CS 240: Programming in C

Lecture 14: Linked Lists, Doubly-linked Lists

PURDUE UNIVERSITY

Announcements

- Homework 7 released
- Last day to request regrade on Midterm 1

Quick note about malloc

- We use our own malloc() library in this class
  - You'll write your own in CS 252!
- It knows when you **malloc()** and do not **free()**
- It knows when you **free()** more than once

Valgrind

- Valgrind is a suite of tools for debugging and profiling programs
- Very useful for identifying memory leaks and errors

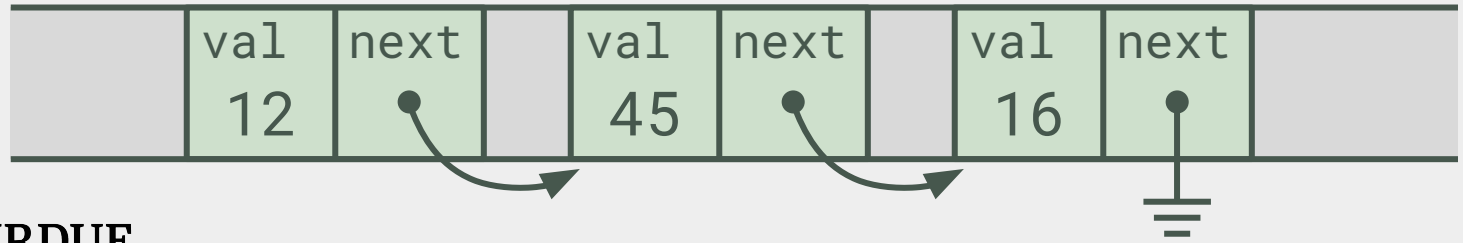
```
$ valgrind ./executable
$ valgrind --leak-check=full ./executable
```

Linked lists

- Consider this structure:

```
struct node {
  int val;
  struct node *next;
};
```

- Create a number of them somewhere in memory
- Let each one point to the next, and the last have a NULL pointer



Linked list creation

- Allocate each node on the heap
  - using malloc()
- Grow the list “forward” or “backward”
  - See last lecture
- Use a head pointer to point to the first node
- Optionally use a tail pointer for the last node

Traversing a linked list

- Usually, you do not know how many nodes are in a linked list
  - Have to “traverse” it to find an item or do work on the structures
- You can traverse a linked list with one extra pointer

```
struct node *p = head;
while (p != NULL) {
  p->val++;
  p = p->next;
}
```

Deleting a linked list

- Deletion of a linked list is a special case of the traversal process

```
p = head;
while (p != NULL) {
  struct node *next = p->next;
  free(p);
  p = next;
}
```

Functions to simplify list mgmt

- Writing code to do operations on lists is
  - Repetitive
  - Tedious
  - Error prone
- It is usually a good idea to encapsulate the functionality into functions to create, delete, insert, and append new structures

Example: create_node()

- Allocate a new node, check the malloc() return value, and set the fields:

```
struct node *create_node(int new_value) {
  struct node *temp = NULL;

  temp = malloc(sizeof(struct node));
  assert(temp != NULL);

  temp->val = new_value;
  temp->next = NULL;

  return temp;
}
```

Bigger “payload”

- Normally a structure in a linked list contains many more elements than just a single value and a list pointer:

```
struct big_node {
  struct big_node *next;
  float height;
  float width;
  float weight;
  int angle;
  float age;
};
```

Linked lists vs. Arrays

Linked list	Array
• Unlimited* capacity	• Fixed capacity
• Sequential access	• Random access
• Overhead per element	• No overhead

Tough questions

- It’s easy to traverse a list from head to tail
  - How about from tail to head?

Tough questions

- It’s easy to traverse a list from head to tail
  - How about from tail to head?
- Can you write a function that will swap a specified node in a linked list with the node that follows it?

