Faculty of Computers and Artificial Intelligence
CS214: Data Structures

Sheet #3 – Linked List

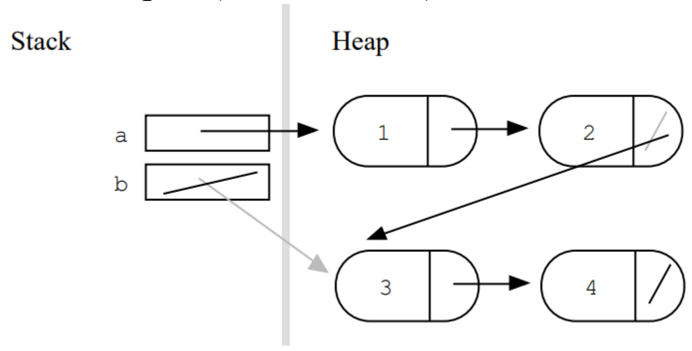
1. Write a Count() function that counts the number of times a given int occurs in a list. The code for this has the classic list traversal structure as demonstrated in Length().

```
void CountTest() {
   List myList = BuildOneTwoThree();      // build {1, 2, 3}

   int count = Count(myList, 2);    // returns 1 since there's 1 '2' in the list
}
```

2. Implement the List ADT using an array (not linked)

3. Write the function **void JoinList (List *pl1, List *pl2)** that copies all entries from pl1 onto the end of pl2. (In both levels).



4. Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position. GetNth() uses the C numbering convention that the first node is index 0, the second is index 1, ... and so on. So for the list {42, 13, 666} GetNth() with index 1 should return 13. The index should be in the range [0..length□1]. If it is not,

GetNth() should assert() fail (or you could implement some other error case strategy).

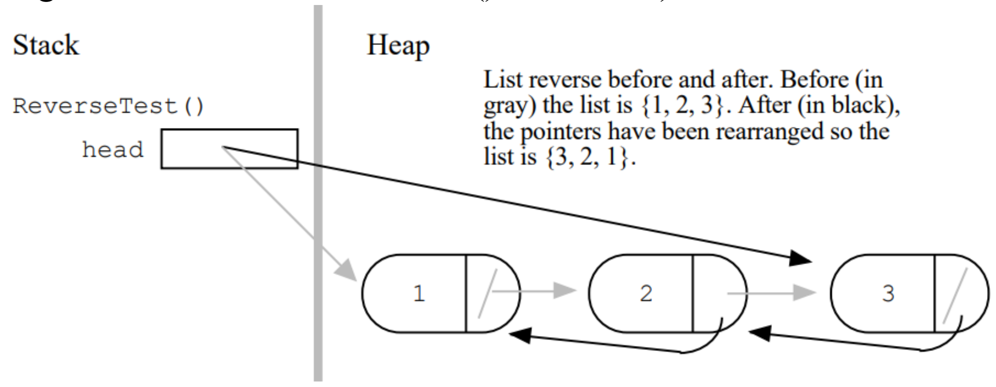
```
void GetNthTest() {
   struct node* myList = BuildOneTwoThree();    // build {1, 2, 3}
   int lastNode = GetNth(myList, 2);            // returns the value 3
}
```

5. Write a Pop() function that is the inverse of Push(). Pop() takes a non-empty list, deletes the head node, and returns the head node's data. If all you ever used were Push() and Pop(), then our linked list would really look like a stack. However, we provide more general functions like GetNth() which what make our linked list more than just a stack. Pop() should assert() fail if there is not a node to pop. Here's some sample code which calls Pop()....

```
void PopTest() {
   struct node* head = BuildOneTwoThree();      // build {1, 2, 3}
   int a = Pop(&head);  // deletes "1" node and returns 1
   int b = Pop(&head);  // deletes "2" node and returns 2
   int c = Pop(&head);  // deletes "3" node and returns 3
   int len = Length(head);          // the list is now empty, so len == 0
}
```

6. Think of the list ADT modified using the following strategy. Whenever an element is located using the isPresent() operation, that particular element is deleted from the current position and reinserted at the beginning of the list. The motivation behind this relocation is that in many situations an element accessed in a list is expected with high probability to be accessed several times in the future. So, keeping the element near the beginning of the list reduces average search time. Modify the list ADT implementations to incorporate this modification.
7. It is required to keep the data of employees in a general list. Write the type definition Entry such that each employee will have the following data (Name -Home Address -Date of Birth -Company (Name, Address, Phone number)), Note that, any date should include day, month, and year; any address should include street, city, and zip code
8. Write a RemoveDuplicates() function which takes a list sorted in increasing order and deletes any duplicate nodes from the list. Ideally, the list should only be traversed once.
9. Write an iterative Reverse() function that reverses a list by rearranging all the .next pointers and the head pointer. Ideally, Reverse() should only need to make one pass of the list. The iterative solution is moderately complex. It's not so difficult that it needs to be this late in the document, but it goes here so it can be next to #18 Recursive Reverse which is quite tricky. The

efficient recursive solution is quite complex (see next problem). (A memory drawing and some hints for Reverse() are below.)


Stack
ReverseTest()
head
Heap
List reverse before and after. Before (in gray) the list is {1, 2, 3}. After (in black), the pointers have been rearranged so the list is {3, 2, 1}.
1
2
3