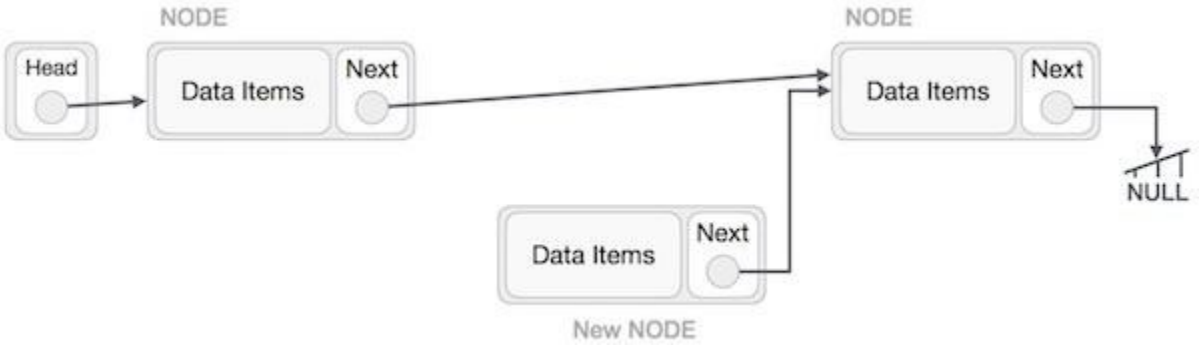
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C –

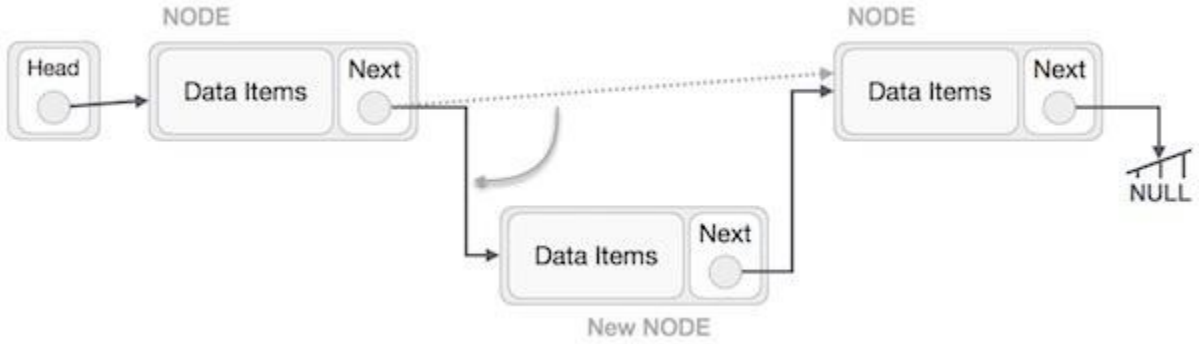
NewNode.next –> RightNode;

It should look like this –



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

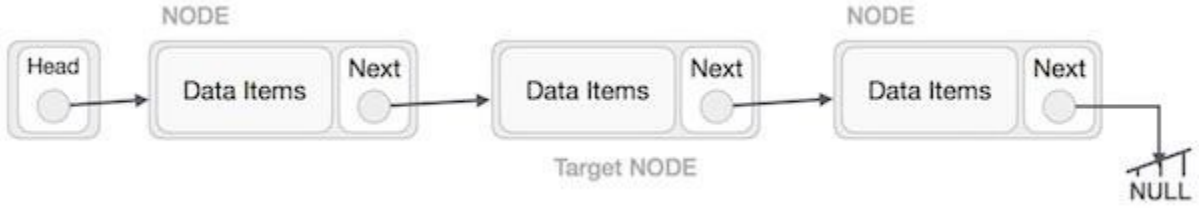


This will put the new node in the middle of the two. The new list should look like this –

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



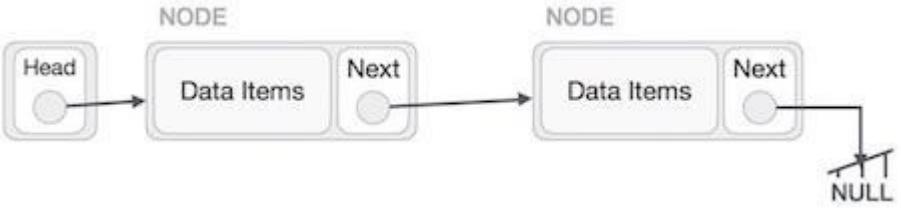
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Exercise:

Question #1

Write a recursive function named Sum_Num(int* , int) which receives an integer array and it's size and returns the sum of even numbers in the array. Call this function from main. Use appropriate parameters and return type.

Question #2

Write a recursive function to reverse the array.

Question #3

Write a Count () function that counts the number of times a given int occurs in a list. The code for this has the classic list raversal structure as demonstrated in Length ().

```
void CountTest() {
List myList = BuildOneTwoThree(); // build {1, 2, 3}
int count = Count(myList, 2); // returns 1 since there's 1 '2' in the list
}
/*
 Given a list and an int, return the number of times that int occurs
 in the list.
*/
int Count(struct node* head, int searchFor) {
// Your code
```

Question #4

Write a program that prompts the users to enter 12 numbers. This program reads the numbers into a linked list. Make another Linked list that will store the average of numbers.
The two lists will be passed to a function for and the average will be calculated by

a. Take First Four nodes of "numbers list" calculate their average and store at first node in "Average linked list".

b. Next time skip the first node of "numbers list" and average the next 4 nodes and store at second node in Average linked list.

c. And this procedure will continue, until Average for all will be stored in Averagelist.