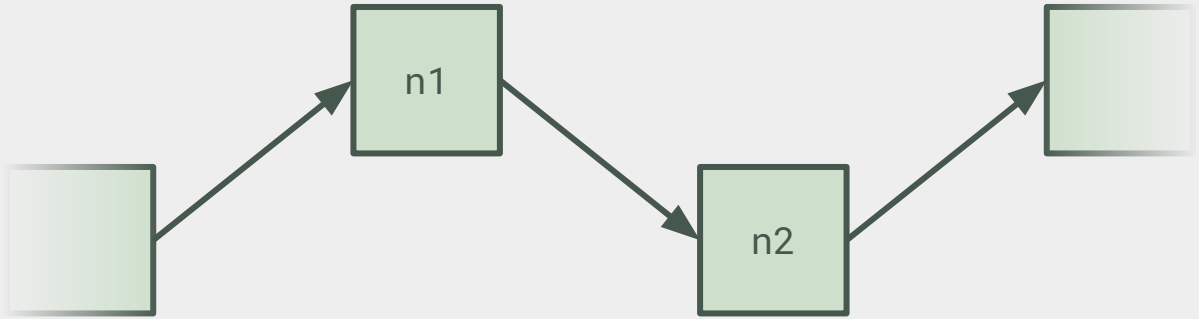
Swap node

```
void swap_node(struct node *n) {
  struct node *n1 = n;
  struct node *n2 = n->next;

  n1->next = n2->next;
  n2->next = n1;
}
```

- Why doesn't this work?

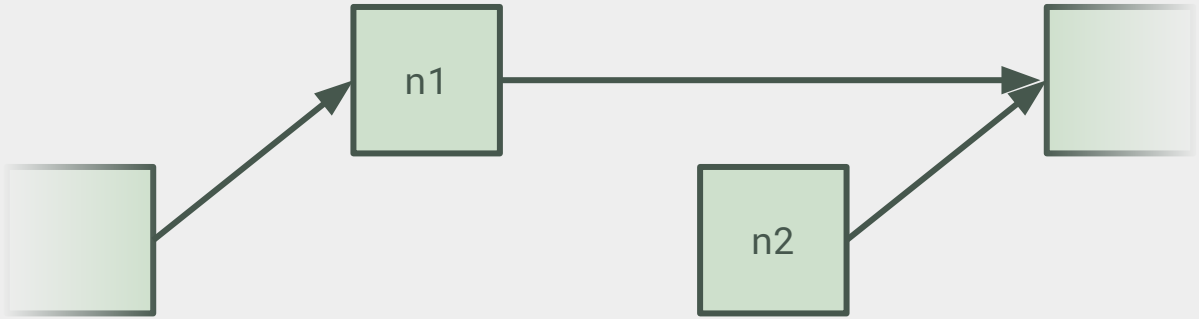
Swap node

```
void swap_node(struct node *n) {
  struct node *n1 = n;
  struct node *n2 = n->next;

  n1->next = n2->next;
  n2->next = n1;
}
```



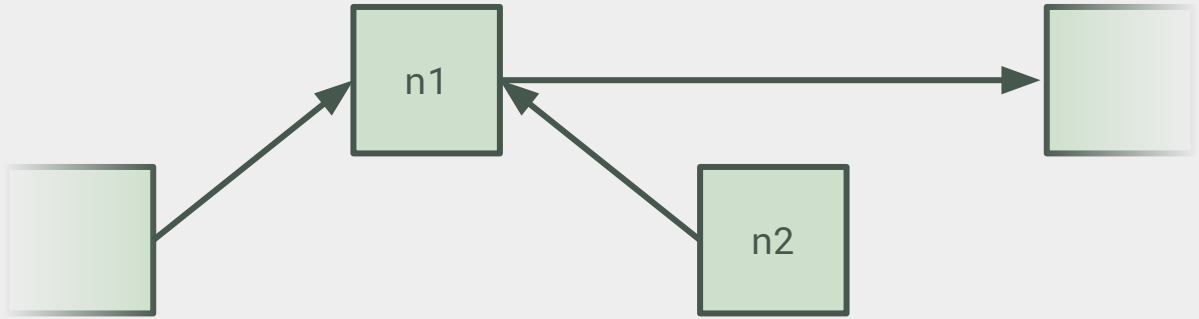
Swap node

```
void swap_node(struct node *n) {
  struct node *n1 = n;
  struct node *n2 = n->next;

  n1->next = n2->next;
  n2->next = n1;
}
```



