COMP2401— Tutorial 7
Function call implementation and Linked Lists

Learning Objectives

After this tutorial, you will:

- Understand how function call is implemented
- How to create and use a linked list and clean up at end of its life-time

Tutorial

1 Warm up: how is a function call implemented?

Purpose: to understand how a function call is implemented using a stack.

Get **stackExample.c** from the code part of Chapter 3. Compile, run and understand the output and the figures on pp108-111 of Chapter 3. Ask us if you don't understand something from the notes or the code.

2 Linked List Creation

2.1 Introduction

In this part you will implement several basic linked list functions. The linked list is designed to manipulate and store employee records. Here we term the field that holds the employee information "data". The reason is that in general one would implement a generic linked list that can be used with any data type (using void *).

Get **myLinkedList.c** in which you are given employee structures and operations as well as the linked list structures and the main function. You have to complete the following six functions for the linked list:

a. **addNode** –adds a record as the first node of the linked list
b. **printList** – print the list from the first node to the last node
c. **printListRecursive** – print the nodes from first to last using recursion
d. **printThirdLast** – print the third node from the end of the list
e. **printListInReverse** – prints the list in reverse from the node to the first node using recursion.
f. **deleteNode** – delete the first node from the linked list.

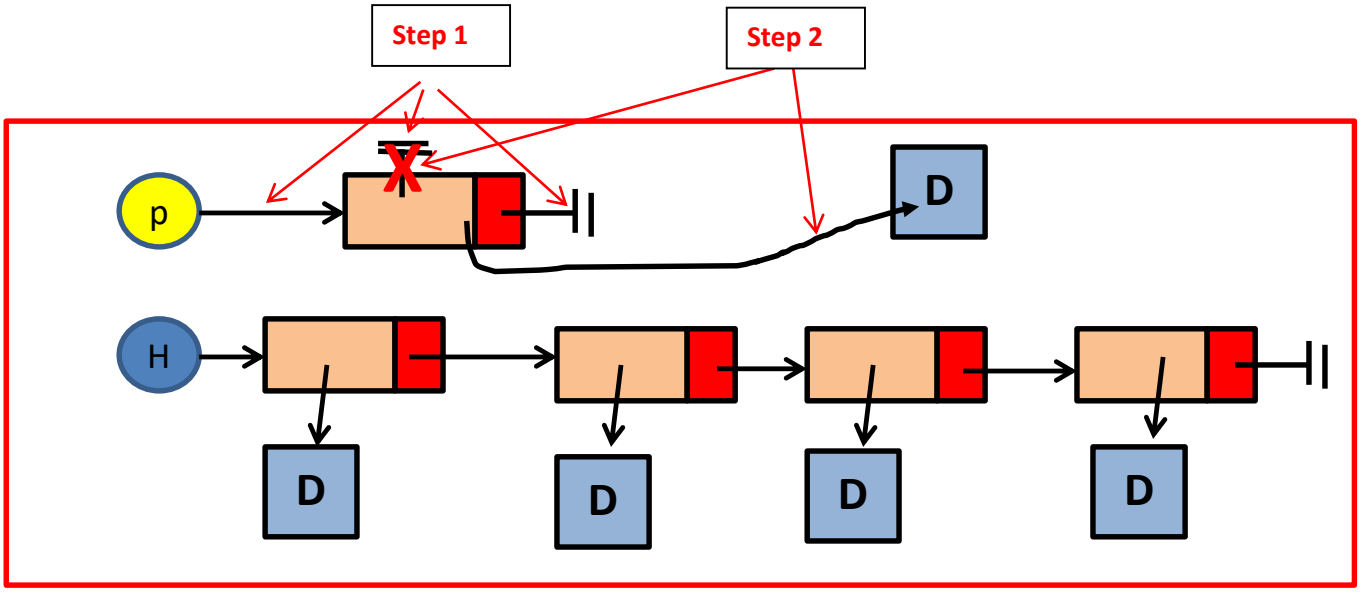
2.2 Task 1

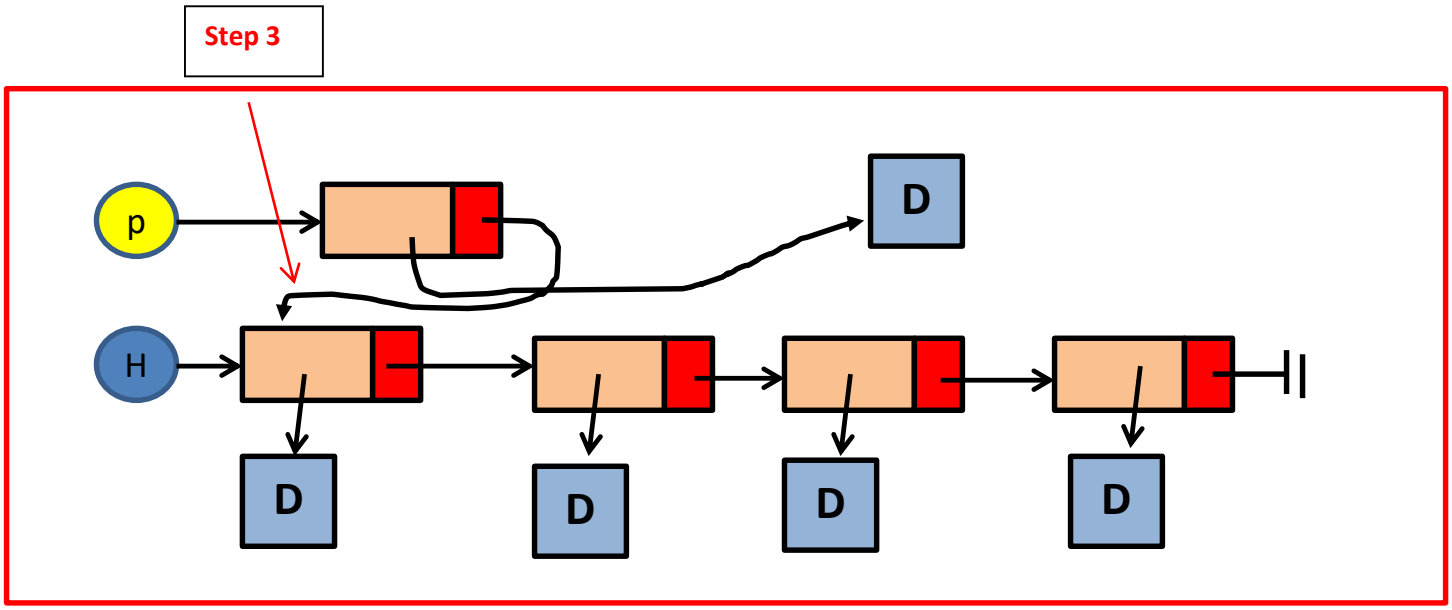
Implement the function addNode(). This function should add the new employee to the linked list as the first node.

