COMP1511/1911 Programming Fundamentals

Week 8
Lecture 1

Week 8 Check-Off

- Lab exams this week
  - Online students please register for an in-person one
    - Link to register: https://buytickets.at/comp1511unsw/2401097
    - Please arrive in the second hour of the lab time
  - This is a great chance to get the feel for an exam environment
  - Email course account if for some reason you are sick or can't attend on the day cs1511@unsw.edu.au



Week 8 Check-Off

- Format
  - 1 hour
  - One one dot array question
  - One two dot array question
  - Two debugging questions
- Attempt and submit at least one question to get a check off
  - There are also regular lab exercises to do in the lab

Week 8 Check-Off

- This is a great opportunity to try out the exam environment and attempt questions that will be the same style as the array hurdles and debugging questions in the final exam!

Revision Session

- Thursday Week 8 10-12pm in Bongo Lab G07A K17 (CSE)
  - Tutorial-style revision of course content so far, great chance to catch up!
- Sign up here
  - https://buytickets.at/comp1511unsw/2396767

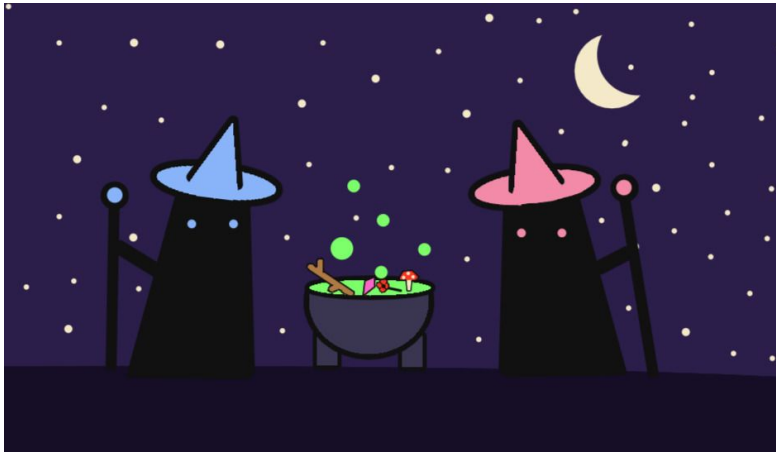


Drop In Sessions

- Chance to get 1-on-1 guidance with anything course related
- Contact cs1511@unsw.edu.au if you want to find out more or book a session

Assignment 2: CS Alchemy

- Assignment: Due Friday Week 10 at 6:00 pm (20/11/2026)
- Watch the Assignment Walkthrough Video to get started!
- Ask questions on the forum and in labs and Help Sessions



Previously On...

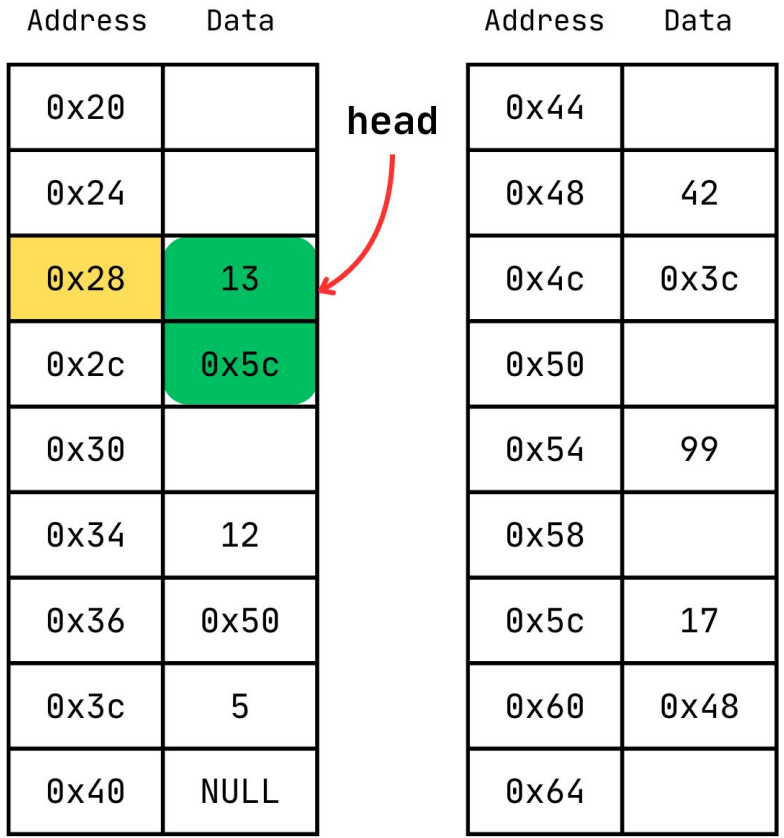
- Linked lists
  - What are they?
  - How do we make them?
  - How do we insert into them?
  - How do we traverse them?

Today

- Recap
  - List basics
  - Creating nodes
  - Printing lists
- Deletion
  - Deleting the first node
  - Freeing the whole list
  - Search and delete
  - Memory leaks
  - Use after free errors

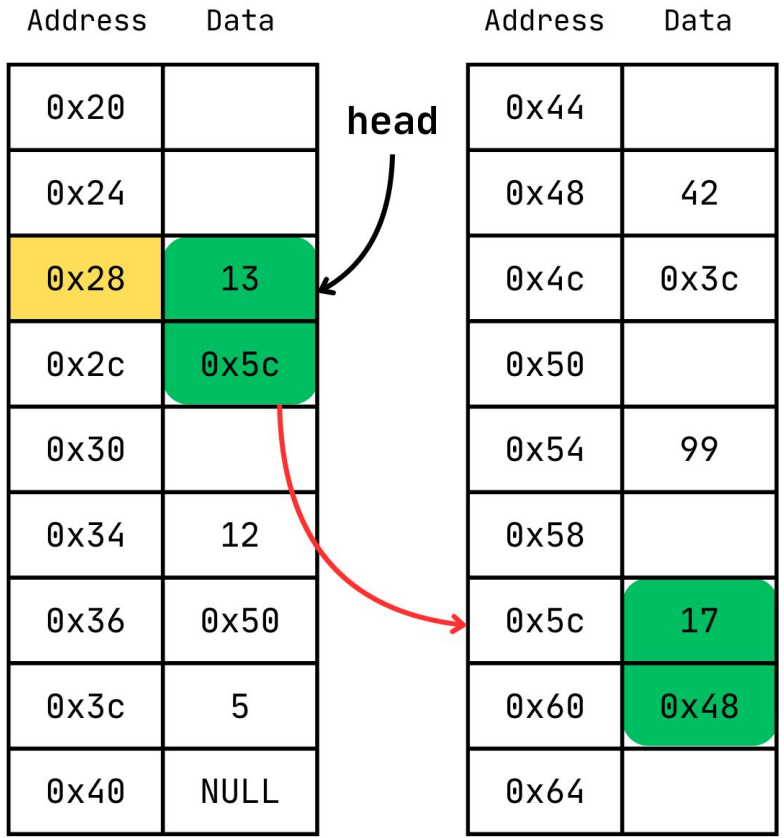
Linked Lists in Memory

- We have a pointer to the first piece of data
- For every piece of data you store in the list, you need to store a link (pointer containing the address) to the next item in the list
  - Like a scavenger hunt



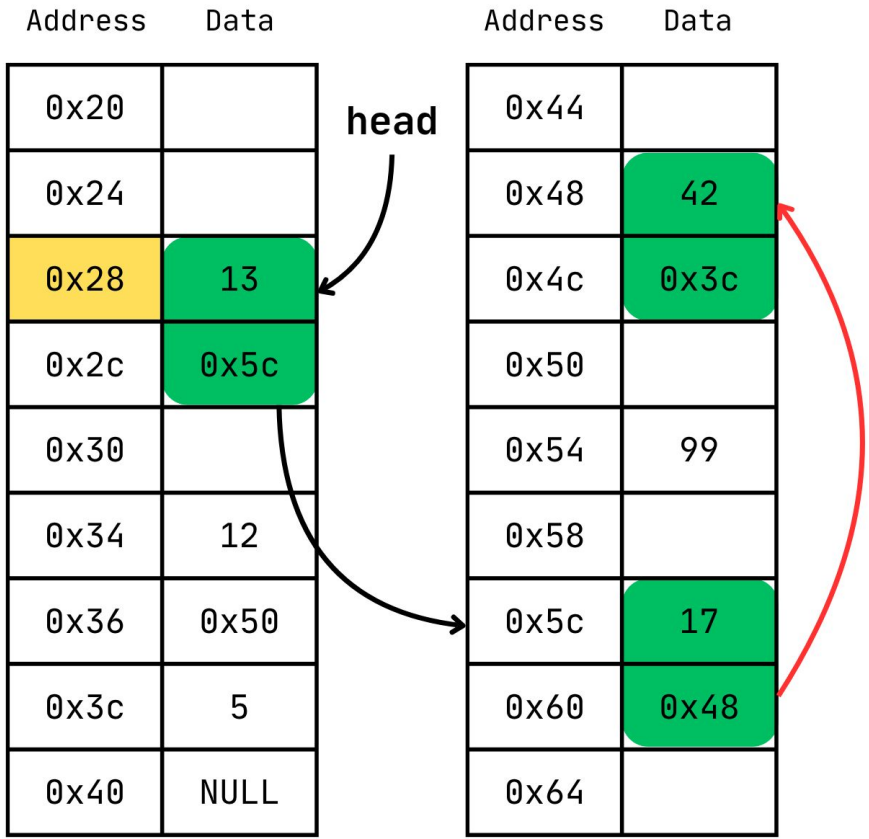
Linked Lists in Memory

- We have a pointer to the first piece of data
- For every piece of data you store in the list, you need to store a link (pointer containing the address) to the next item in the list
  - Like a scavenger hunt



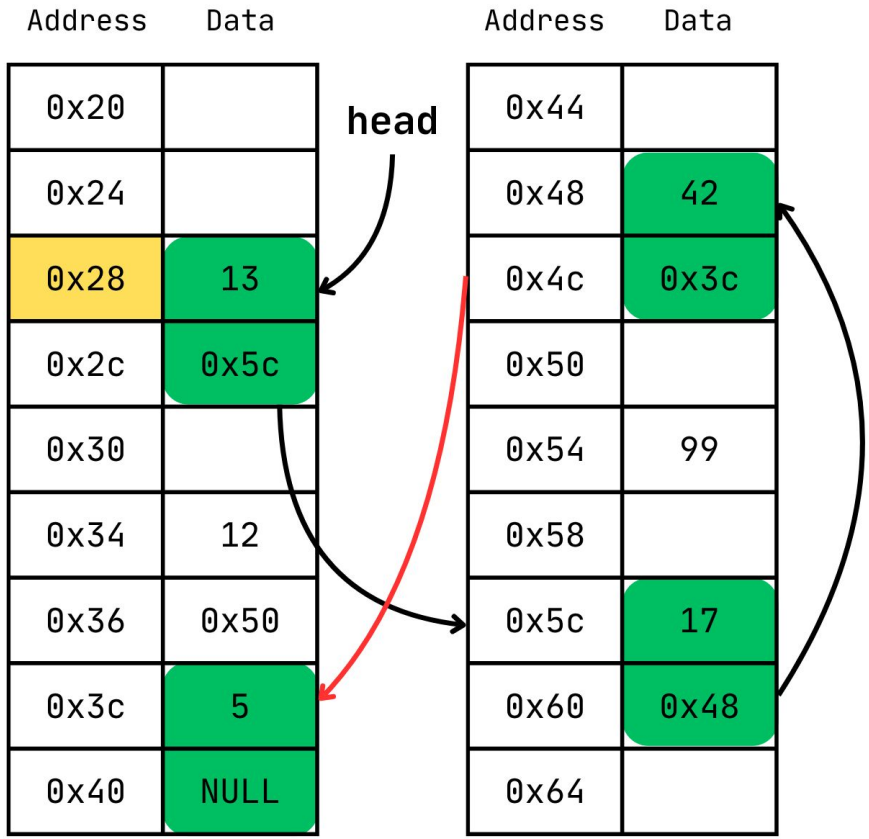
Linked Lists in Memory

- We have a pointer to the first piece of data
- For every piece of data you store in the list, you need to store a link (pointer containing the address) to the next item in the list
  - Like a scavenger hunt



Linked Lists in Memory

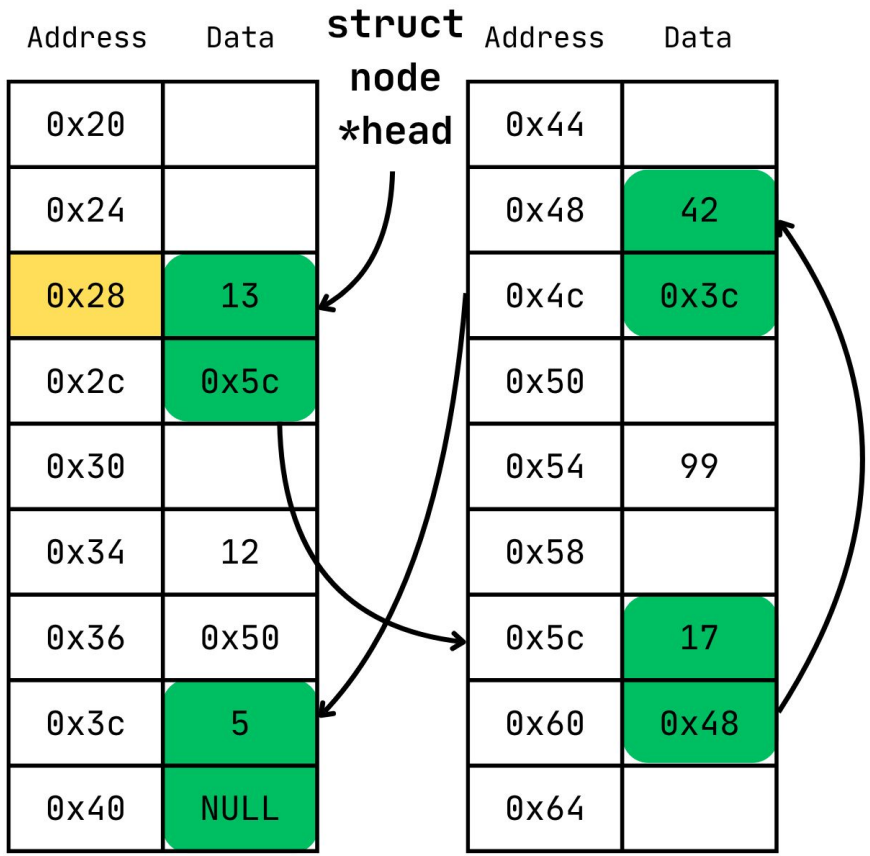
- When the value to the next piece of data is NULL, you reached the final node
  - Known as the tail
- Congratulations! You have traversed a linked list