Swap node

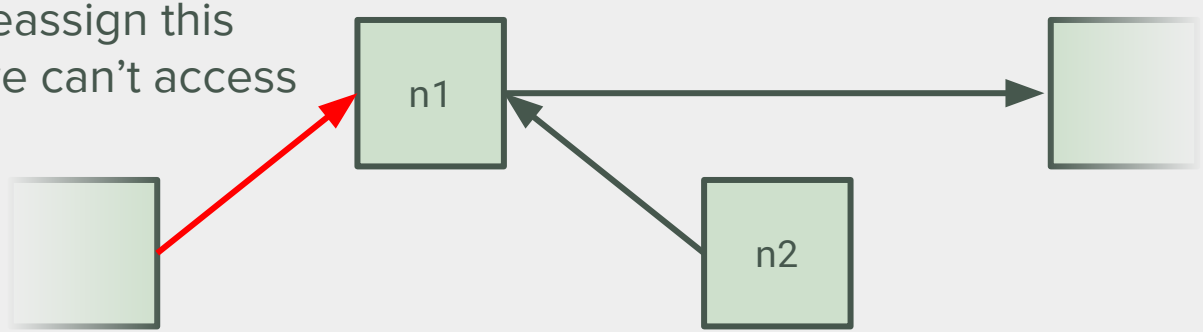
```
void swap_node(struct node *n) {
  struct node *n1 = n;
  struct node *n2 = n->next;

  n1->next = n2->next;
  n2->next = n1;
}
```



Swap node

```
void swap_node(struct node *n) {
  struct node *n1 = n;
  struct node *n2 = n->next;

  n1->next = n2->next;
  n2->next = n1;
}
```

We need to reassign this pointer, but we can't access it directly



Swap node

```
void swap_node(struct node *n, struct node *head) {
  struct node *n1 = n;
  struct node *n2 = n->next;

  n1->next = n2->next;
  n2->next = n1;

  struct node *p = head;
  while (p != NULL && p->next != n1) {
    p = p->next;
  }
  if (p != NULL) {
    p->next = n2;
  }
}
```

Tough questions

- It’s easy to traverse a list from head to tail
  - How about from tail to head?
- Can you write a function that will swap a specified node in a linked list with the node that follows it?
  - Can we do it without referencing the head?

