(1) Try on paper:

Would this work? (hint: no!!!)

SNode \*n = new SNode(9);

node4->next = n;

n->next = node4->next;

Why not??

**Answer:** this would not work because the second line states that the next node after node4 is n, which in doing this, node4 now points to n (node4 is no longer pointing to the next node—rather the new node you created). And the third line sets node n’s next property to node4’s next property, which is n. Thus n’s next property is itself and the list cuts off the node in the list that is supposed to go after node n

Line 1:



Node n:



Line 2: node3:



Line 3:



(2) Try on Paper:

a. Define the terms (on page 1)

**ADT:** ADT stands for Abstract Data Type. The ADT should describe what the data type is and what it should do. You can have an abstract concept of a data structure and then be able to implement the concept in various ways.

**List:** in a List, all variables/objects have the same data type, are in order (doesn’t mean the order has any significance), a known size, and a data can be duplicated in the list



**Push:** this command places the data onto the adding an item to the end of a list

**Pop:** removes the value from the end of the list and returning it.

**Stack:** a part of the computer’s memory where the compiler piles each functions variables and parameters until they go out of scope.

**Arrays:** all arrays are of just one type, one item occurs after another in a list, has a known size, and allows data to be duplicated

**Time Analysis:** how efficient code is able to run “behind the scenes”. And example is arrayList in java. When the arrayList is filled, there is a new array that is created and copies all of the data of the old array into the new, bigger array. This concept works, but is not efficient. So in Time Analysis, we say it takes o(n) with arrayList.

**Linked List:** the data in a linked list is not located sequentially in memory, no fixed size, also every data element in a linked list has the data itself and a pointer to another data element.

**Friend:** when a class makes another class its friend, that other class now has access to its private and protected properties and methods within the class.

**Kluge:** when you make a quick and dirty solution that is clumsy, inefficient, difficult and hard to understand

**b.)** Try on paper:

Given the following code, and the following linked list, if you run the method func4 with this linked list, what would

be the resulting list?

a->k->b->o->t->a->h->l->v->a->

class SNode {

friend class SLL;

char c; // as opposed to int data;

SNode \*next;

};

void SLL::func4() {

SNode \*tmp = first->next;

delete first;

first = tmp;

while ((tmp->next != NULL)&&(tmp->next->next != NULL)) {

Snode \*t2 = tmp->next;

tmp->next = tmp->next->next;

tmp = tmp->next;

delete t2;

}

if (tmp->next != NULL) {

tmp->next = NULL;

}

last = tmp;

}

**Answer for b:** the returning list is k->o->a->l->a



v

a

l

h

a

t

a

k

b

o



t2

t2



t2

t2

tmp

tmp

tmp

tmp

tmp

tmp=last

tmp=first

**c.**) With the method pop(), you must loop through the entire list each time, even though the linked list class has a pointer to the last node in the list. Why?



**Answer for c:** For a list of n elements, using the pop() method you have to loop through the list to locate the (n-1)th node in the linked list. You set the current last property to a temp node, then you set the next property of the (n-1)th node to NULL and then set the last property to the (n-1)th node. From there you can delete the temp variable and the old last node is expunged from memory. If you were to have deleted the last variable straight away that would cause a memory leak because (n-1)’s next would point to wasted space in memory.

**d.)** When inserting into a list in the kth position, why do you loop to position k-1 and not to position k?

**Answer for d:** You need to loop to the (k-1)th position because in order to insert a new node in between k-1 and k you have to adjust k-1’s next property to be the new node, and then set the next property of the new node to k, which in reference to the list is now the k+1th position.

**e.**) Based on the class for a singly linked list, described above, why would writing a method that either reverses the list or traverses the list in reverse order be difficult?

**Answer for e:** the reason it would be difficult to go through a list in reverse is because a every data/node in a linked list points to the next data/node. Since each data points to the next data, its hard to traverse in reverse because the last node is not pointing to any previous node. This fallback is corrected in the doubly linked list with the previous property. Data in a linked list is not sequential in memory, so the only connection to other nodes in the singly linked list is the next property.

(3)Try on paper:

**a.)** Why is it only O(1) to find the kth element in an array?

**Answer for a:** because in order to find the kth element all you have to do is “array[k];” this statement immediately gives you the value in the kth position in the array. This is possible because elements in arrays are SEQUENTIAL in memory, so all you have to do is add element address + k to get the kth element, this is one operation.

**b.)** Why is it O(n) to find out whether x is in the list when implemented either as an array or a linked list?

**Answer for b:** for both array’s and linked list, in order to find out whether x is in the list , you must traverse through the list and check each index in the list to see if the value in them index is equal to x. There is no other way to do this, which is why for time analysis, it has 0(n)