Linked List Nodes

- We store our data and a pointer together in a struct
- The head variable is a pointer to the first node in the list

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



Creating a Node

```
struct node {
    int data;
    struct node *next;
};
struct node *create_node(int num) {



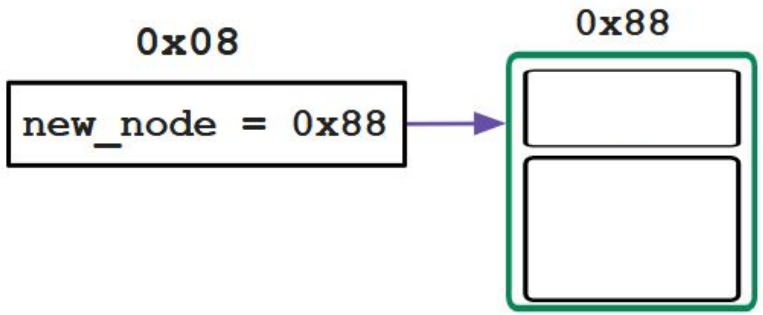
}
```

Creating a Node

```
struct node {
    int data;
    struct node *next;
};
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));



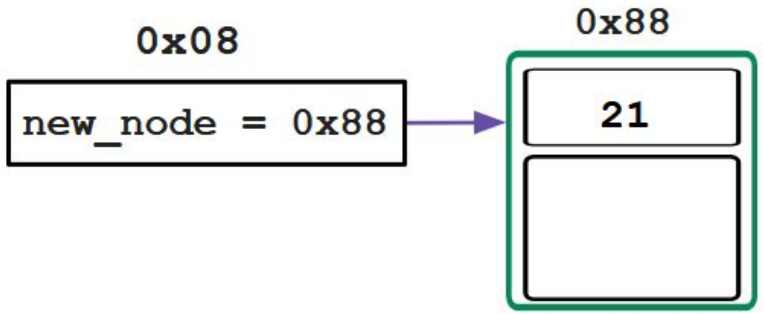

}
```



Creating a Node

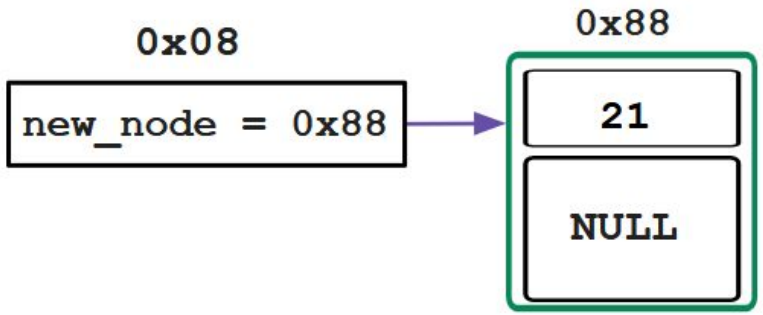
```
struct node {
    int data;
    struct node *next;
};
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = num;


}
```



Creating a Node

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

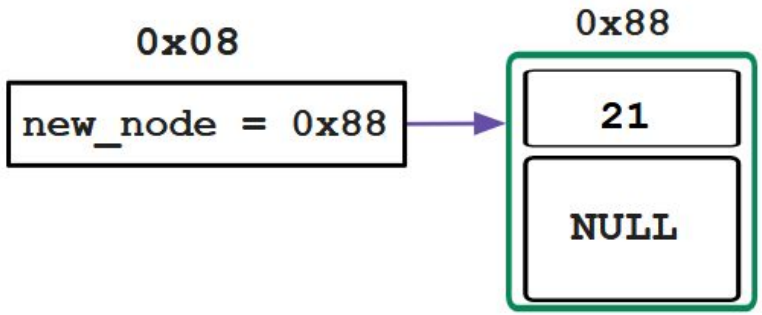



```
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = NULL;

}
```

Creating a Node

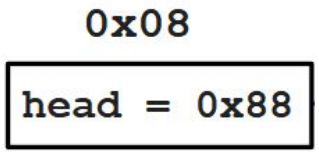
```c
struct node {
    int data;
    struct node *next;
};
```




```c
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = NULL;
    return new_node;
}
```

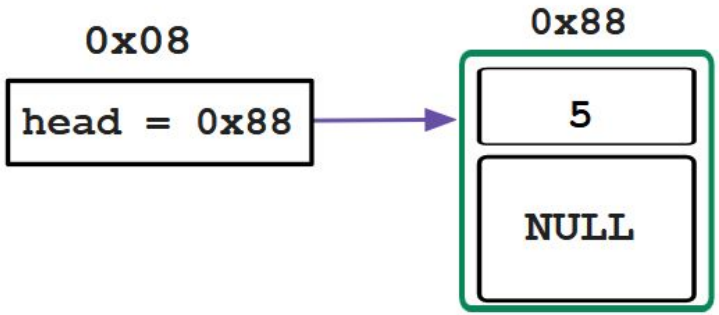
Creating a Linked List with 1 Node

```
struct node {
    int data;
    struct node *next;
};
struct node *head = NULL;
```

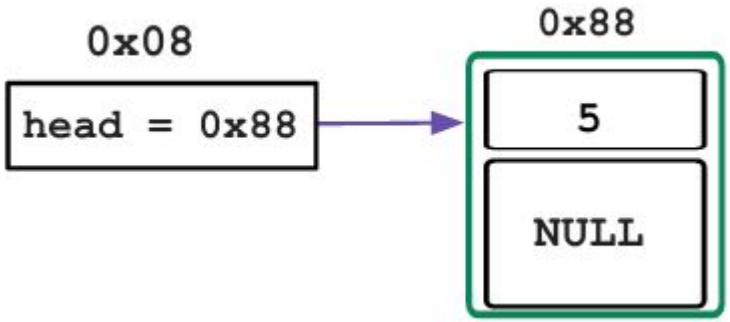


Creating a Linked List with 1 Node

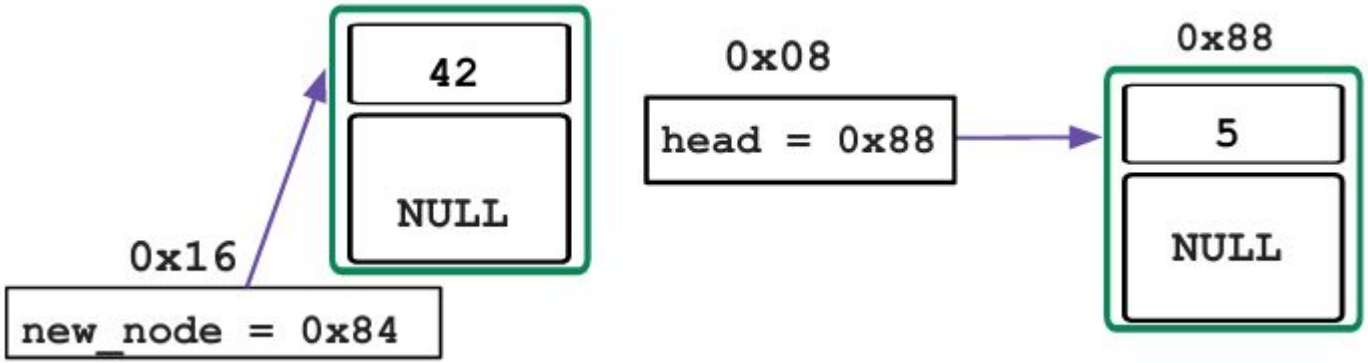
```
struct node {
    int data;
    struct node *next;
};
struct node *head = NULL;
head = create_node(5);
```



Adding a Node to the Front of a Linked List

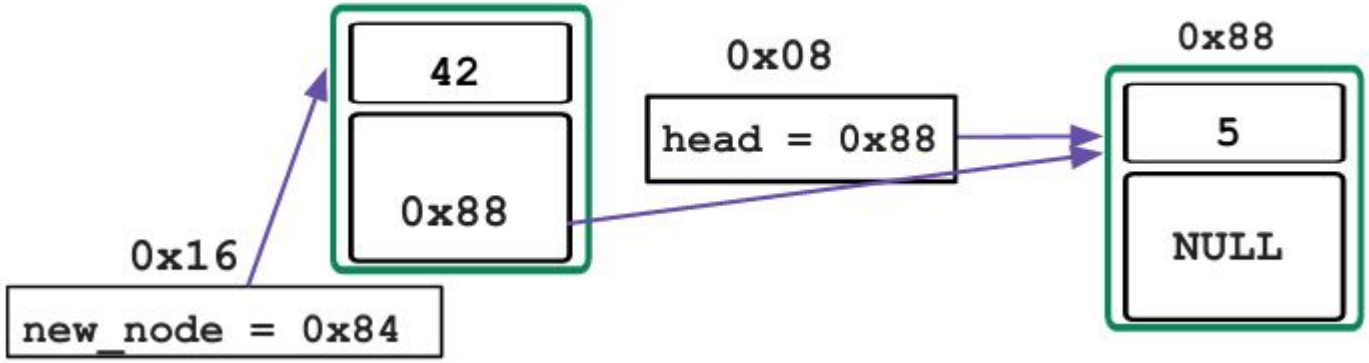


Adding a Node to the Front of a Linked List



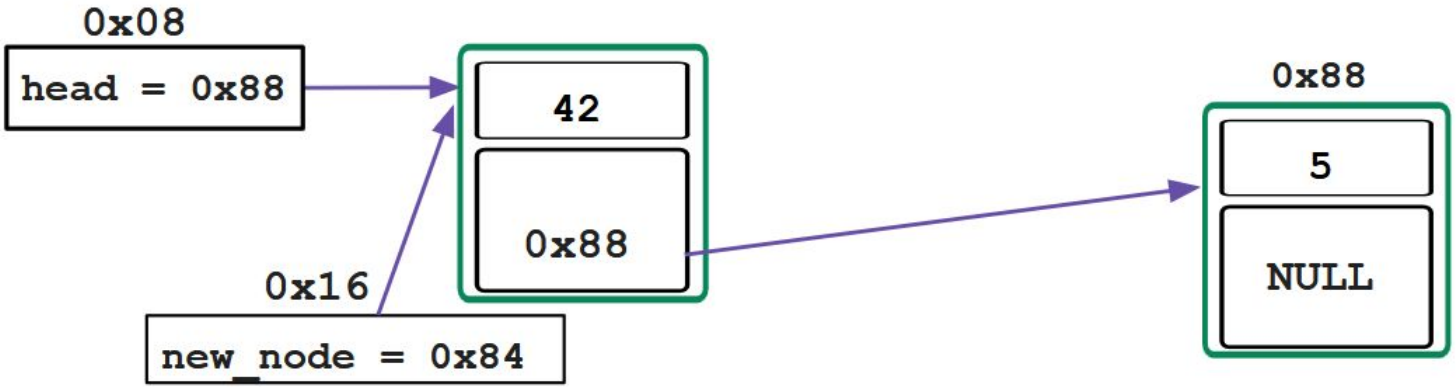

```
struct node *new_node = create_node(42);
```

Adding a Node to the Front of a Linked List




```
struct node *new_node = create_node(42);
new_node->next = head;
```

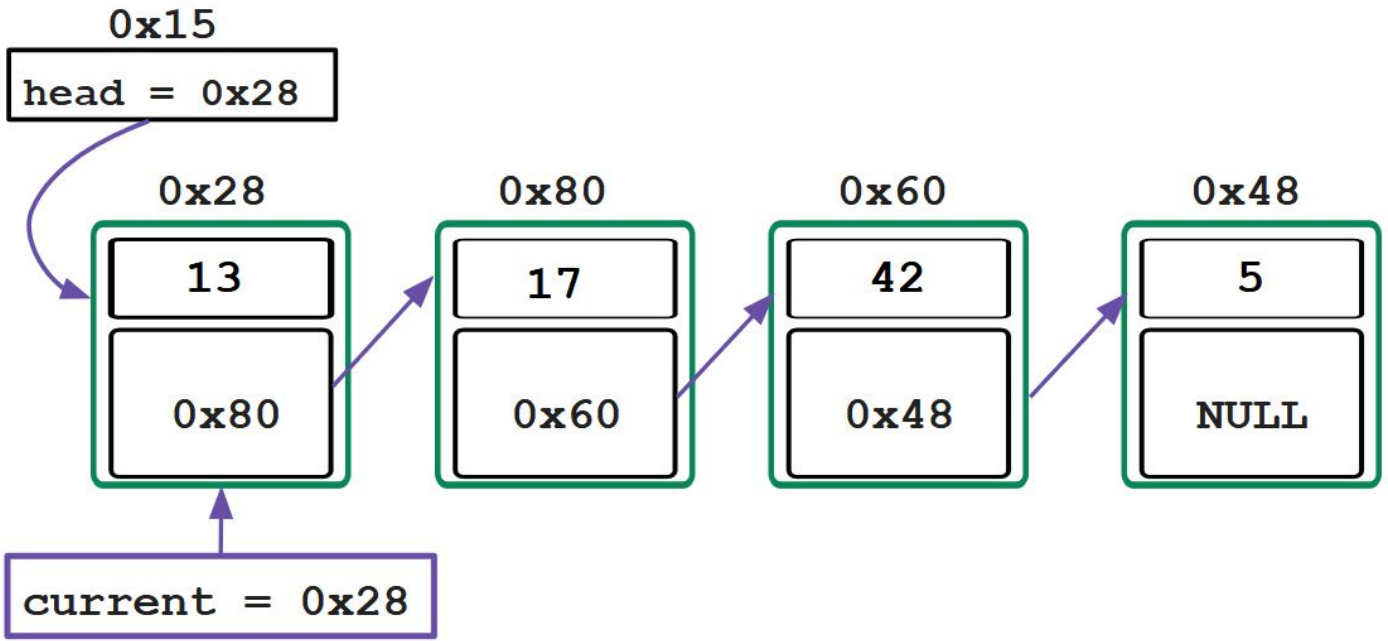
Adding a Node to the Front of a Linked List




```
struct node *new_node = create_node(42);
new_node->next = head;
head = new_node;
```