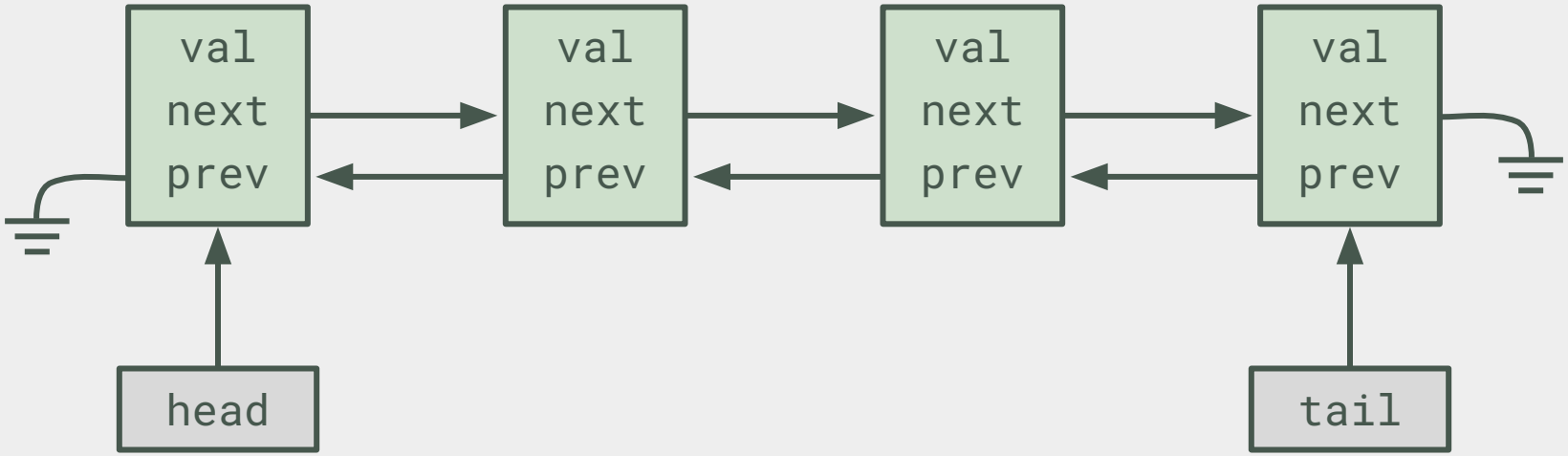
Tough questions

- It’s easy to traverse a list from head to tail
  - How about from tail to head?
- Can you write a function that will swap a specified node in a linked list with the node that follows it?
  - Can we do it without referencing the head?
- Can you write a function that will prepend a node before a given node in the list?
  - Without referencing the head?

Doubly-linked list

- Without the head, the answers to the previous questions are “no”.
- The lists we’ve looked at so far are called singly-linked lists -- each node only points to its next neighbor
- A doubly-linked list contains two pointers:
  - A “next” pointer
  - A “previous” pointer

Example of a doubly-linked list



Example declaration

```
struct dbl_node {
  int val;
  struct dbl_node *next;
  struct dbl_node *prev;
};
```

Creation routine

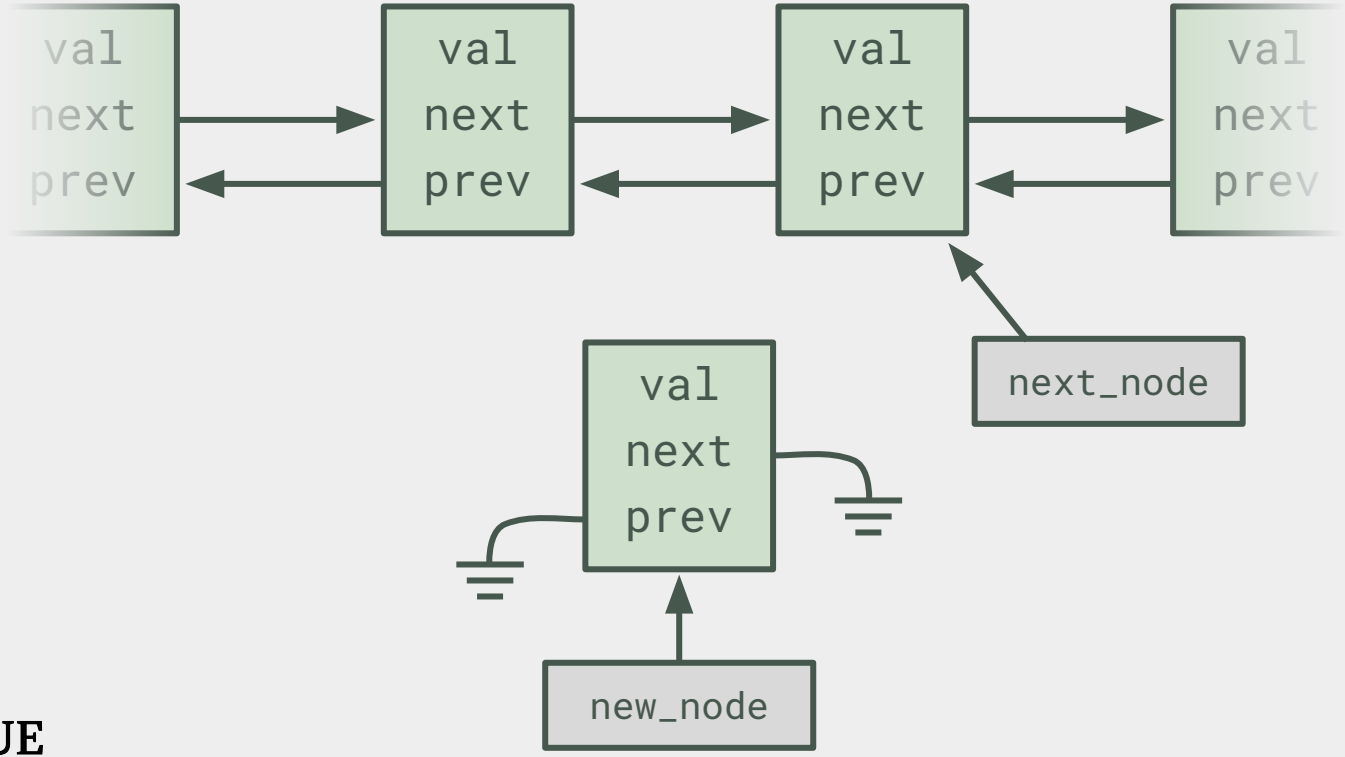
```
struct dbl_node *create(int value) {
  struct dbl_node *temp = NULL;

  temp = malloc(sizeof(struct dbl_node));
  assert(temp != NULL);

  temp->next = NULL;
  temp->prev = NULL;

  temp->val = value;

  return temp;
}
```

