Try on Paper exercises:

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

SNode \*n = new SNode(9);

node4->next = n;

n->next = node4->next;

* Its n’s next would be n

a. Define the terms:

**ADT**- collection of elements/nodes

**List**- an abstract data type that has an order of elements that are of the same datatype

**Push**- to add to the end of a list

**Pop**- remove a value off the end of a list

**Stack**- the memory on which the compiler piles each function’s variables until they go out of scope; last in first out

**Arrays**- an implementation of the ADT list; elements are all one type with an order, has a size and allows duplicates in the data

**Time Analysis**- the amount of time it takes to complete a function based on the number of values in the list

**Linked List**- data is not sequential in memory, so every piece of data has its own address and the address of the next piece of data; no fixed size and no empty spaces between elements

**Friend**- a class can make another its friend, allowing the other class to access its private and protected properties and methods

**Kluge**- code that works but isn’t pretty

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;

}

* The result is **“koala”**

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?

* Because the linked list is in sequence, when you call the linked-list you start at the beginning. pop() removes the last, inorder to reach the last you must start from first and when you get to the lass then you can pop the last

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

* Adding an element at k-1 makes it the kth element , and the element that was previously kth is now k+1

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?

* In a singly linked list, each element only has reference to its own address and the address of the next element



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

* Only one step is involved in finding the kth element

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?

* You have to traverse through the list n times to find x