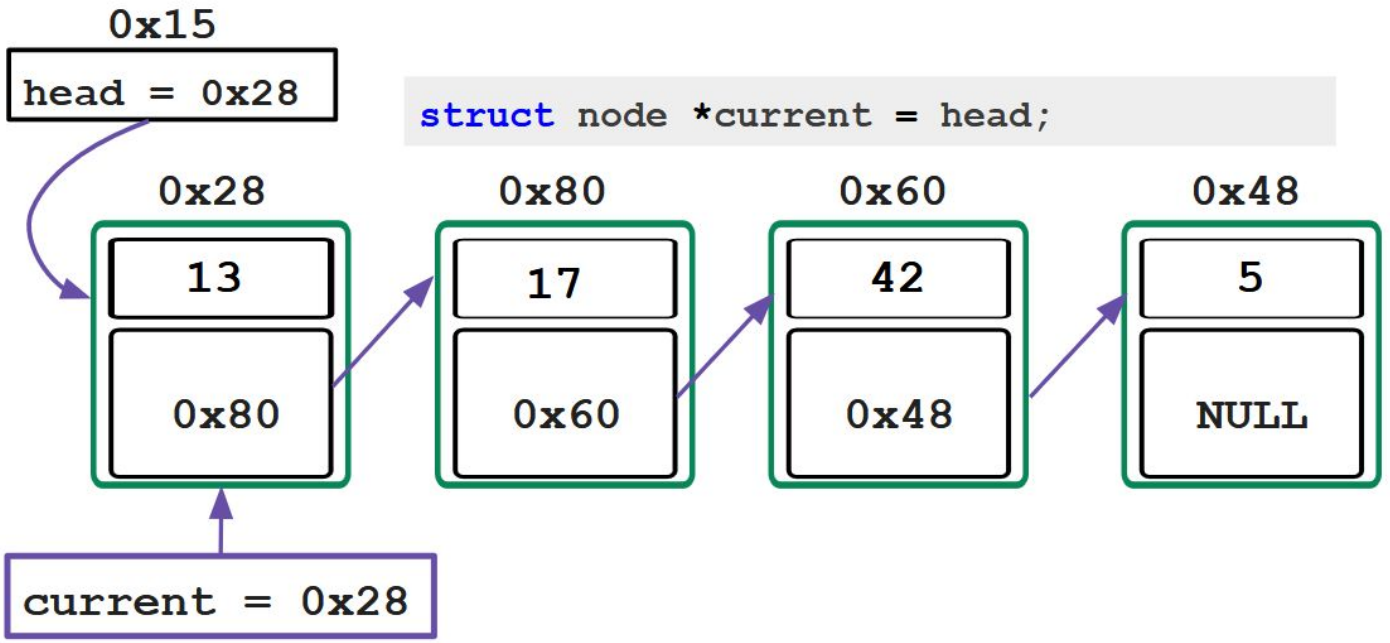
Traversing a List

- Set a pointer to the beginning of the list



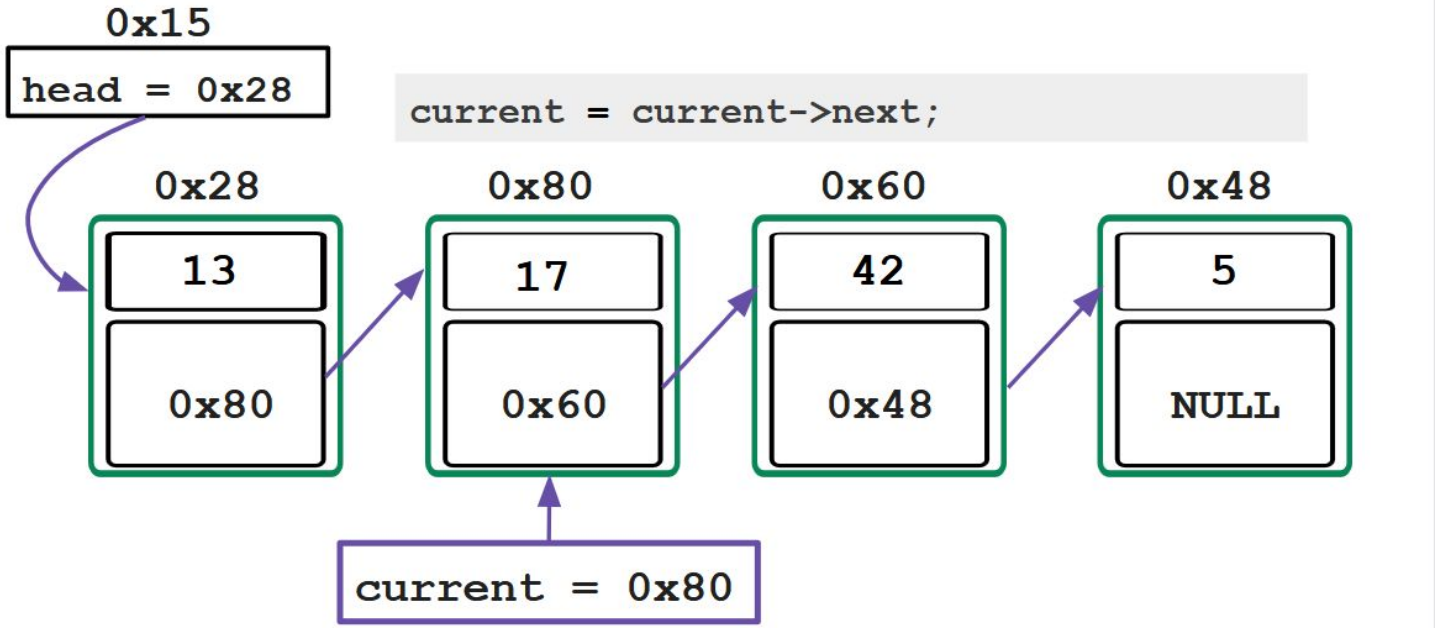
Traversing a List

- Set a pointer to the beginning of the list



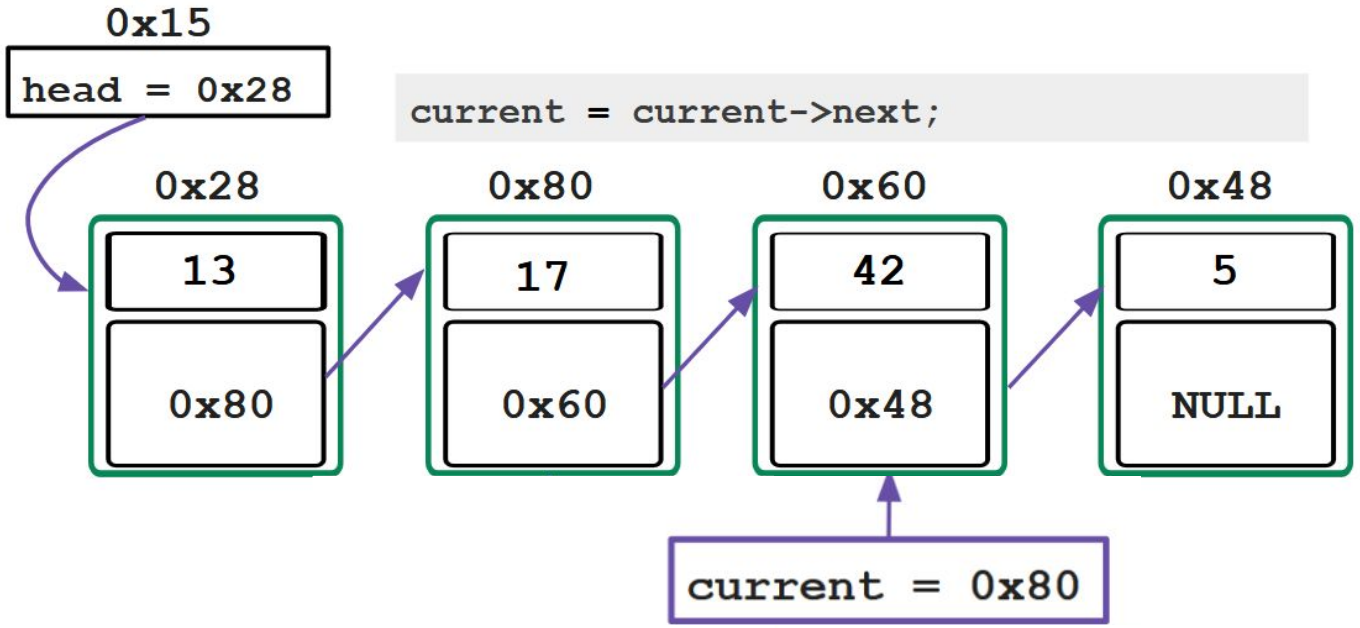
Traversing a List

- Move current along



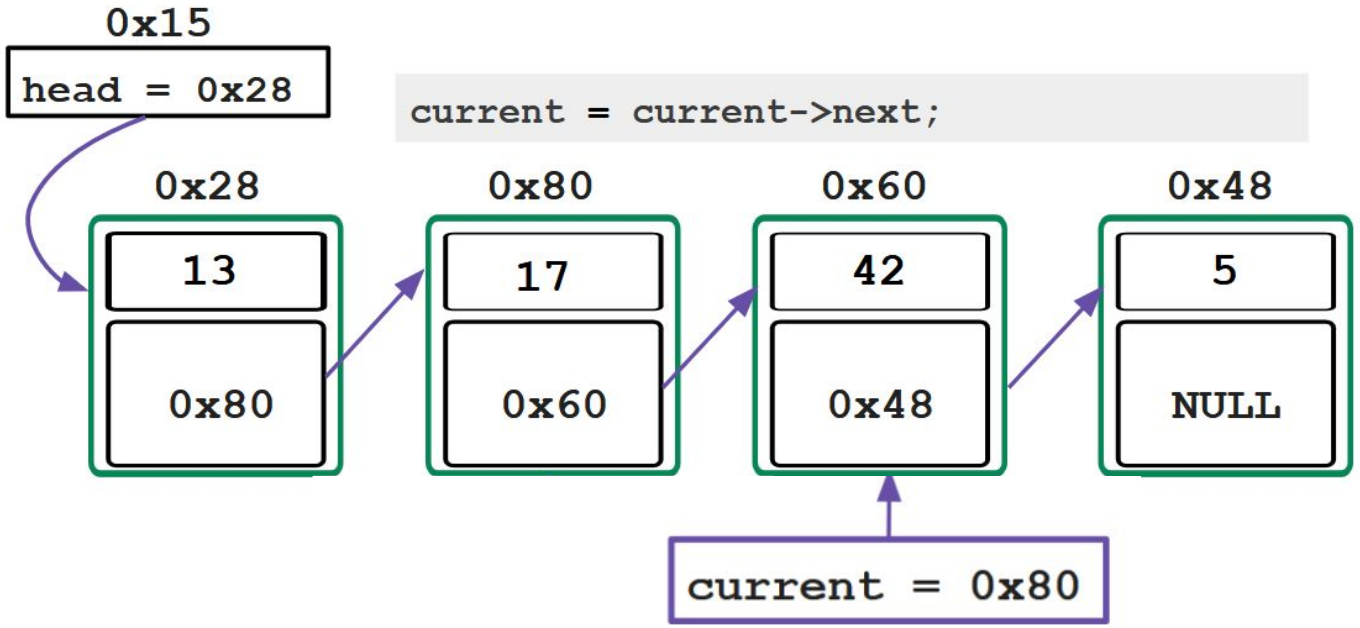
Traversing a List

- Move current along



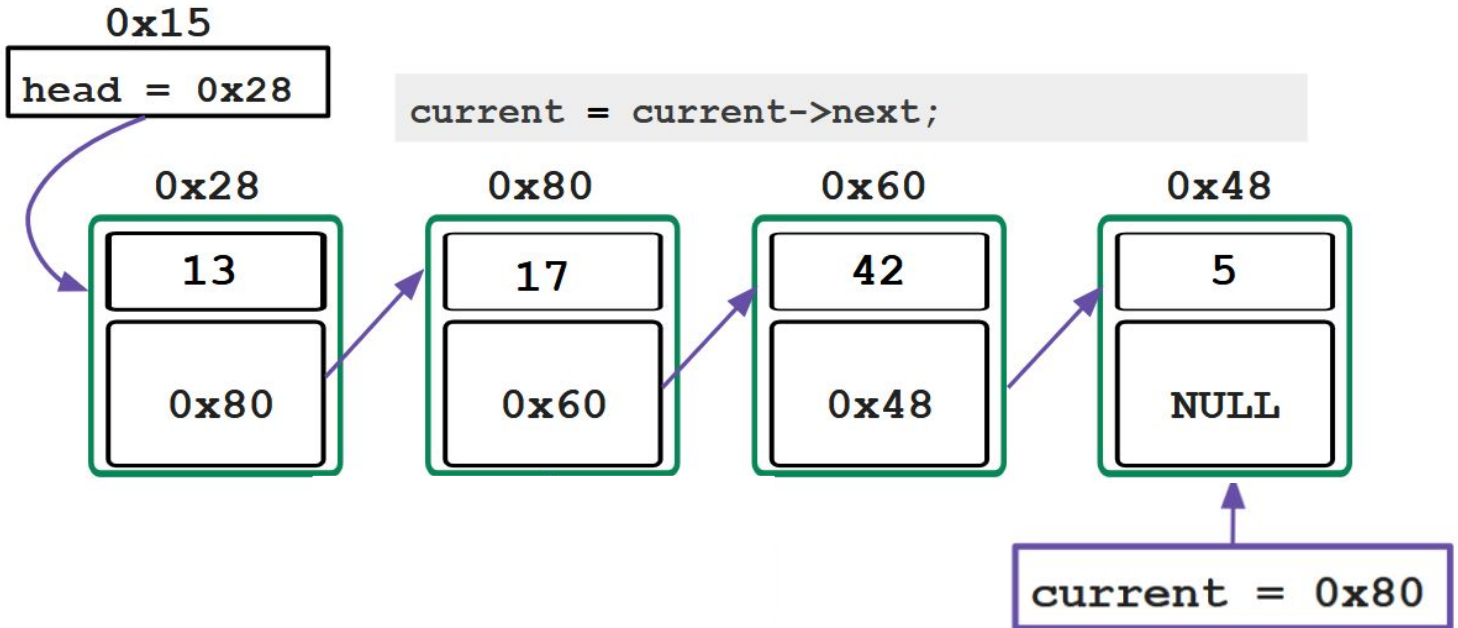
Traversing a List

- Move current along