Prepend routine

- Insert new_node before next_node in list

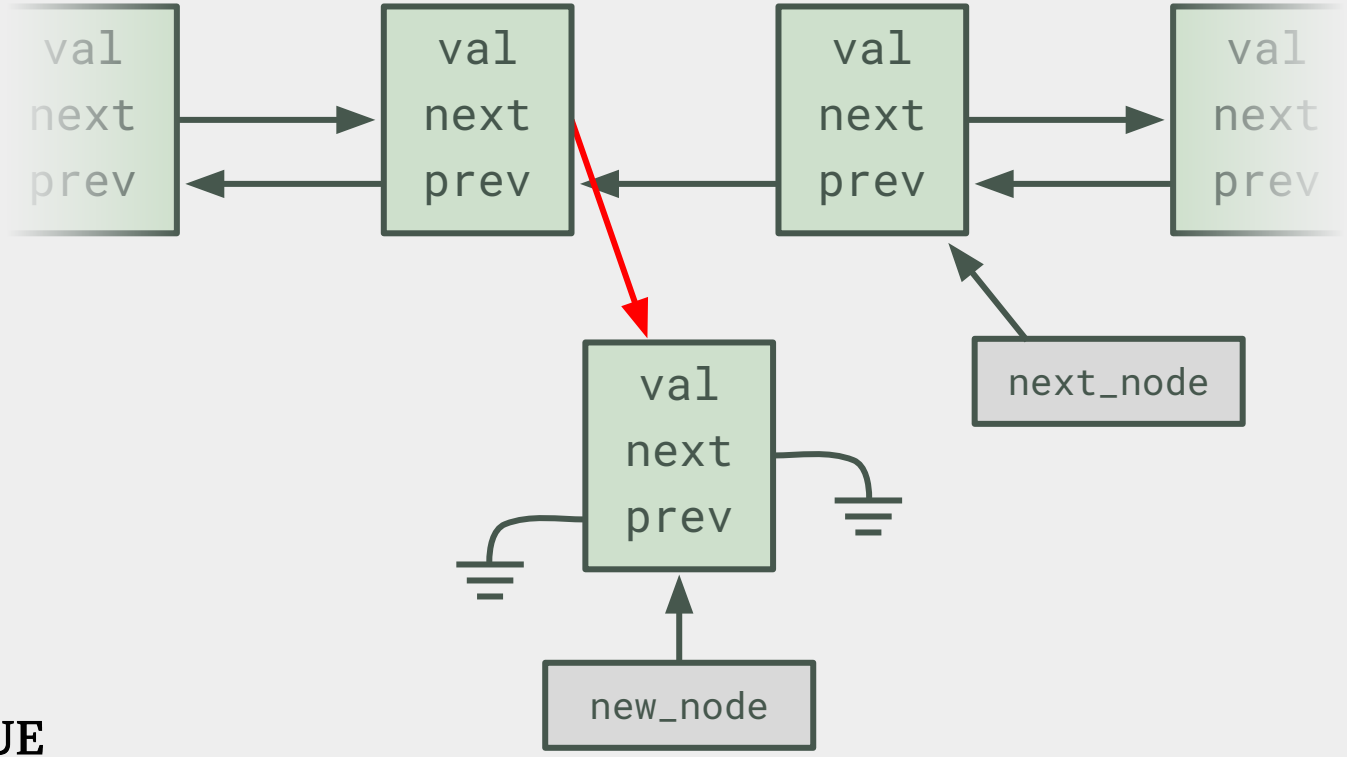
```
void prepend(struct dbl_node *new_node,
             struct dbl_node *next_node) {

  if (next_node->prev != NULL)
    next_node->prev->next = new_node;   1

  new_node->prev = next_node->prev;   2

  new_node->next = next_node;   3

  next_node->prev = new_node;   4
}
```

