Birla Institute of Technology & Science, Pilani
Computer Programming (CS F111)
Second Semester 2015-2016
Lab Sheet # 12 & 13

Objectives

1. Linked List

[Note: You are suppose to do this lab sheet during Week 12 and Week 13. It is advised that you should do up to inserting a node in a Linked List during Week 12, and the rest of the portion during Week 13. Since these concepts are in continuation, corresponding lab sheets are not partitioned.]

---

[1] INTRODUCTION

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

[1.1] Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted. For example, in a system if we maintain a sorted list of IDs in an array id[].

```
id[] = [1000, 1010, 1050, 2000, 2040].
```

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

[1.2] Advantages over arrays

Dynamic size, Ease of insertion/deletion

Drawbacks:

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. Extra memory space for a pointer is required with each element of the list.

[1.3] Representation in C

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL. Each node in a list consists of at least two parts:
(a) data
(b) pointer to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

```
struct node
{
  int data;
  struct node *next;
};
```

[1.4] **<u>First Simple Linked List in C</u>** Let us create a simple linked list with 3 nodes.

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
  int data;
  struct node *next;
};

// Program to create a simple linked list with 3 nodes
int main()
{
  struct node* head = NULL;
  struct node* second = NULL;
  struct node* third = NULL;

  // allocate 3 nodes in the heap
  head = (struct node*)malloc(sizeof(struct node));
  second = (struct node*)malloc(sizeof(struct node));
  third = (struct node*)malloc(sizeof(struct node));

  /* Three blocks have been allocated  dynamically.
     We have pointers to these three blocks as first, second and third
       head           second           third
        |                |               |
        |                |               |
    +---+-----+      +----+----+     +----+----+
    | #  | #  |      | #  | #  |     |  # |  # |
    +---+-----+      +----+----+     +----+----+

   # represents any random value.
   Data is random because we haven't assigned anything yet  */

  head->data = 1; //assign data in first node
  head->next = second; // Link first node with the second node

  /* data has been assigned to data part of first block (block
    pointed by head).  And next pointer of first block points to
    second.  So they both are linked.

       head          second         third
        |              |              |
        |              |              |
    +---+---+     +----+----+     +-----+----+
    | 1  | o----->| #  | #  |     |  #  | #  |
    +---+---+     +----+----+     +-----+----+
  */

  second->data = 2; //assign data to second node
  second->next = third;

  /* data has been assigned to data part of second block (block pointed by
     second). And next pointer of the second block points to third block.
    So all three blocks are linked.

       head          second          third
        |              |               |
        |              |               |
    +---+---+     +---+---+     +----+----+
    | 1  | o----->| 2 | o-----> |  # |  # |
    +---+---+     +---+---+     +----+----+      */

  third->data = 3; //assign data to third node
  third->next = NULL;

  /* data has been assigned to data part of third block (block pointed
    by third). And next pointer of the third block is made NULL to indicate
    that the linked list is terminated here.

     We have the linked list ready.

           head
             |
             |
        +---+---+     +---+---+       +----+------+
        | 1  | o----->|  2  | o-----> |  3 | NULL |
        +---+---+     +---+---+       +----+------+


    Note that only head is sufficient to represent the whole list.  We can
    traverse the complete list by following next pointers.    */

    return 0;
}
```

[2] **Linked List Traversal** In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general purpose function printList() that prints any given list.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
  int data;
  struct node *next;
};
// This function prints contents of linked list starting from the given node
void printList(struct node *n)
{
  while (n != NULL)
  {
     printf(" %d ", n->data);
     n = n->next;
  }
}
int main()
{
  struct node* head = NULL;
  struct node* second = NULL;
  struct node* third = NULL;

  // allocate 3 nodes in the heap
  head  = (struct node*)malloc(sizeof(struct node));
  second = (struct node*)malloc(sizeof(struct node));
  third  = (struct node*)malloc(sizeof(struct node));

  head->data = 1; //assign data in first node
  head->next = second; // Link first node with the second node

  second->data = 2; //assign data to second node
  second->next = third;

  third->data = 3; //assign data to third node
  third->next = NULL;

  printList(head);


  return 0;
}
```

[3] <u>Insertion in a Linked List</u>
A node can be added in three ways
(a) At the front of the linked list
(b) After a given node.
(c) At the end of the linked list.