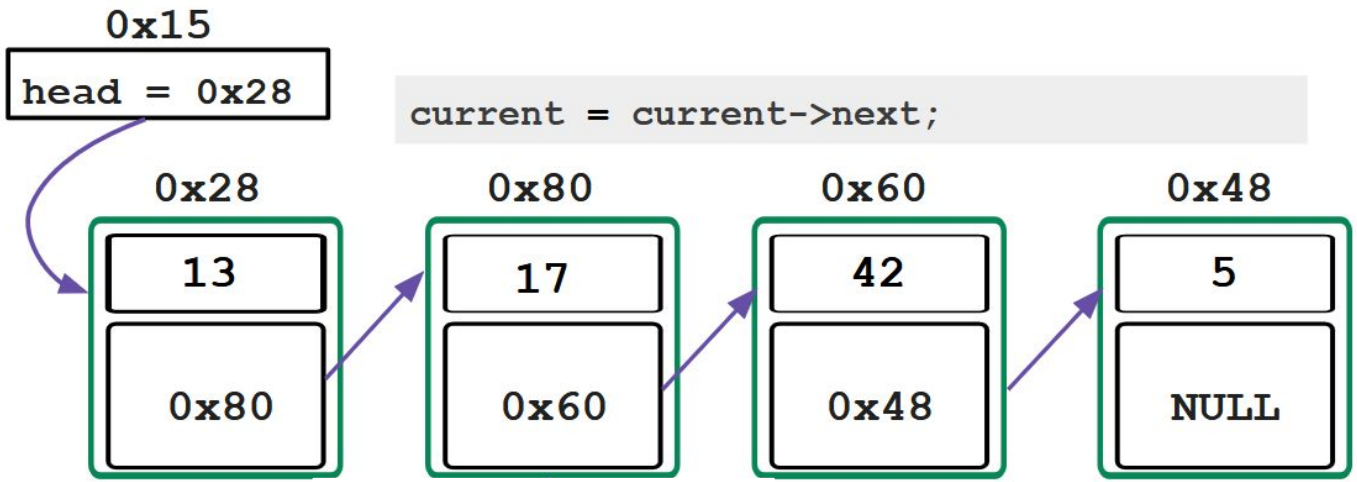
Traversing a List

- Move current along



Traversing a List

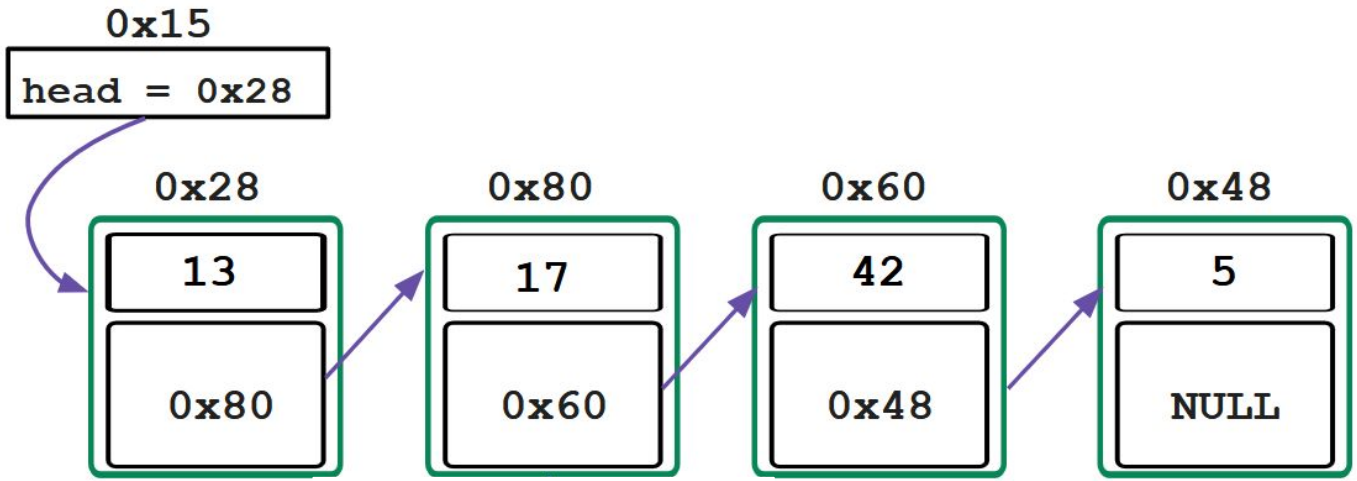
- Move current along



current = NULL

Traversing a List

- We should stop now that current is NULL

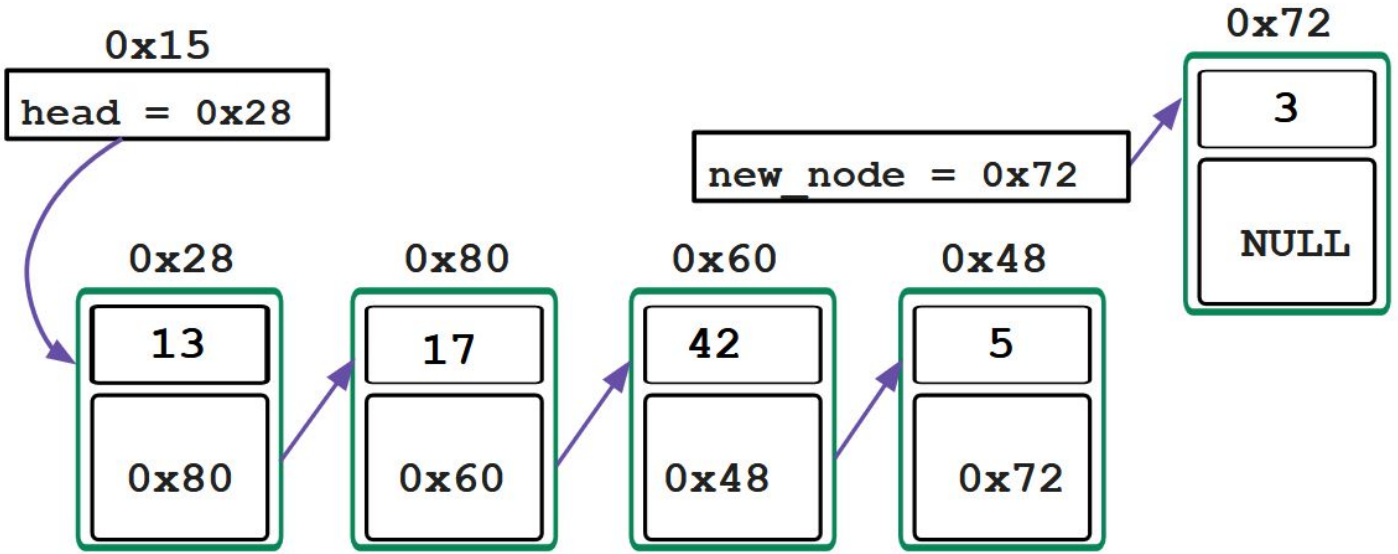


Inserting at the Tail (End) of the List

- To insert at the end of the list we need to
  - Find the last node in the list
  - Connect the last node in the list to the new node