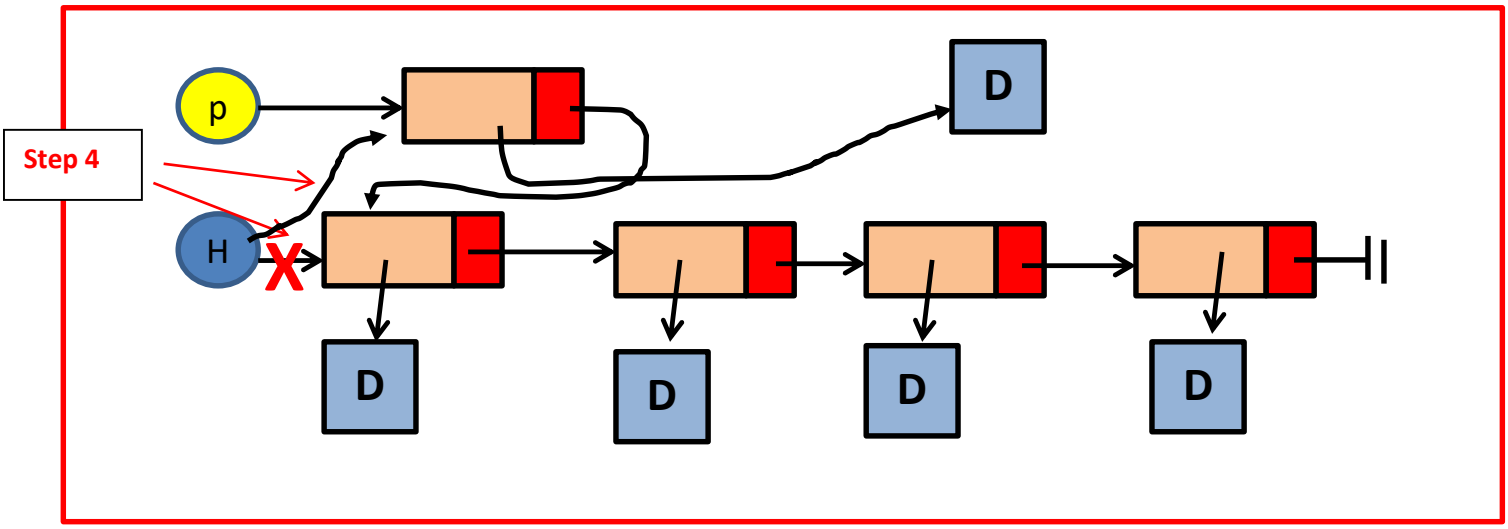
Pseudo code

```
ListNode *p = NULL;

// Step 1 allocate memory for the node and initialize all the pointers to NULL

// Step 2 assign the employee record to data

// Step 3 update the next field of the new node to point to the first node in the
list as a next node

// Step 4 update the list head to point to the new node.
```

Check your code using **valgrind**. Valgrind should show that five blocks of code were used.







2.3 Task 2

Implement the function printList(). This function should print the linked list from the first node to the last node.
This function should be implemented using the iterative approach of traversing the list.

This is a simple “for” loop. The function needs to traverse the list until it reaches the sentinel condition. Here the sentinel is a NULL. The sentinel indicates that end of the list when the pointer “next” is NULL.

Note, that here one can take advantage of the fact that the parameter “head” is a call by value and therefore its value can be changed without any side effects outside the function. Be sure to use the printEmployee function in employee.c

Pseudo code

```
// While list is not empty
        //Print the node using the printEmployee() function
        // advance to the next node
```

2.4 Task 3

Implement the function printListRecursive(). This function should print the linked list from the first element of the list to the last element in the list.
This function should be implemented as a recursive function.

Pseudo code

```
// Check boundary condition (list is not empty)

// Do work - Print the first node using the function printEmployee()

// Call the function recursively with the next node
```

Note, printing the list in reverse order can be easily achieved by changing the order of the "do work" step and the "recursive call" step.
Print the list in reverse order using the function printListInReverse().

2.5 Task 4

Implement the function printThirdLast(). This function should print the node in the linked list that is the third last.
For example: if the linked list consists of H->2->5->4->7->9->1->||| then the function should print 7.

This function should be implemented using the recursive approach of traversing the list. Namely it should be a recursive function.

Test your function using lists of length 0, 1, 2, 3, and 5.

Note that here the boundary conditions are somewhat more complex because one must ensure that the list has at least three nodes. If the list has less than three nodes then the function should return a 1 otherwise the function should return a 0.

Take a few minutes to determine the tests that check if a list has at least three nodes. Check your solution against the one described below. Note that using the ideas of solution below will allow you to determine if the list has exactly three nodes

Pseudo code

```
// If the list has less than three nodes return a 1

// If the list has exactly three nodes  then

        //print the first node using the function printEmployee()

        //return a 0

// else

    // Call the function again with the next node

    //return the result of recursive call
```

Testing if the list has less than three nodes:

1. Check if the list is empty (head is NULL)
2. Check if the list has only one node – check if the "next" field of the first node is NULL (e.g., head->next is NULL)
3. Check if the list has only two nodes – check if the "next" field of the second node is NULL (e.g., head->next->next is NULL)

2.6 Task 5

Implement the function deleteNode(). This function should delete the first node in the linked list

The function should return the employee record that is stored in the node and then delete it.

Notes

1. Here head and data are declared as **. This is because each of them is a pointer and the function must create a side effect outside the function.