Prepend (initial setup)



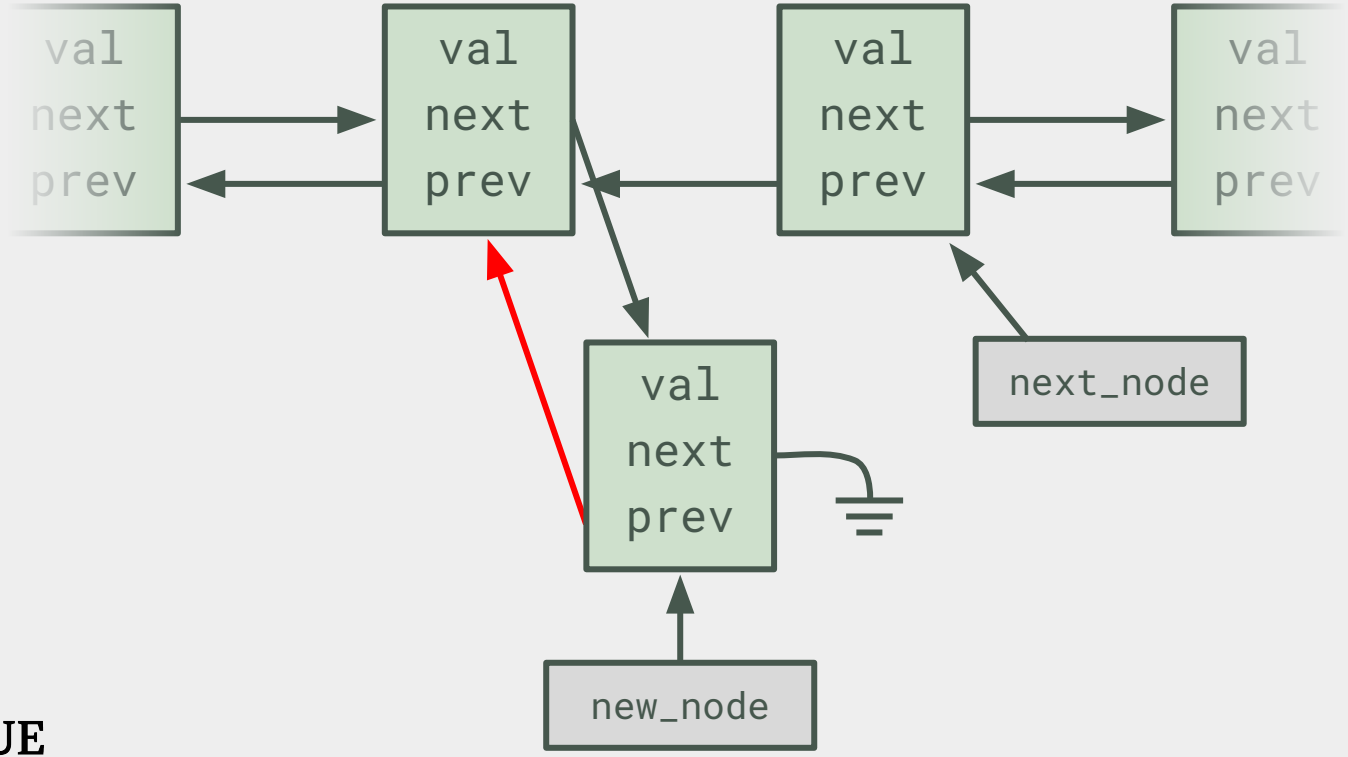
Prepend (step 1)

```
next_node->prev->next = new_node;
```