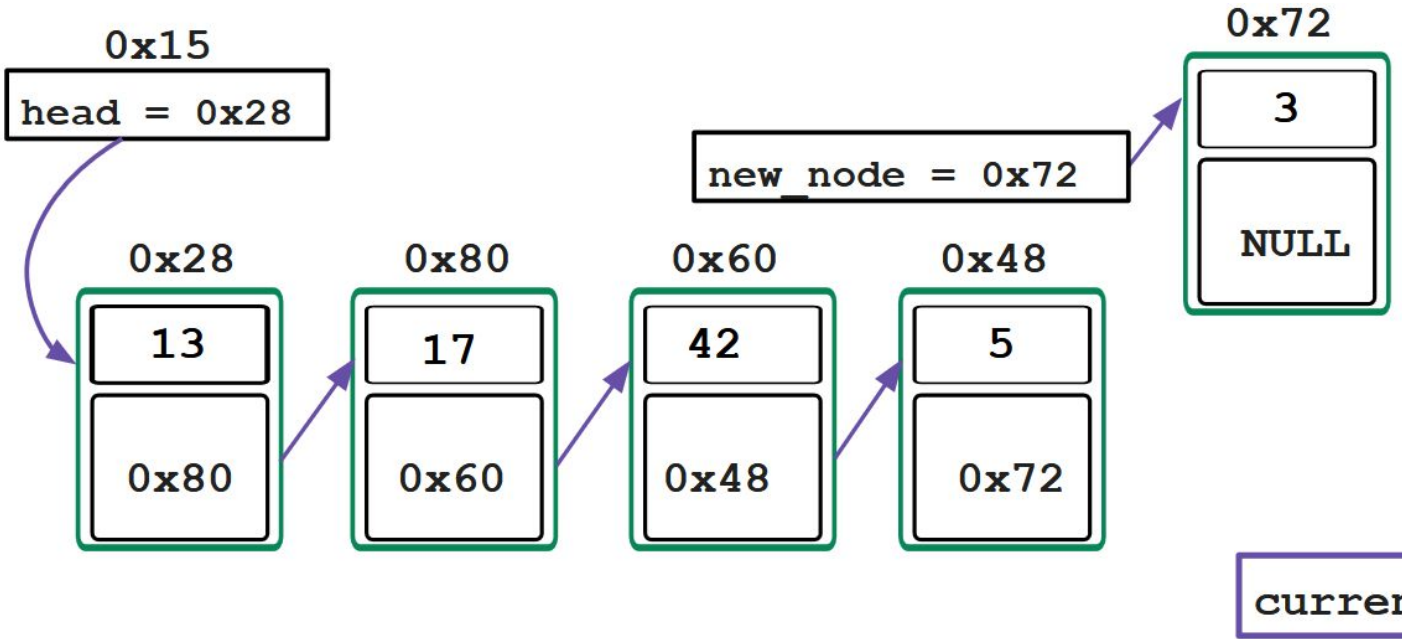
Inserting at the Tail (End) of the List

- If we stop traversing when current == NULL
  - We go past the end of the list



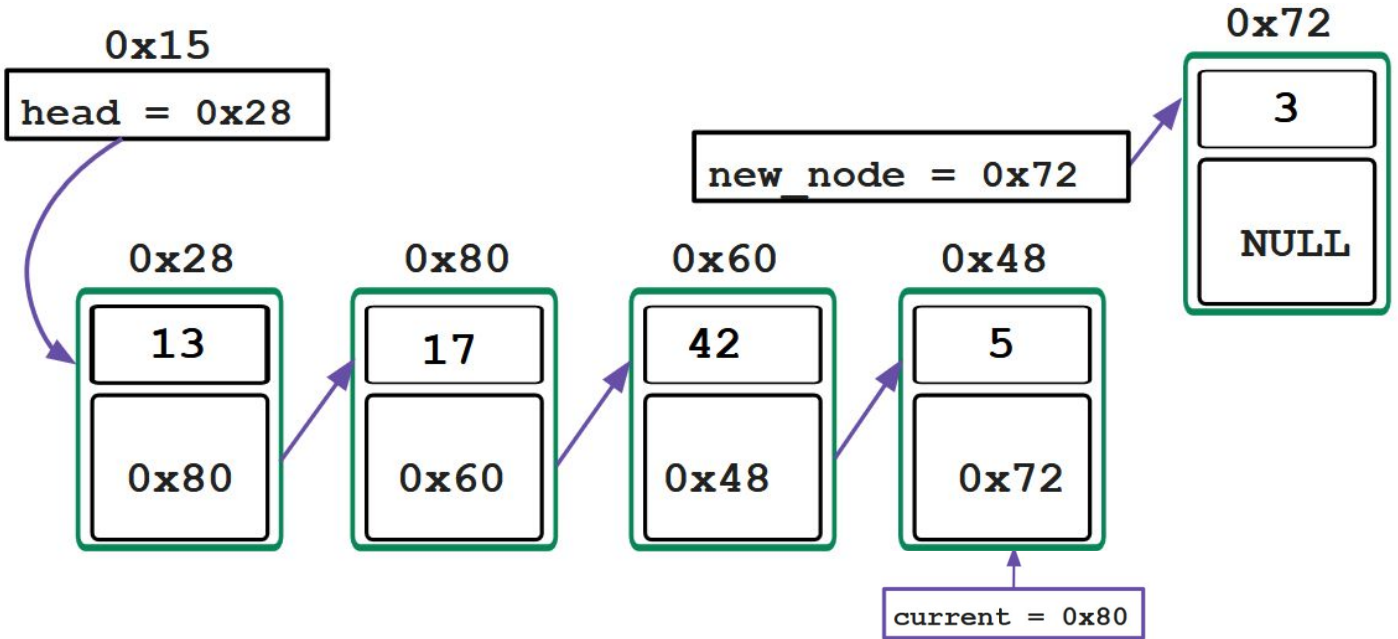
Inserting at the Tail (End) of the List

- If we stop traversing when current == NULL
  - We go past the end of the list



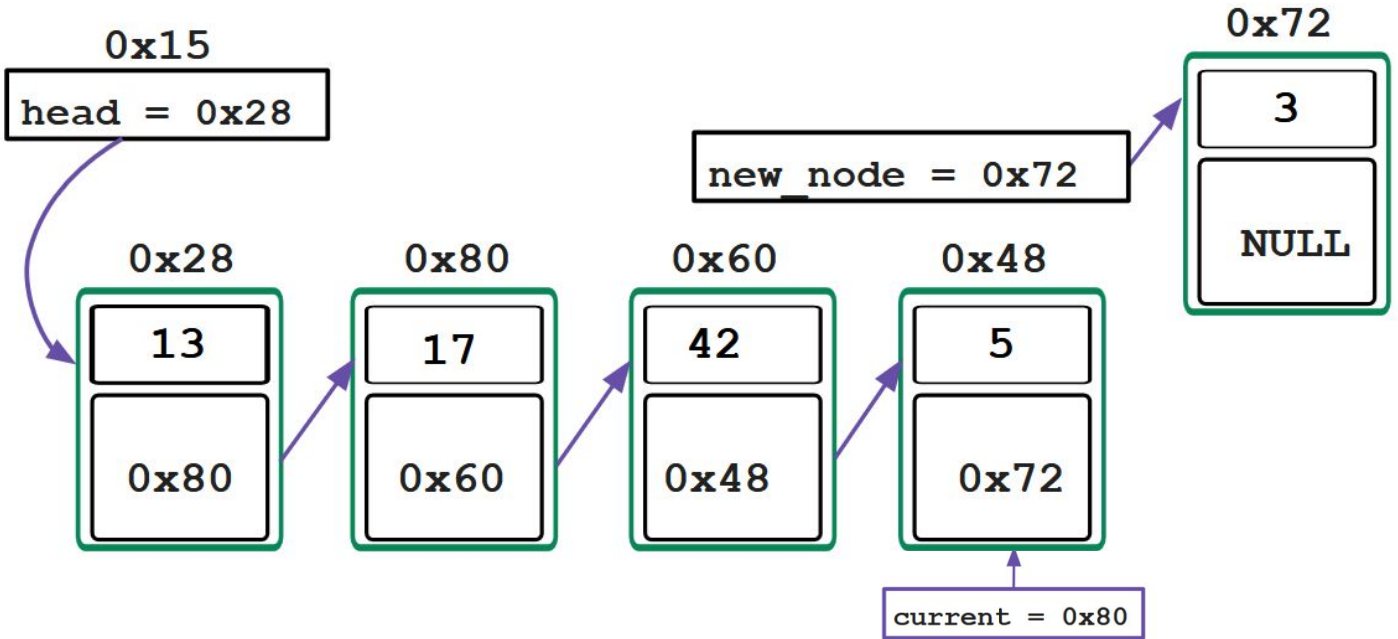
Inserting at the Tail (End) of the List

- We want to stop at the last node
  - How can we tell when we're there?



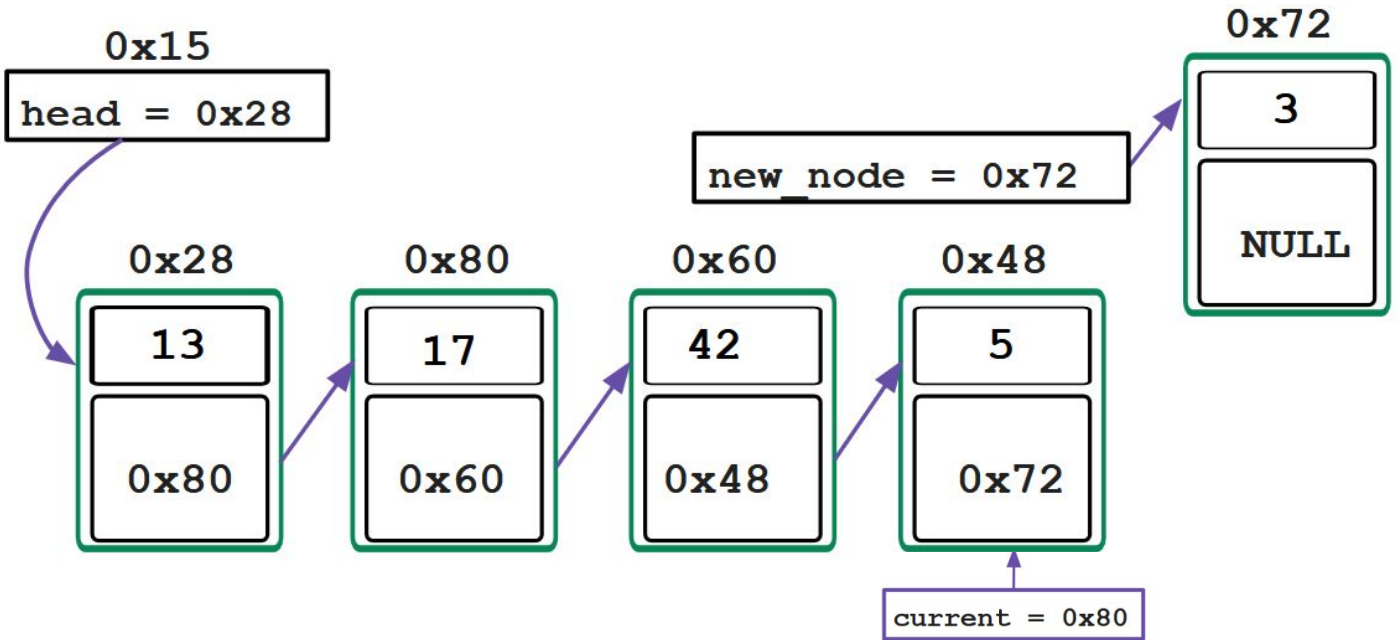
Inserting at the Tail (End) of the List

- We want to stop when
  - current->next == NULL



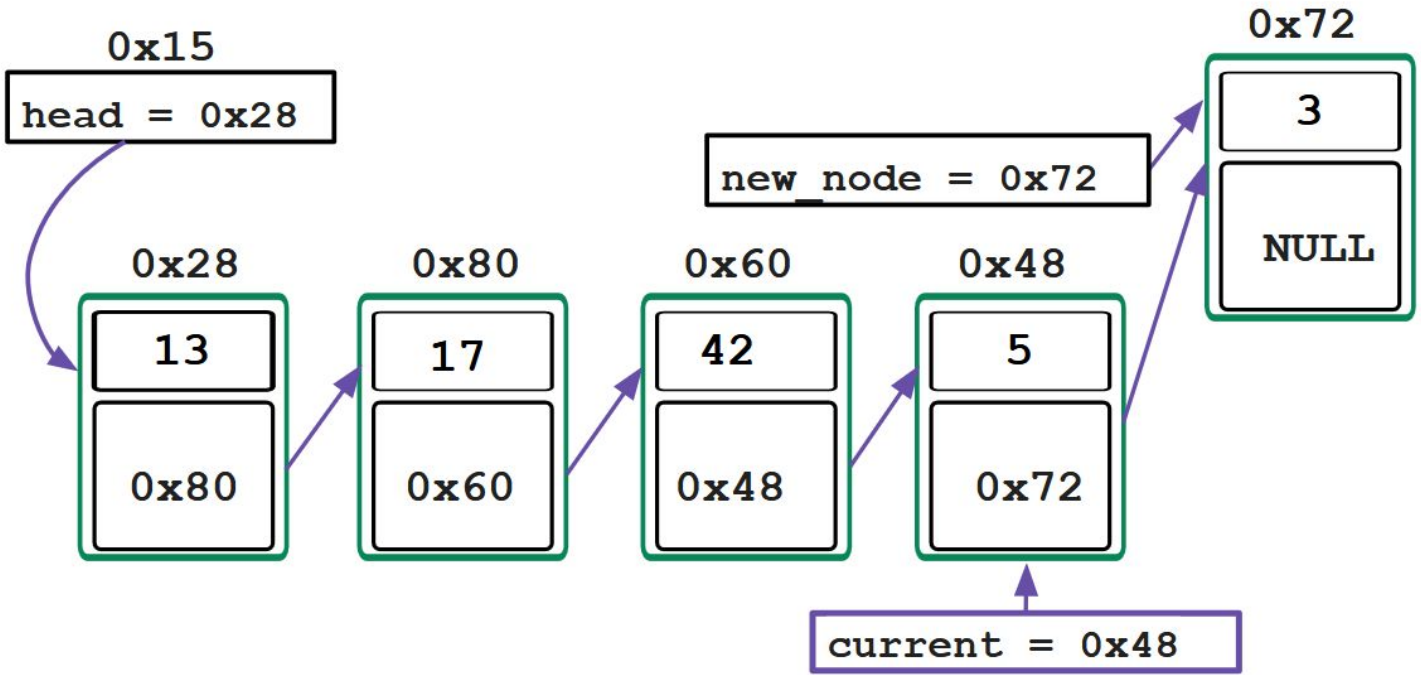
Inserting at the Tail (End) of the List

- Then we can link the last node to the new node



Inserting at the Tail (End) of the List

- `current->next = new_node`

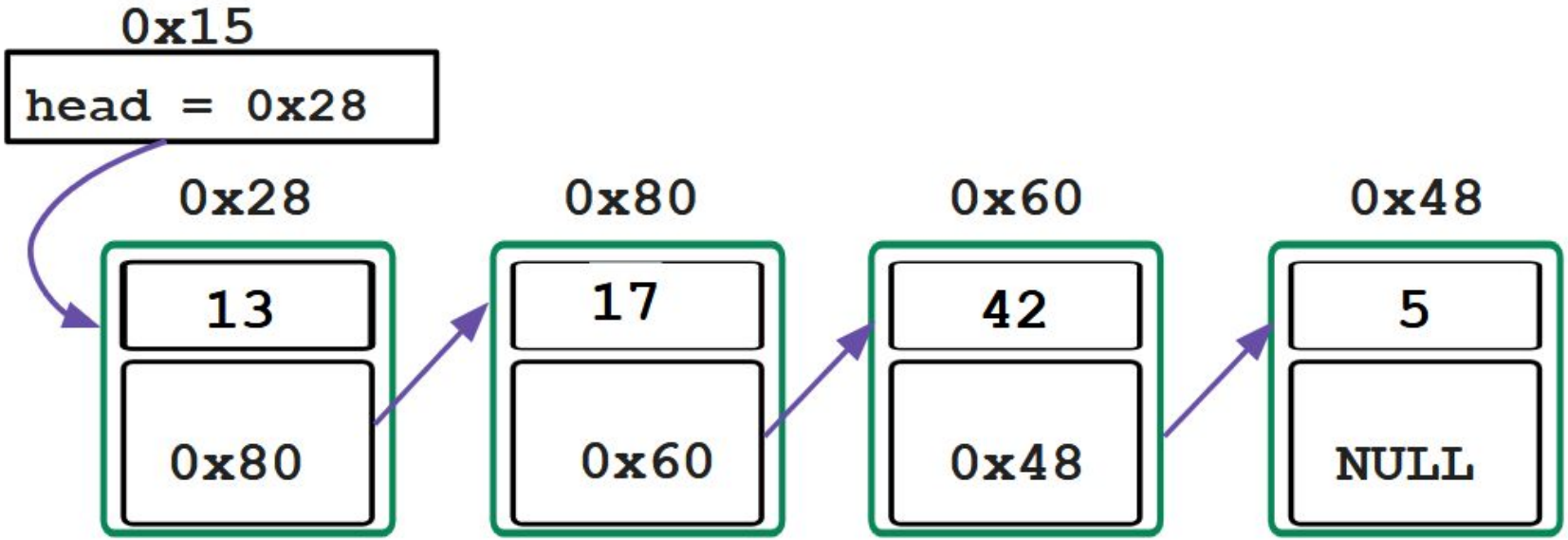


Inserting Node at a Given Position

- Last week we were inserting nodes based on the values that came before or after it
  - Now we want to insert at a given position
    - Kind of like an index in an array