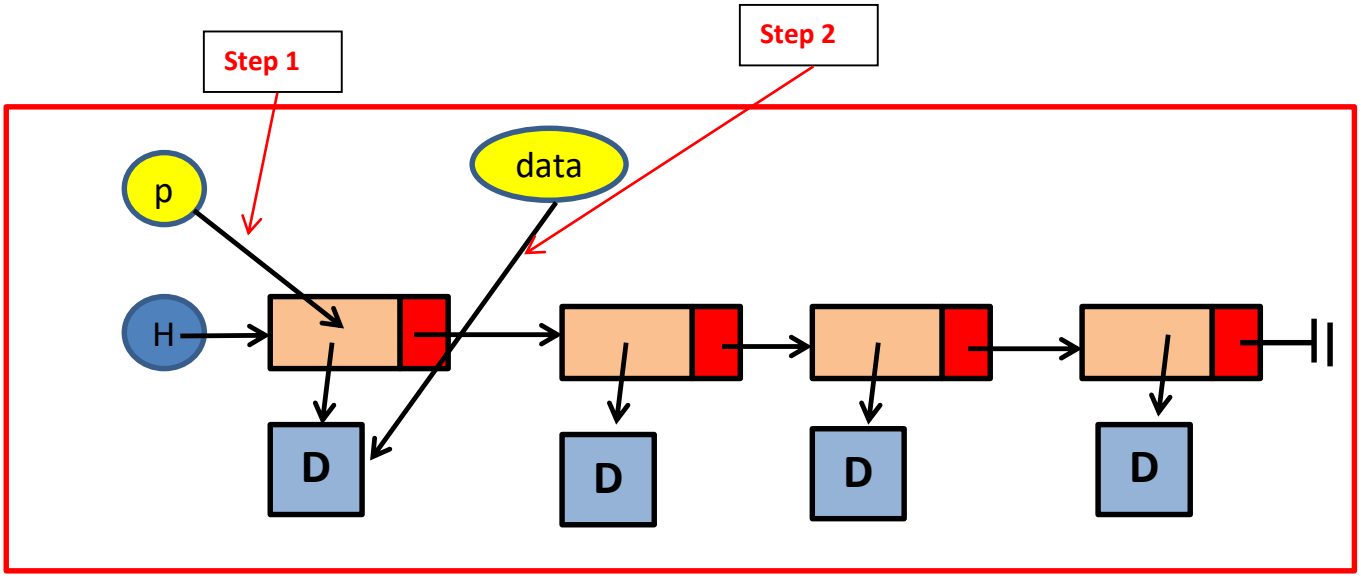
Output – the address of the employee record that was stored in the linked list

