Setup

We begin by constructing a simple linked list:

```
node* head = new node;
head->data = 14;
head->next = nullptr;
add_to_tail(head, 36);
add_to_tail(head, 42);
add_to_tail(head, 9);
```

Q1 - For

Print out the items in the linked list with a `for` loop.

```
for (                                                    ) {



}
```

Q2 - While

Print out the items in the linked list with a `while` loop.

```
while (                                                  ) {



}
```

Q3 - Recursion

Print out the items in the linked list with a recursive loop.

```
void print(node* n) {



}
```

Q4 - Recursive Reverse

Print out the items in the linked list in reverse order recursively.

```
void printReverse (node* n) {



}
```

CSCI 262 Lecture 5 – Linked Lists, part 2

Outline

- Continue adding methods to our linked list class
  - Focus on operations at front and back of list – enough to implement stacks & queues (next time)
- Deallocating (destroying) dynamically allocated nodes
  - Just the syntax for now – we'll explore what it all means in a future lecture
- Introduction to Big O performance measure

Readings

For today: Chapter 14.1 – 14.2 in your textbook

Note that the textbook implements a much fuller-featured linked list than we did in class!

For Monday: Chapter 14.4 – 14.6 in your textbook

Self Check

1. The _____ operator "destroys" a node. (We'll talk later about how this relates to the program memory.)
2. Big O is a measure of performance that focuses on how the cost of an operation _____ with the size of the input. We typically use the letter N (or n) to represent the size of the input.
3. The two costs we've seen so far are _____ , representing a constant cost (e.g., when adding an element to the front of a linked list); and _____ , representing a cost that is a linear function of the size of the input.

For Further Practice

One use for linked lists can be found in (some) process scheduling algorithms. Modern operating systems give the illusion of having all open programs running simultaneously; in fact, modern processors can only run a handful of programs simultaneously, and the illusion of having everything running is maintained by giving every program a little slice of processor time many times a second (e.g., each process might get 0.1 milliseconds to run, then the next process gets a turn, etc.)

While process schedulers need to keep in mind certain high-priority processes, one scheduling algorithm that works well in general follows a Least Recently Used (LRU) model – the process which has waited the longest gets to go next.

There's an easy way to keep track of all processes and which one goes next in the LRU model: create a *circular* linked list – one where the tail just connects to the head, instead of a nullptr. In this scheme, in fact, there is no notion of head or tail – just a pointer to the current node!

Consider taking the linked_list class code created for the last two lectures and re-working it to implement an LRU data structure. Instead of add_to_front etc., you'll want these operations:

- Get current value
- Insert after current – create a new node that will be next in line when we're done with the current node
- Remove current
- Advance to next
- Print values, starting at current node (tricky - don't loop forever!)