Inserting Node at a Given Position

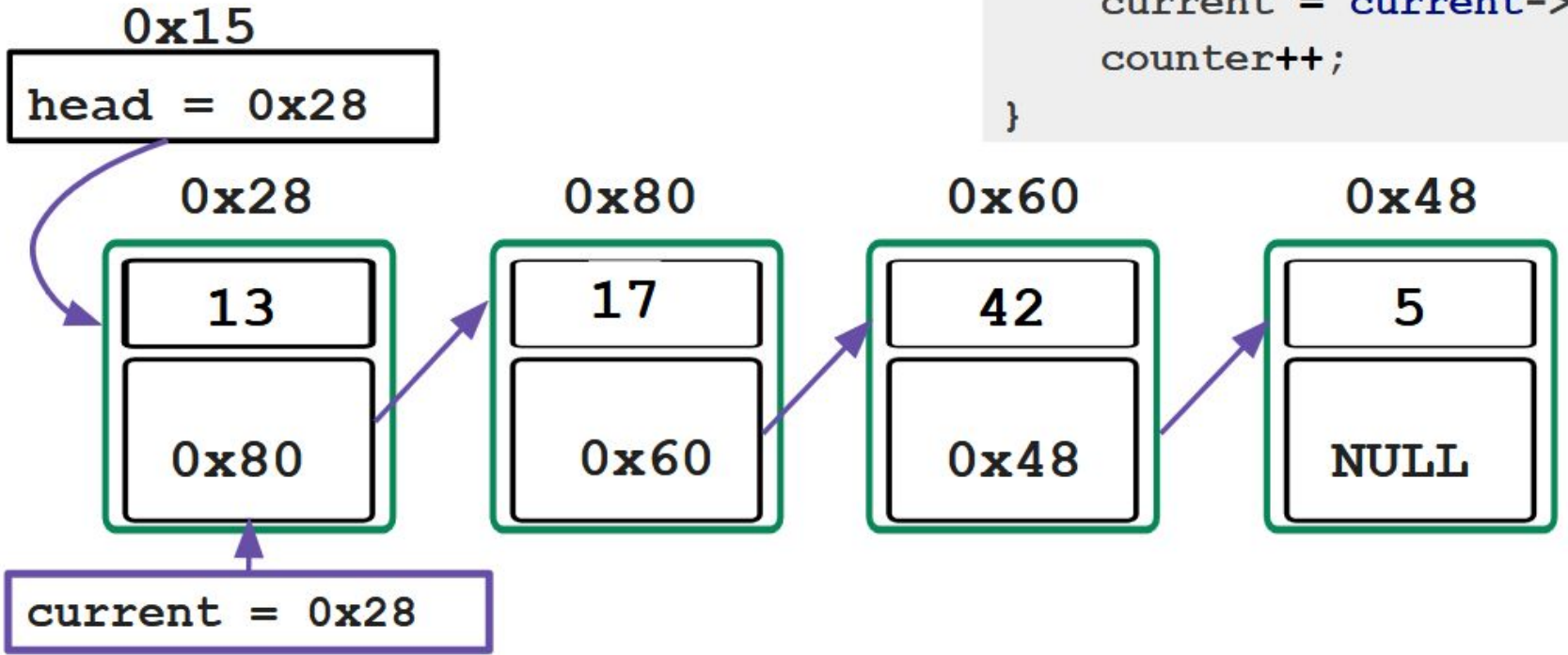
- We will use a counter to find the correct position and insert



Inserting at Position

- counter = 1

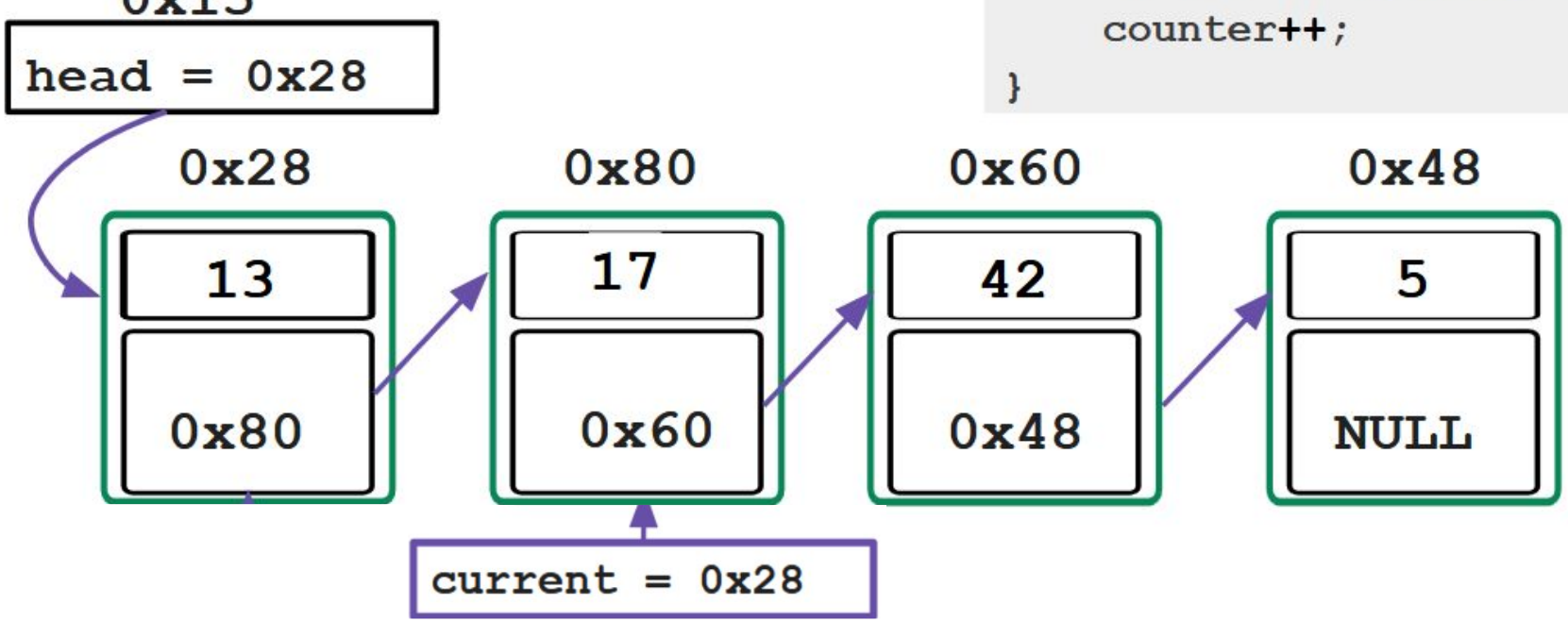
```
struct node *current = head;
int counter = 1;
while (counter < position) {
    current = current->next;
    counter++;
}
```



Inserting at Position

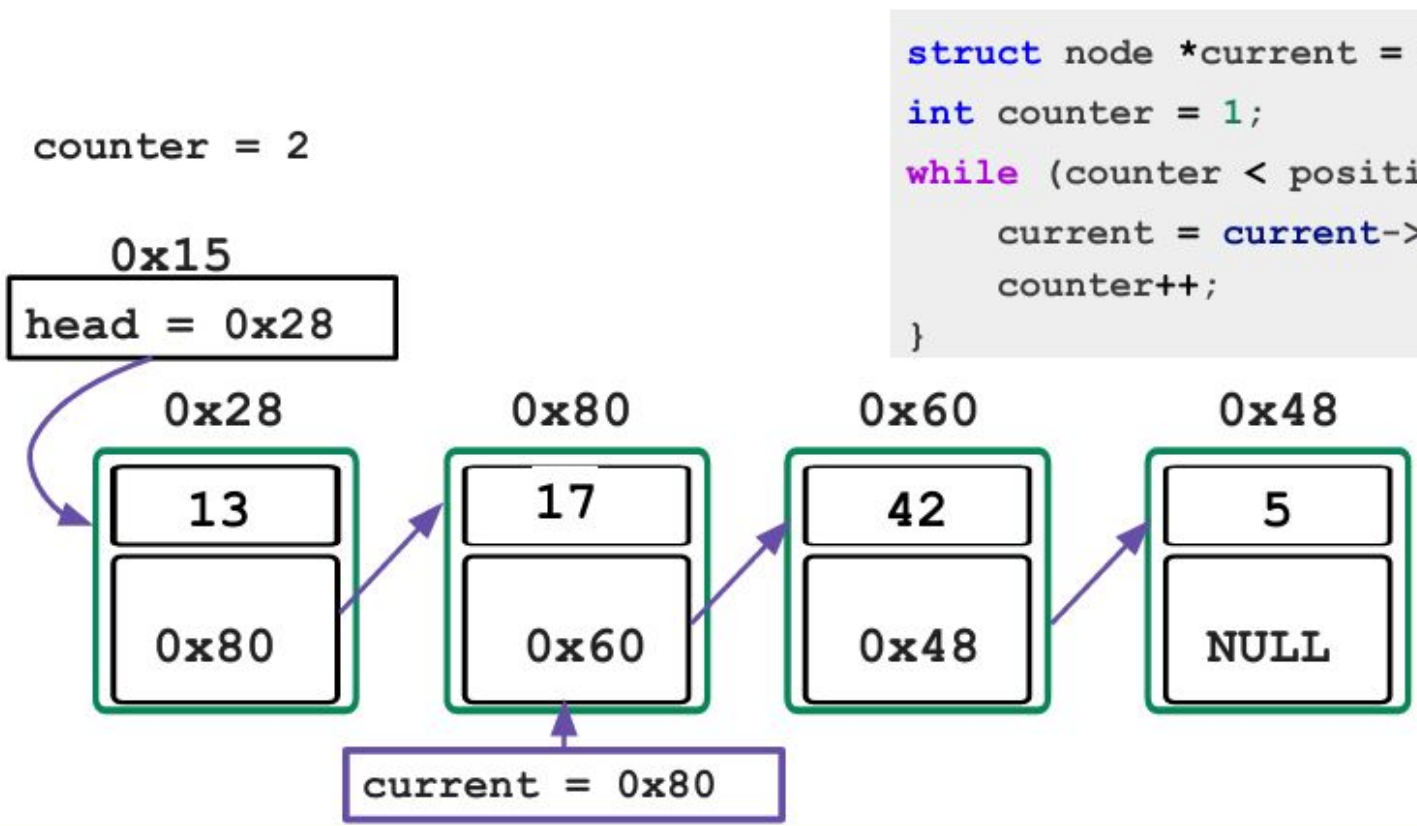
- counter = 2

```
struct node *current = head;
int counter = 1;
while (counter < position) {
    current = current->next;
    counter++;
}
```



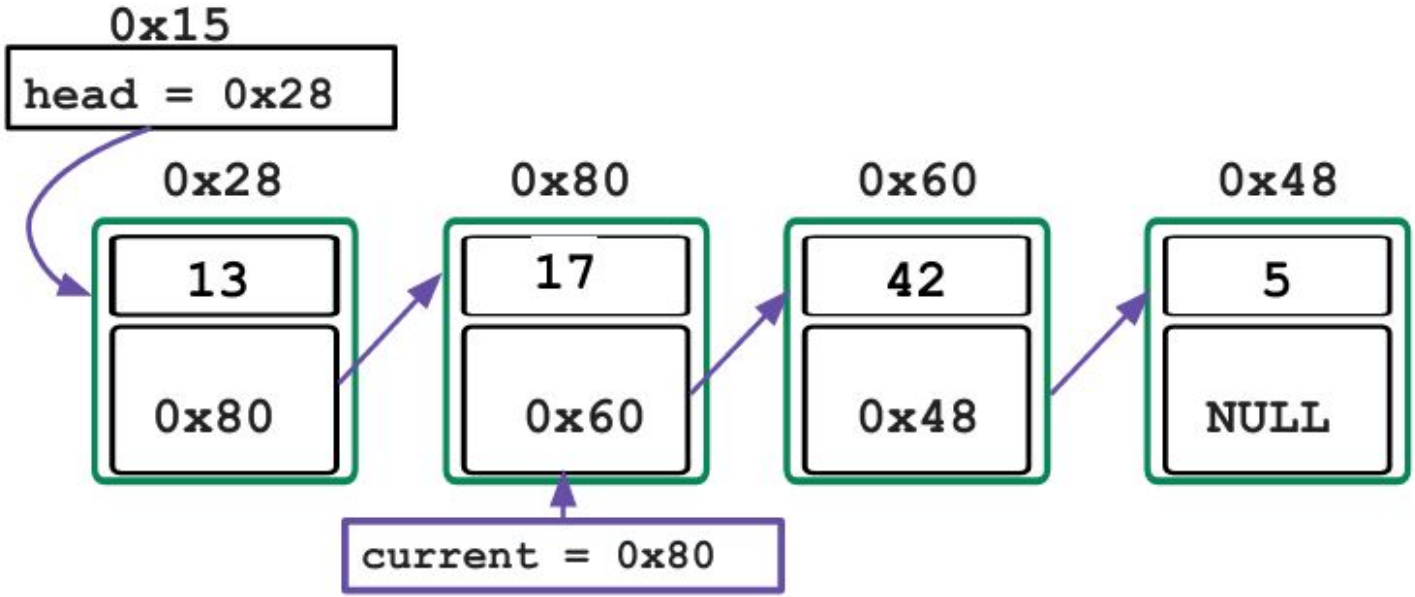
Inserting at Position

```
struct node *current = head;
int counter = 1;
while (counter < position) {
    current = current->next;
    counter++;
}
```