<u>(a) Add a node at the front (four step process)</u>
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

```
/* Given a reference (pointer to pointer) to the head of a list and an int,
   inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
```

```
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

(b) Add a node after a given node (five step process)
We are given pointer to a node, and the new node is inserted after the given node.

```
/* Given a node prev_node, insert a new node after the given prev_node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
      printf("the given previous node cannot be NULL");
      return;
    }

    /* 2. allocate new node */
    struct node* new_node =(struct node*) malloc(sizeof(struct node));

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

(c) Add a node at the end (six step process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30. Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

```
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    struct node *last = *head_ref;  /* used in step 5*/

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. This new node is going to be the last node, so make next of it as
          NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```

(d) Driver program to test above functions

```
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  // Insert 6.  So linked list becomes 6->NULL
  append(&head, 6);

  // Insert 7 at the beginning. So linked list becomes 7->6->NULL
  push(&head, 7);

  // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
  push(&head, 1);

  // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
  append(&head, 4);

  // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
  insertAfter(head->next, 8);

  printf("\n Created Linked list is: ");
  printList(head);

  return 0;
}
```

[4] Deletion algorithm is one of the basic algorithm which is required to maintain the list. Basically the deletion algorithm adjusts the pointer of the relevant nodes to make sure that the list remains in the correct order (consistent) after the node is removed.

Consider the diagrammatic representation of the linked list given below which holds a character value:

```
list (start pointer)
 |
 v
---------     ---------     ---------
| a | --+---> | b | --+---> | c | --+---> NULL
---------     ---------     ---------
info  next    info  next    info  next
```

The diagram above shows the final form of the linked list. The task is to remove a node form the linked list with a given value X (which can be a parameter to the function delete).

This function should be called as:

ListDelete (&list, X); // address of the list and value of the node must be passed.

We'll consider our elements and nodes to have the following types:

```
typedef char elementT;

typedef struct nodeTag {
  elementT element;
  struct nodeTag *next;
} nodeT;
```

and thus, the pointer to the beginning of the list will be: `nodeT *list;`

Different cases:
There are a few steps to deleting a specific element from the list:
1. Find the node with the element (if it exists).
2. Remove that node.
3. Reconnect the linked list.
4. Update the link to the beginning (if necessary).

Finding the node in question is a matter of traversing the list and looking at each node's element. Reconnecting the list once a node is to be removed is more interesting. Let's consider at least 3 cases:
- Removing a node from the *beginning*.
- Removing a node from the *middle*.
- Removing a node from the *end*.

Removing from the beginning

When removing the node at the beginning of the list, there is no relinking of nodes to be performed, since the first node has no preceding node. For example, removing node with **a**:

```
list
 |
 v
---------     ---------     ---------
| a | --+---> | b | --+---> | c | 0 |
---------     ---------     ---------
```

However, we must fix the pointer to the beginning of the list:

```
list
 |
 +-------------+
               |
               v
---------     ---------     ---------
| a | --+---> | b | --+---> | c | 0 |
---------     ---------     ---------
```

Removing from the middle

Removing a node from the middle requires that the preceding node *skips over* the node being removed. For example, removing the node with **b**:

```
list
 |
 v
---------     ---------     ---------
| a | --+--+  | b | --+---> | c | 0 |
---------  |  ---------     ---------
           |                ^
           +----------------+
```

This means that we need some way to refer to *the node **before** the one we want to remove*.

Removing from the end

Removing a node from the end requires that the preceding node becomes the new end of the list (i.e., points to nothing after it). For example, removing the node with **c**:

```
list
 |
 v
---------     ---------     ---------
| a | --+---> | b | 0 |     | c | 0 |
---------     ---------     ---------
```

Note that the last two cases (middle and end) can be combined by saying that *"the node preceding the one to be removed must point where the one to be removed does."*

In addition, since we need to fix the pointer to the beginning of the list, we'll need some way to get the new beginning pointer out of the function. One way to do so (and the one we'll initially use) is to return the pointer from the deletion function. This means we will have a deletion function as follows:

```
nodeT *ListDelete(nodeT *list, elementT value);
```

and use it as:

```
list = ListDelete(list, value);
```

Implementation:

Again, we'll need to be able to change the pointer to the list, i.e., *list*, in the case that the first node in the list is removed. However, instead of passing back a new value for the beginning of the list via the return mechanism, as in:

```
list = ListDelete(list, value);
```

we'll pass in the pointer to the beginning of the list *by reference* (as a pointer to a pointer), so that we can change it in `ListDelete()` when necessary. Thus, the prototype for our function will be:

```
void ListDelete(nodeT **listP, elementT value);
```

Note: This time, we pass the *address* of the "list" variable to the function, so it makes sense to call the parameter that receives that address "listP", since it is a *pointer* to a "list".

Now we will iterate (i.e., loop) through the list. It's easy to see how to start a pointer at the beginning and move it from one node to the next:

```
---------     ---------     ---------
| x | --+---> | y | --+---> | z | --+--->
---------     ---------     ---------
  ^
  |
 currP
```

In other words:

- Start at the beginning:
- `currP = *listP`
  Note that to access the address of the first node, we have to *dereference* `listP` with the star (*) since `listP` is a "pointer to a pointer to the first node" (and we want to remove one level of pointer-ness to get the "pointer to the first node").
- Advance to the next node when necessary:
- `currP = currP->next`
- Stop when there are no more nodes, i.e., make sure that:
- `currP != NULL`

It's easy to combine these into a `for` loop:

```
for (currP = *listP; currP != NULL; currP = currP->next)
```

However, when we find the one to remove, we'll also need a pointer to the previous node:

```
---------      ---------      ---------
| x | --+---> | y | --+---> | z | --+--->
---------      ---------      ---------
  ^             ^
  |             |
 prevP        currP
```

Thus, we also need to maintain a previous pointer at each step of the loop:

```
for (currP = *listP; currP != NULL; prevP = currP, currP = currP->next)
```

(Notice that there are 2 *increments* separated by a comma.)

To complete the function, we'll have to:

- Have some way to indicate when there is no *previous* node. Remember that removing the first node is a special case because...
  - It requires no relinking of nodes.
  - We have to update the pointer to the beginning of the list. *We don't have to return the new pointer to the beginning (in case it was changed) since we have direct access to it in the function (via `listP`).*
- Relink the list (i.e., skip over the node to be deleted).
- Remove the node.

One implementation of this function might be:

```
void ListDelete(nodeT **listP, elementT value) {
  nodeT *currP, *prevP;

  // For 1st node, indicate there is no previous
  prevP = NULL;

  // Visit each node, maintaining a pointer to the previous node visited
  for (currP = *listP; currP != NULL; prevP = currP, currP = currP->next) {
     if (currP->element == value) {  // Found it
       if (prevP == NULL) { // Fix beginning pointer
           *listP = currP->next;
       } else {
          // Fix previous node's next to skip over the removed node
           prevP->next = currP->next;
       }

       // Deallocate the node
         free(currP);