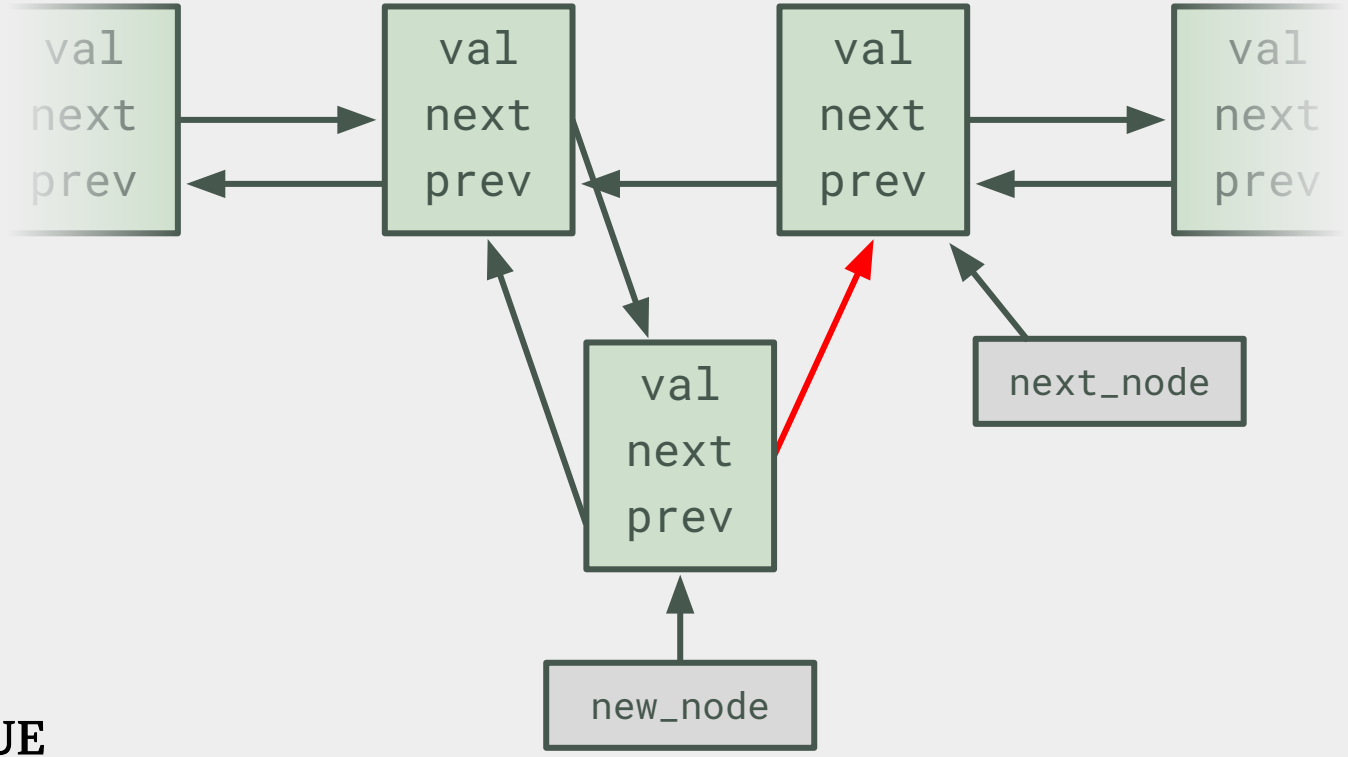
Prepend (step 2)

```
new_node->prev = next_node->prev;
```