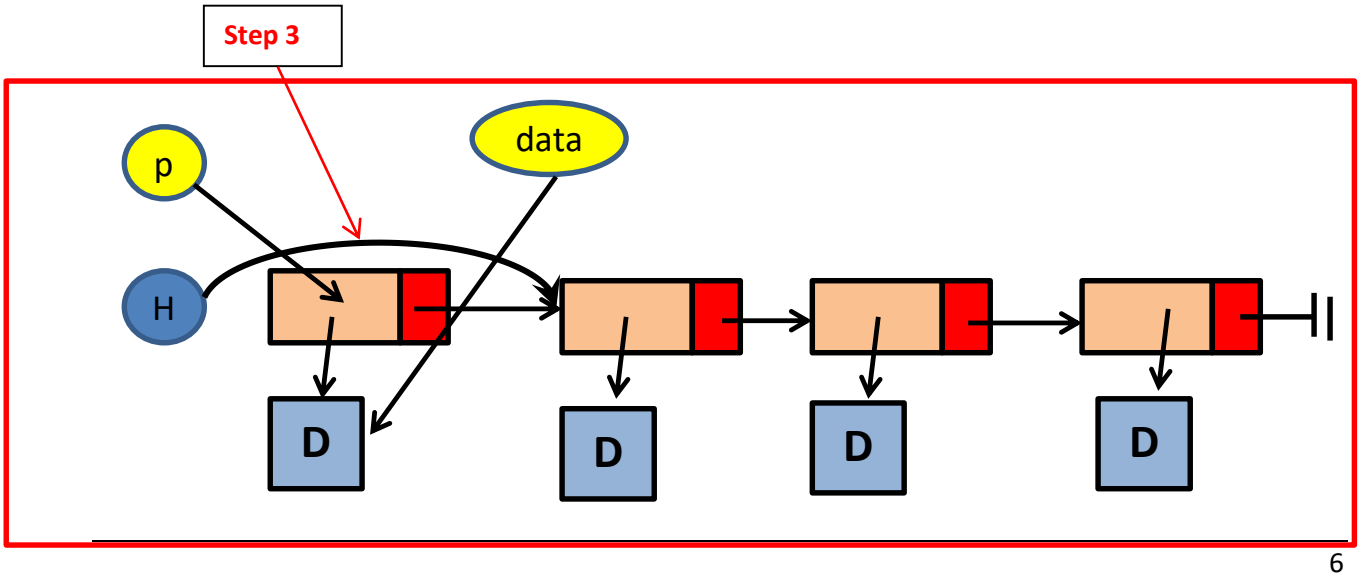
Pseudo code

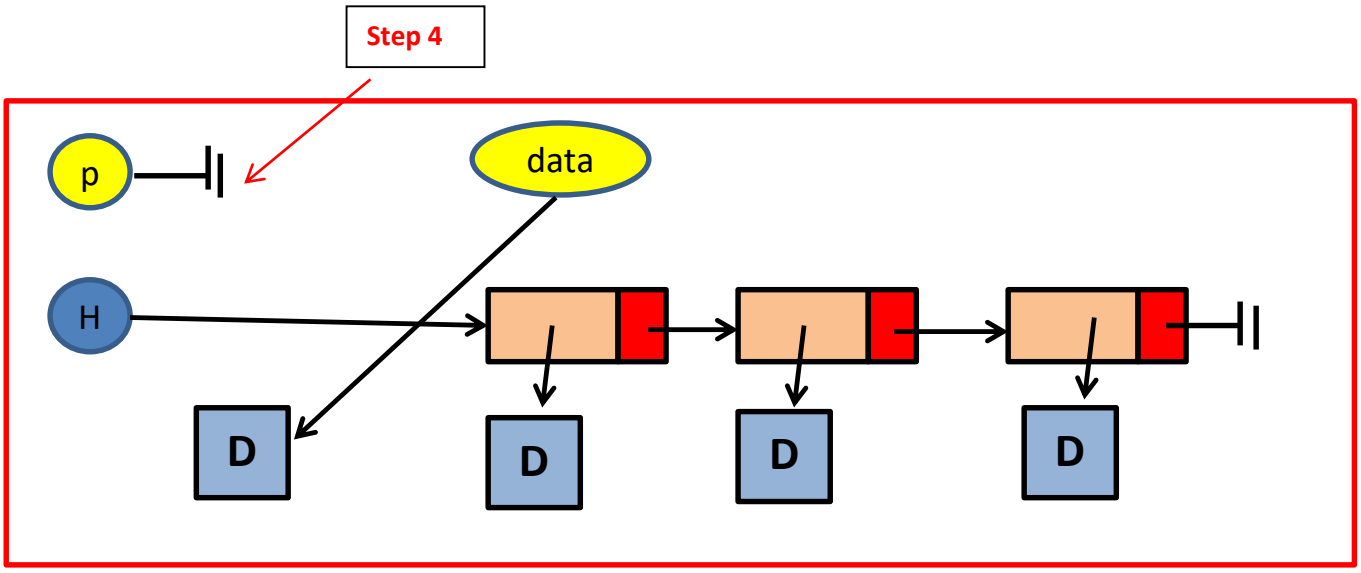
```
ListNode *p = NULL;

// if the linked list is empty then
      // set output data to NULL
      // return

// Step 1 assign to p the address of the first node

// Step 2 assign data in node to the output data
```

```
// Step 3 update the head to point to the next node in the list (head->next)
      // this can be done either by using p as *head = p->next;
      // or by using the head as *head = *head->next

      Can you recall why one needs to use (*head)->next?

// Step 4 free the node pointed to by p and set p to NULL
```

Check your code using **valgrind**. Valgrind should show that no blocks of code where lost







Submit a tar file – **t7.tar** - containing your **myLinkedList.c** and the code you tried from the notes.