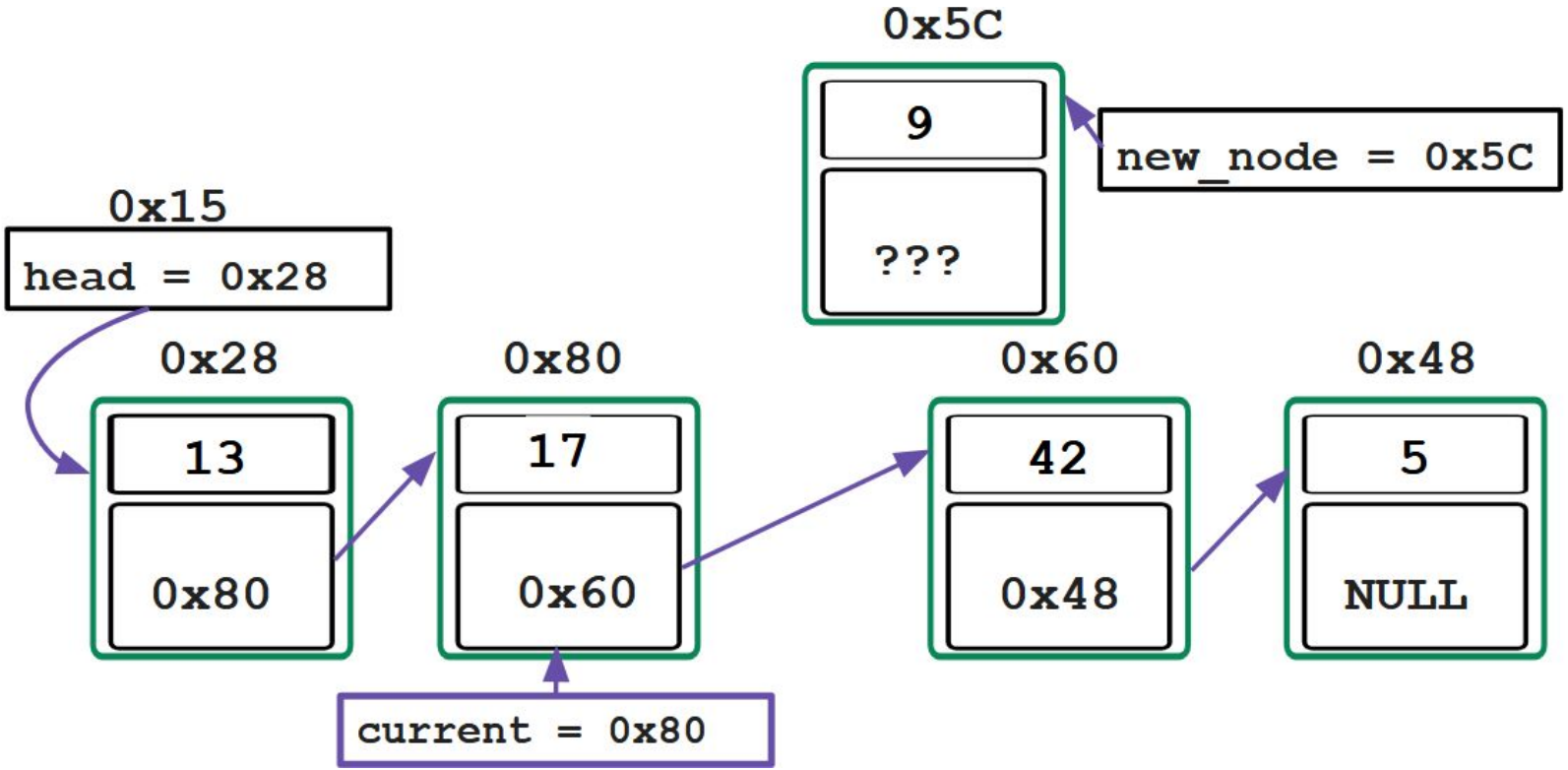


Inserting at Position

- Now we want to connect our new node. It should come after the current node, but before current->next

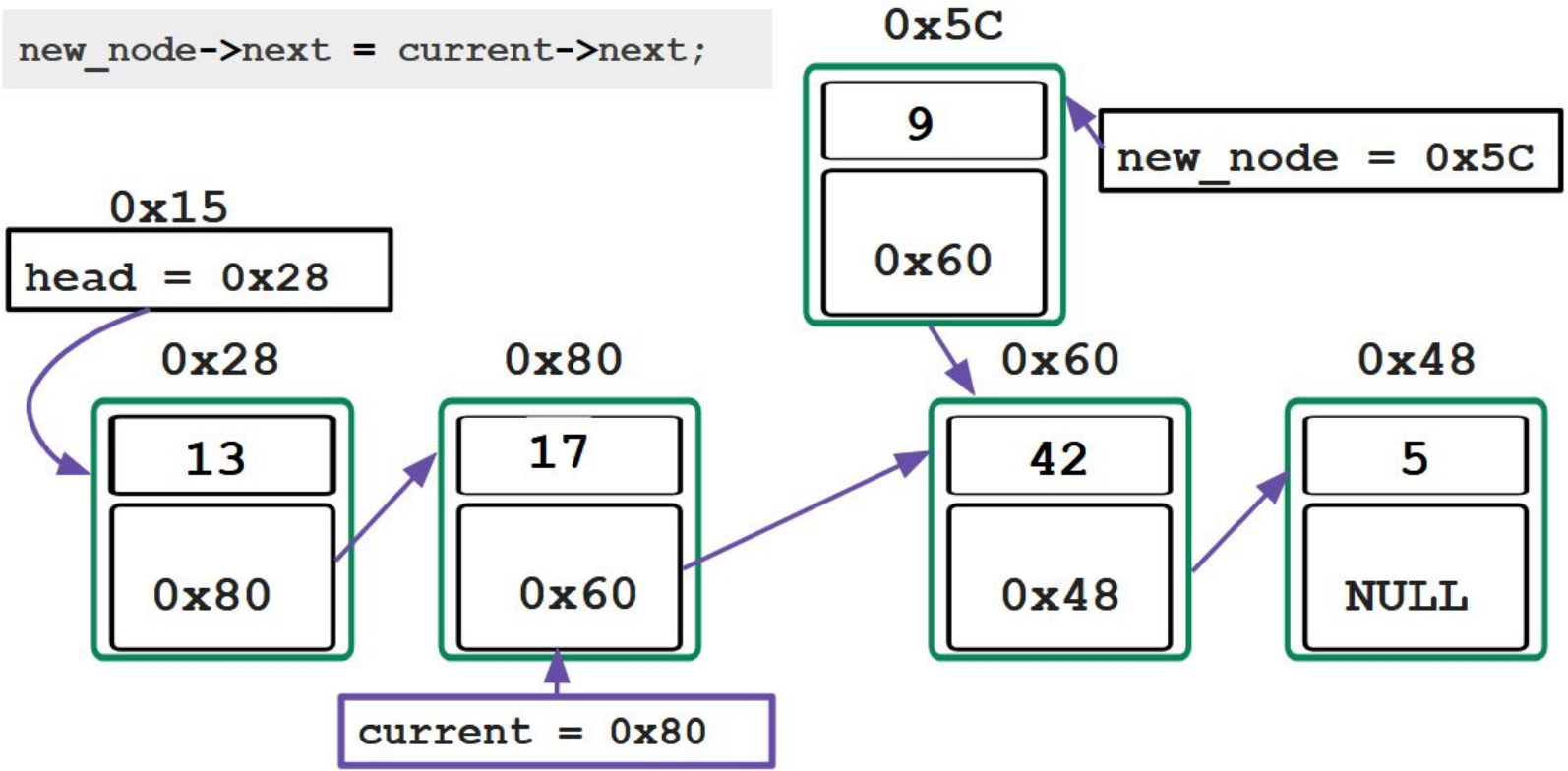


Inserting at Position



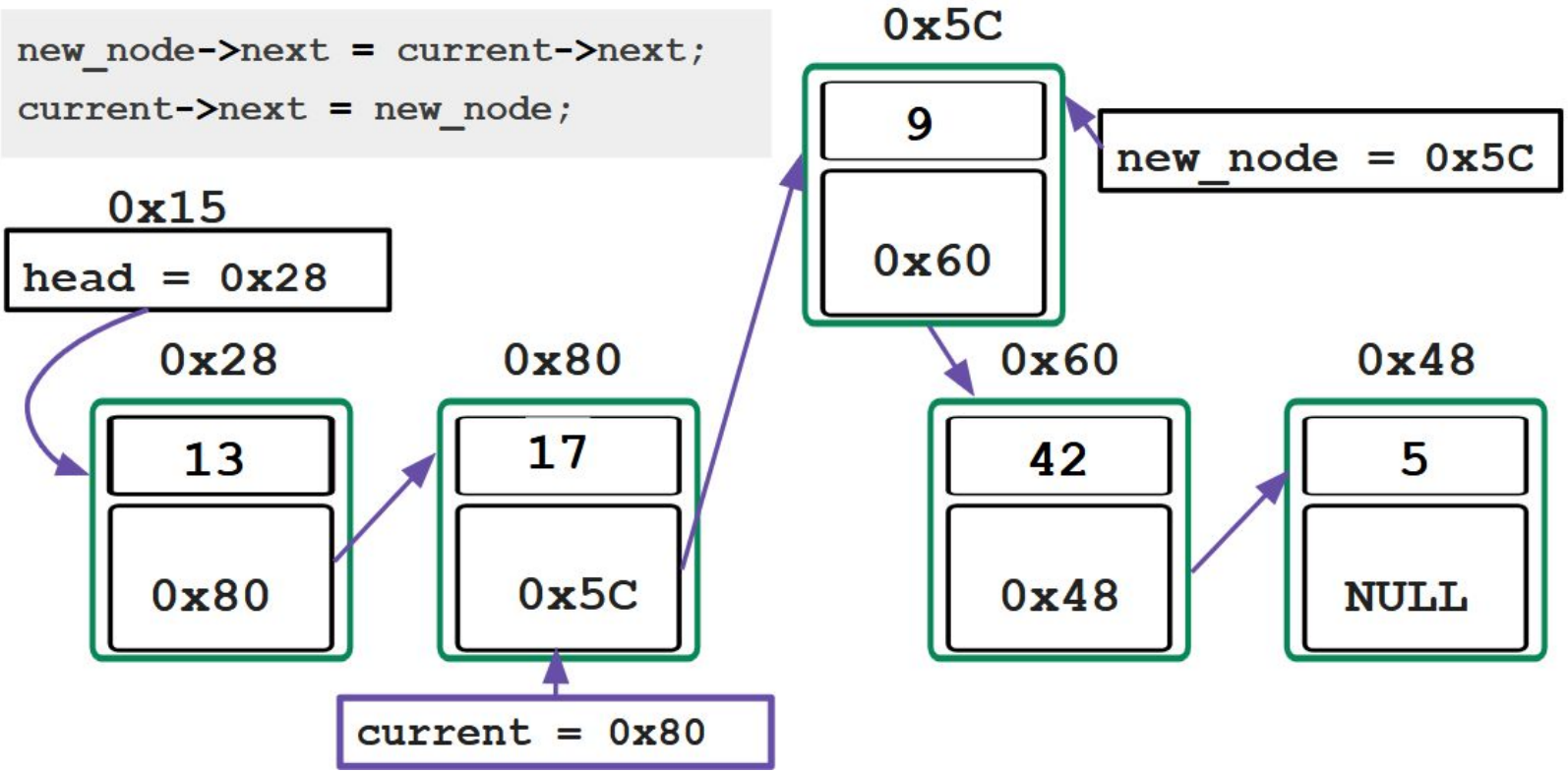
Inserting at Position

```
new_node->next = current->next;
```



Inserting at Position

```
new_node->next = current->next;
current->next = new_node;
```



Coding: Inserting at Position

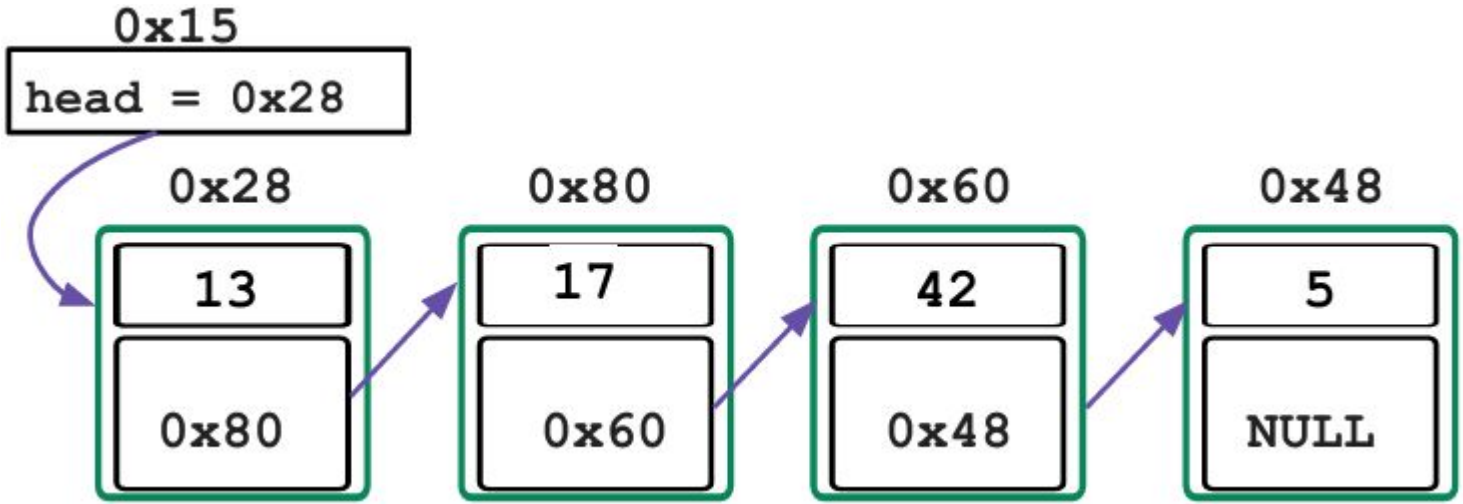
- What other special edge cases are there?
  - Empty list
  - Insert at the start (position 0 or less)
  - Insert at the tai
  - Insert past tail
  - Anything else we should check?