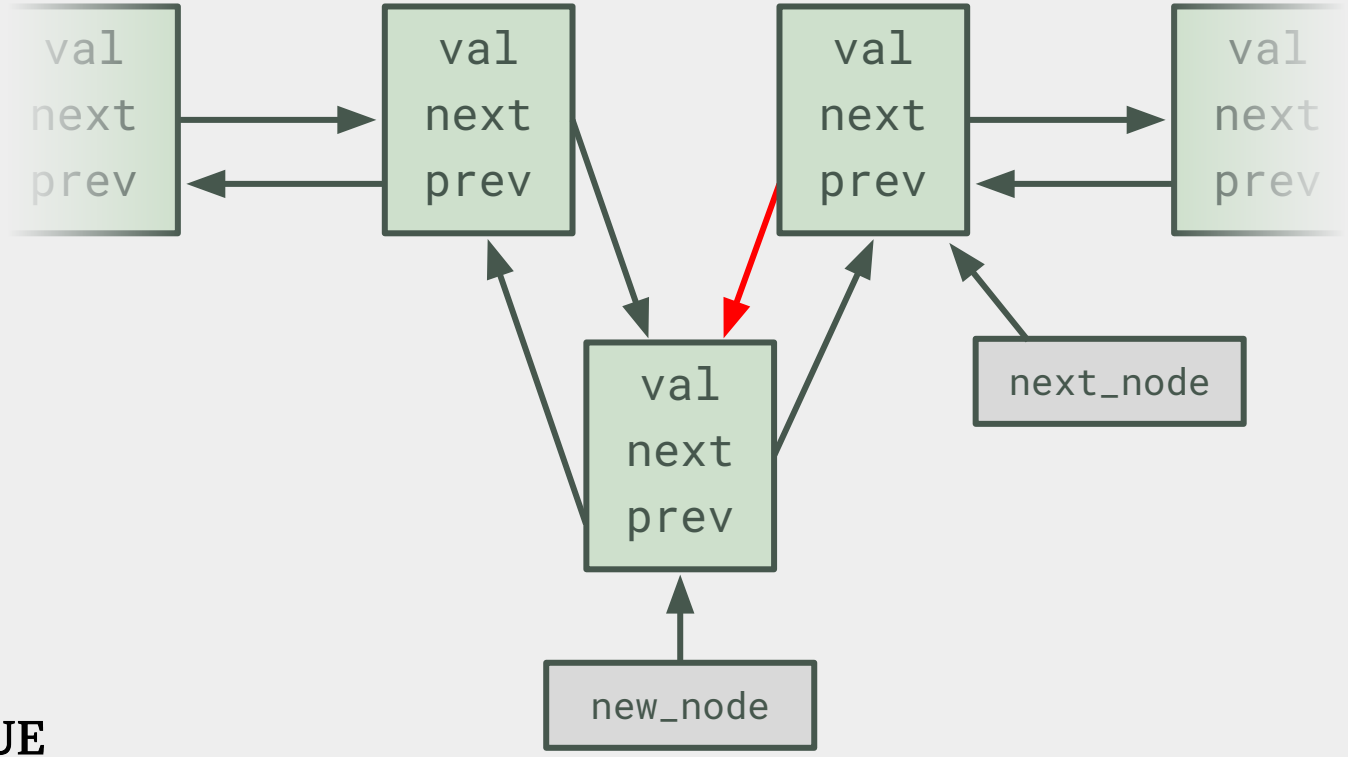
Prepend (step 3)

```
new_node->next = next_node;
```



Prepend (step 4)

```
next_node->prev = new_node;
```



Important points

- There are **four steps**
  - Why?
- It is imperative to put those steps in the right order
  - Some steps are interchangeable, some are not!
- You should practice this on paper

Important points

- There are **four steps**
  - Why? Because there are **four pointers** to reassign
- It is imperative to put those steps in the right order
  - Some steps are interchangeable, some are not!
- You should practice this on paper

Removing an element

- With a doubly-linked list, we can remove an element from anywhere within the list

```
void remove_dbl(struct dbl_node *ptr) {
  if (ptr->next != NULL)
    ptr->next->prev = ptr->prev;

  if (ptr->prev != NULL)
    ptr->prev->next = ptr->next;

  ptr->next = NULL;
  ptr->prev = NULL;
}
```

Takehome Quiz #3

- Write a function to swap two nodes in a doubly-linked list
- The nodes **are not** adjacent!
  - **Extra credit** if your code *also* correctly handles the adjacent case
- Your function should look like this:

```
void swap_dbl_nodes(struct dbl_node *a,
                    struct dbl_node *b);
```

- Consider: how many steps are there?
- Hint: you may need to store temporary variables
- Assume the struct is already declared (slide 25)
- Handwritten answers ONLY
  - No need to use the template anymore
  - Limit your response to one double-sided sheet of paper

For next lecture

- Work on Homework 7!!
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.