       // Done searching
         return;
     }
   }
}
```

We do handle the situation of removing the first node, i.e., when there is no previous. See how we use a previous pointer value of `NULL` to indicate this and do the right thing.

Case study for you to implement:

Now that you have learnt about the Linked List, consider the following case and implement the functions that follow. *Try it during the lab (if time permits) or at home, but do try it*. This will clear most of the concept about Linked List.

ABC ltd. manufactures teddy-bears (*henceforth mentioned as TB*) of different varieties, where each TB is of same dimensions. Periodically, the company keeps on adding new such varieties. To showcase its TB's to the customers, company maintains a small show-room, where only one sample of each variety of TB is showcased. The show-room is so small that often it cannot accommodate the sample of all the varieties.
The company also maintains the popularity index of each variety of TB. For it, a list of all the TB's currently showcased is maintained. The position of TB in the list denotes the popularity index of corresponding TB; i.e. popularity of first TB in the list > popularity of second TB in the list > popularity of third TB in the list, and so on. This popularity index gets assigned/changes as per following cases:
Case 1: As soon as customer places an order for a TB (say A): The position of this TB in the list is changed so that it becomes the first one in the list.
Case 2: Company launches a new variety: Following two cases occur here:
Case 2a: If there is a space in the showroom, then this TB is showcased and it takes the first place in the list.
Case 2b: If there is no space in the showroom, then the TB with least popularity index is deleted from the list; it is removed from the showroom; the newly launched TB takes the first position in the list; and it is showcased in the showroom.
Assuming that list of TB's in the showroom is maintained as a linked list, following definitions maintain the information about the showcased TB's:

#define showcase_size 10
//Number of TB's which can be accommodated in the showcase.
 	
typedef enum
{PINK=5,BROWN,WHITE} color;
// Three colors of TB's
	typedef struct
	{
	int bear_id;
	float bear_cost;
	color bear_color;
	} teddy_bear;
	//Information about a TB
		typedef struct node
		{
		teddy_bear tb;
		struct node *next;
		} NODE;
		typedef struct node *Link; // Node of a Linked List
			typedef struct
			{
			NODE *first;
			int size;
			} LIST;
			// Represent a Linked List

Implement the following functions. We are not providing function prototypes; you should decide on them.

1) Function to create an empty linked list, i.e. which returns an instance of LIST such that first points to NULL and size = 0.

2) Function which takes the information about a TB from the user and returns the corresponding structure (i.e. teddy_bear).

3) Function to which LIST is passed as an argument and it prints the three properties (i.e. bear_id, bear_cost, and bear_color) of all the nodes of the linked list.

4) Function to which LIST and a teddy_bear structure is passed as argument. This function inserts the teddy_bear as a first node of the list.

5) Function to which LIST is passed as an argument and it deletes the last node from the list.

6) Function to which LIST and a newly launched teddy_bear structure is passed as argument. This function inserts the LIST as per rules specified in the question. Here, make sure you use the functions defined in (4) and (5) above.

7) Function to which LIST and color or TB is passed as an argument and it counts and returns the number of TB's in the list with specified color.

8) Function to which LIST and a teddy_bear structure is passed as argument. This function inserts the teddy_bear in sorted order as per TB color of the list. You can assume here the order of colors.

9) Function to which LIST and a teddy_bear structure is passed as argument. This function deletes the teddy bear (if it exists) from the list. Teddy bear is compared on all its three attributes.

TRY these questions after completing Week 13.

1) A file stores the record of all the TB's which company has manufactured till date. Each record contains three fields (i.e. bear_id, bear_cost, and bear_color), with one record per line. The first entry of the file indicates the total number of records (an integer) present in the file. Write a function which function which reads these records and stores them in an array of teddy_bear structure. Two arguments are passed to it: name of file, and a pointer nor. Initially, this pointer points to some garbage value. Within the function, the number of records in a file gets stored at the memory location pointed by nor. The function returns the base address of an array of teddy_bear structure.

2) Assuming that showroom is not empty, write a function to print the colors of TB's not showcased in the showroom and returns number of such TB's. The input arguments passed to it are array of teddy_bear structures (populated in Q1 above), its size (nor) and the linked list of TB's in the showroom.

3) Write a recursive function, named printList. Using recursion, this function prints the bear_id of each TB in the list in reverse order, i.e. bear id of last node, followed by bear id of second last node, and so on. Input argument head is the pointer to the first node of the list.