Single Linked Lists problems covered in ExamQuiz

Updated 9/12/2023

You will have a random problem (function to write) from the ones listed below in the quiz. You need to write correct C code for it.

Since the problems are predefined, there will be very little partial credit for incorrect solutions.

For all the functions below, you must use the SINGLE linked list representation from class. The list is a pointer to the first node with data. Such a list, L, can have:

- no node at all. In this case L is NULL
- one or more nodes. L is the address of the first node. That node has useful data in it.

For every function you must:

1. Write proper C code: all variables declared, correct syntax, etc.
2. write the function definition. The function should not crash whatever list it is given (including NULL)
3. implement all the functions you want to use. Do NOT assume certain functions exist and call those (e.g. new_node()). If you want to use such a function, give the definition for it as well. E.g. if you give the definition for new_node() you can then call new_node() however many times you want.
4. Deal with all the special cases so that your function implementation will NOT crash. You should deal with NULL pointers passed to the function and lists being shorter than needed.
5. Whenever you need to insert or delete, adjust the links for the nodes, do NOT copy the data (like for an array).

For these problems we assume a node has the definition we used in class:

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

1. `nodePT list_from_array(int N, int A[N])` - creates and returns a linked list with data from the array, A. You can assume A has N elements and all of those need to be copied into nodes, in the same order as they are in A. E.g. if A = [5, 2, 4, 9] the new list will be 5->2->4->9.
2. `nodePT list_from_list(nodePT L)` - creates and returns a new linked list that holds a copy of the data from linked list, L. The data should be in the same order. E.g. if L is 6->1->9->0, the new list will be 6->1->9->0. You cannot use a second array to do this.
3. ~~`void`~~ **`nodePT destroy_list(nodePT L)`** - delete the list (and free all the nodes from it). Return `NULL` .
4. `nodePT reverse(nodePT L)` – reverse a list. Do NOT use an array in any way. Do not use a copy list. If L is 3->1->7 after the call `reverse(L)`, the nodes will point 7->1->3, and the pointer to node 7 should be returned. You must "move the nodes" by changing the links, NOT by copying the data.
5. `nodePT deleteDuplicates(nodePT L)` – remove duplicates from L. Assume L is sorted in increasing order. E.g. if L is: 1->3->3->5->5->5->6->8->8 after `removeDuplicates(L)`, L will be 1->3->5->6->8. Adjust the links. Do not copy the data content. ALL nodes that were removed, MUST BE FREED.
6. `nodePT deleteByValue(nodePT nd, int val)` remove from the list and free the first node that has value val in it. The function will *return* the list head (the address of the first node in the list). Adjust the links. Do not copy the data content. E.g. if a list, L, has 5->8->20->9->20->7 , `deleteByValue(L, 5)` removes and frees node 5 and returns the address of the node 8.
    a. If there is no node with value val, no node is removed.
    b. If there are two or more nodes with value val, only the 1st one is removed. E.g. if L has 5->8->20->9->20->7 , `deleteByValue(L, 20)` removes and frees the first node with value 20 and returns the address of the node 5 (the list will now be 5->8->9->20->7).

Sample solution:

Let's say you needed to implement a function `nodePT insert_node (nodePT L, nodePT n1, nodePT n2)` that takes two node pointers and it assumes n1 is part of a linked list, L, and it inserts/links n2 after n1.

```
nodePT insert_node(nodePT L, nodePT prevP, nodePT newP){
```

/ The code that goes here is part of the function that I have to implement so I cannot assume any other functions on linked lists exist and take advantage of them here. I have to write that code myself. I cannot use new_node(), array_2_list() or insert_node_after() here. If I want to use insert_node_after(), I should include the definition for that as well. E.g.: */*

```
  if (prev == NULL) {  // inserts at the begining of the list L
         newP->next = L;
         return newP;  // return address of the new first node
  }
  else {
         insert_node_after(prev, newP); // does not affect the list head
         return L;
  }
}
```

And I also give the code for `insert_node_after()`:

```
void insert_node_after(nodePT prev, nodePT newP) {
    if (prev == NULL) {
        printf("\n Cannot insert after a NULL node. No action taken.");
    } else {
        newP->next = prev->next;
        prev->next = newP;
    }
}
```