Break Time

...probably

Deleting the First Node in a Linked List

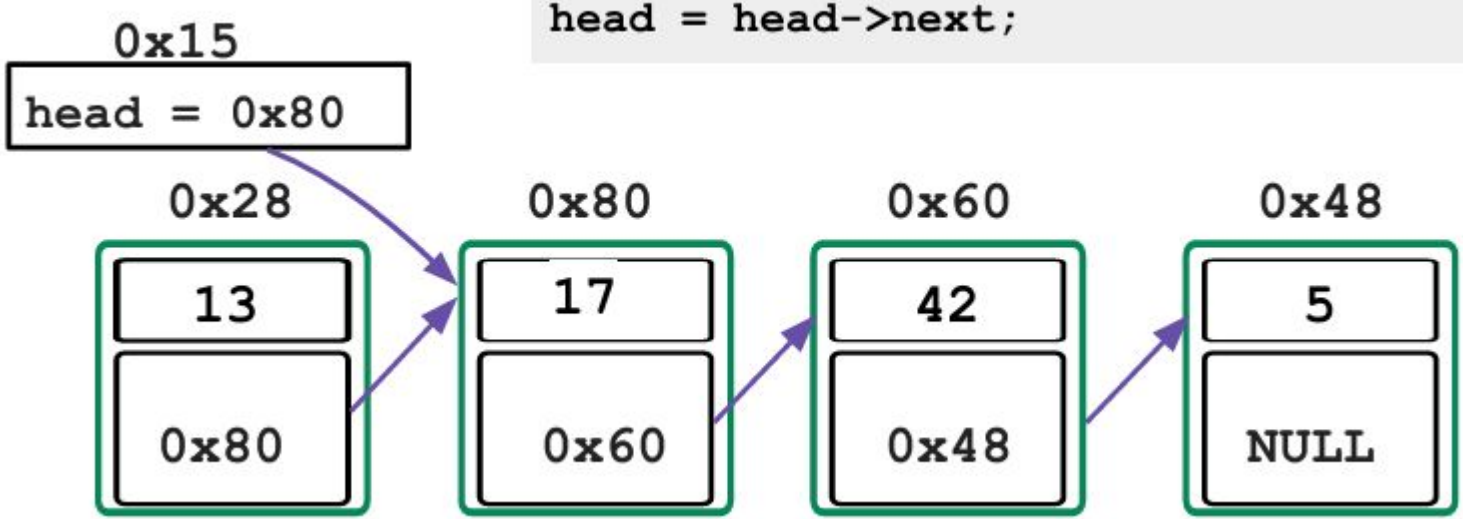
- If our list is not empty, we want to make the second node the new head of the list, and free the first node



Deleting the First Node in a Linked List

- What would be the problem with updating head first?

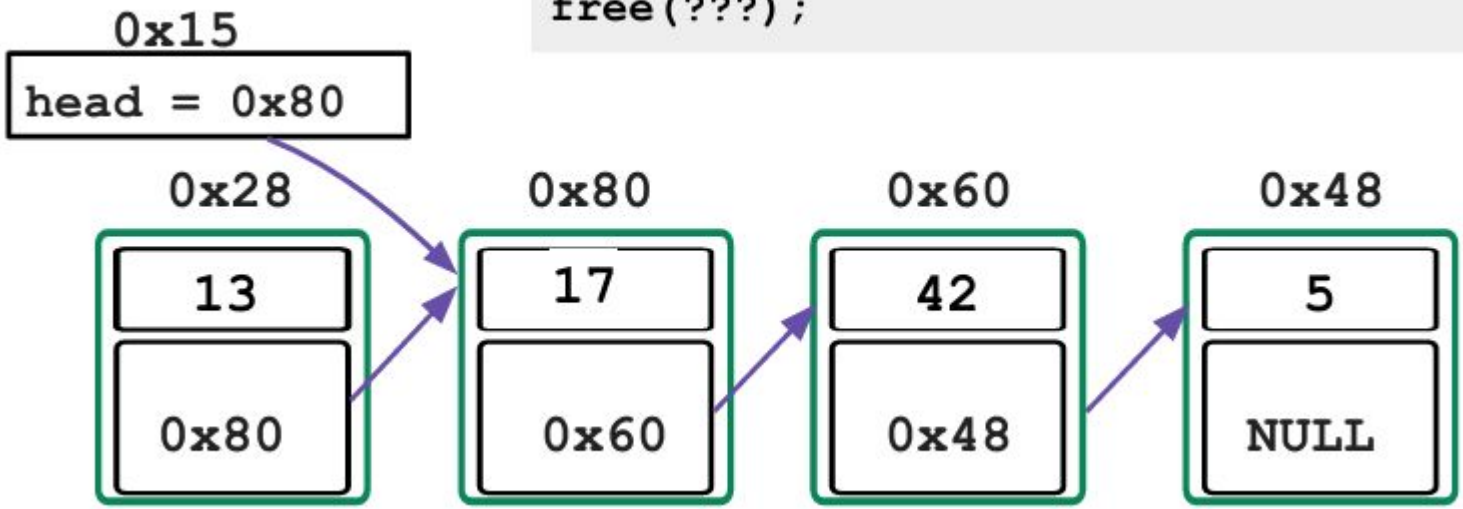
```
head = head->next;
```



Deleting the First Node in a Linked List

- We now have no pointer to the first node, so we can't free it!

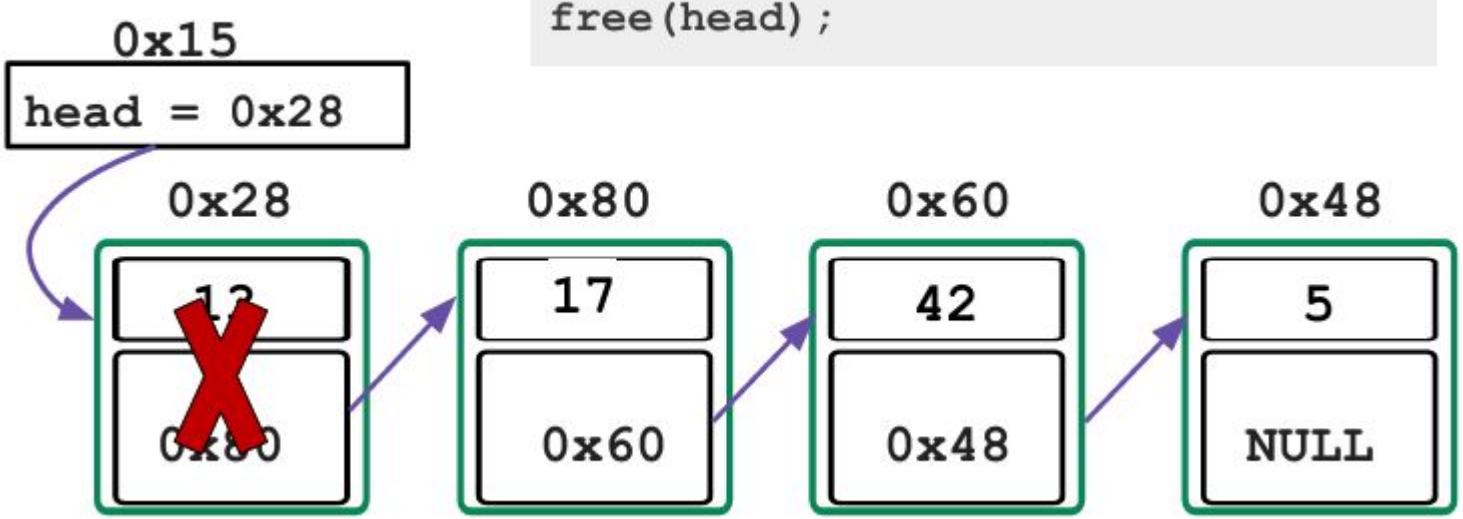
```
free(???);
```



Deleting the First Node in a Linked List

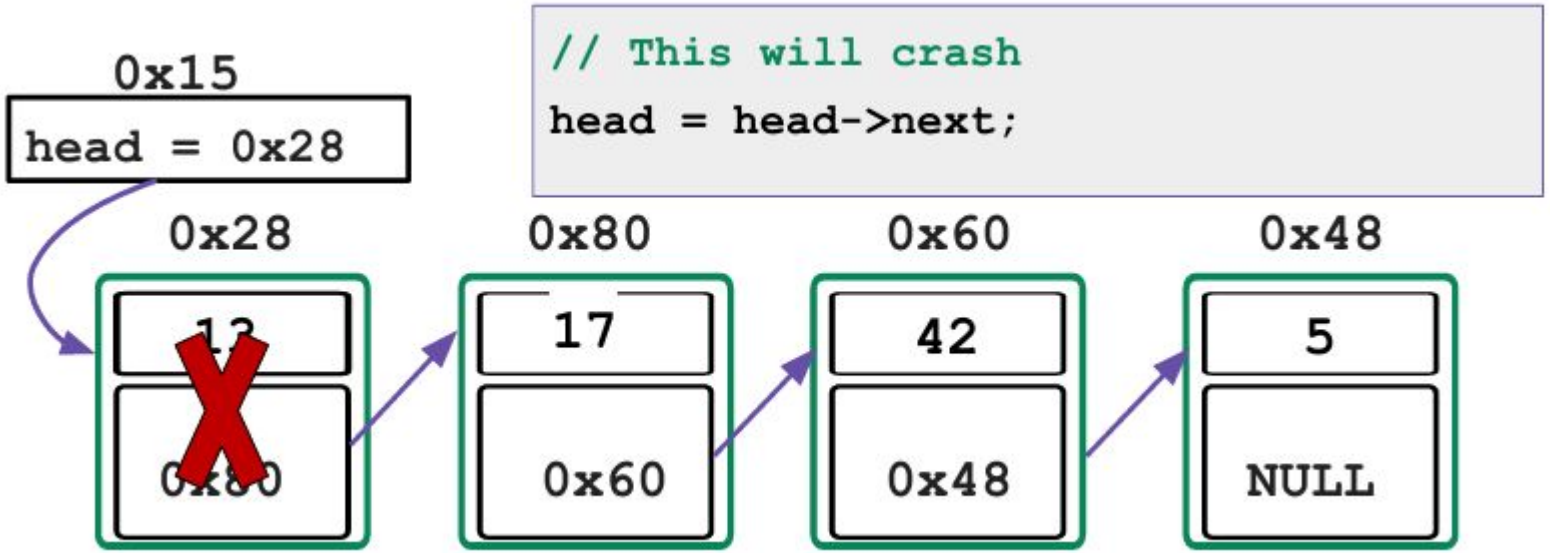
- What would be the problem calling free on head first?

```
free(head);
```



Deleting the First Node in a Linked List

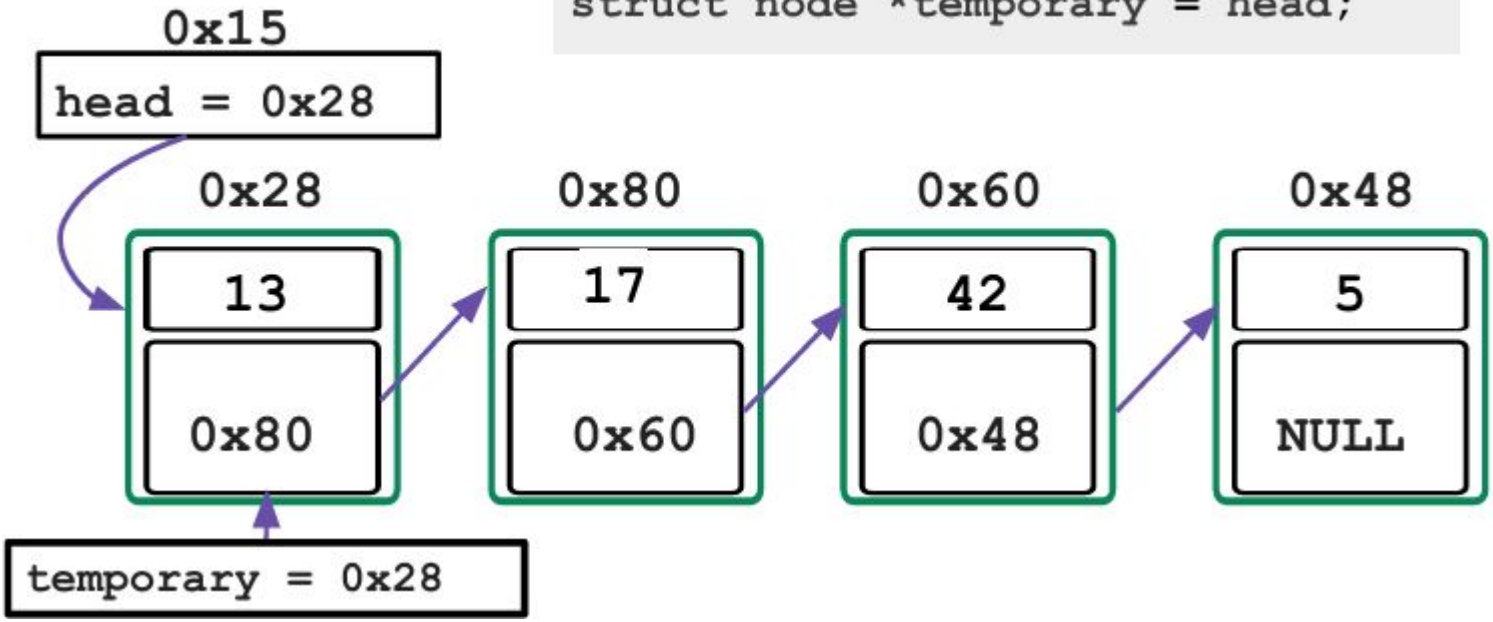
- We can’t access memory that has been freed!
  - We have lost the rest of the list



Deleting the First Node in a Linked List

- Let’s create a pointer to the first node

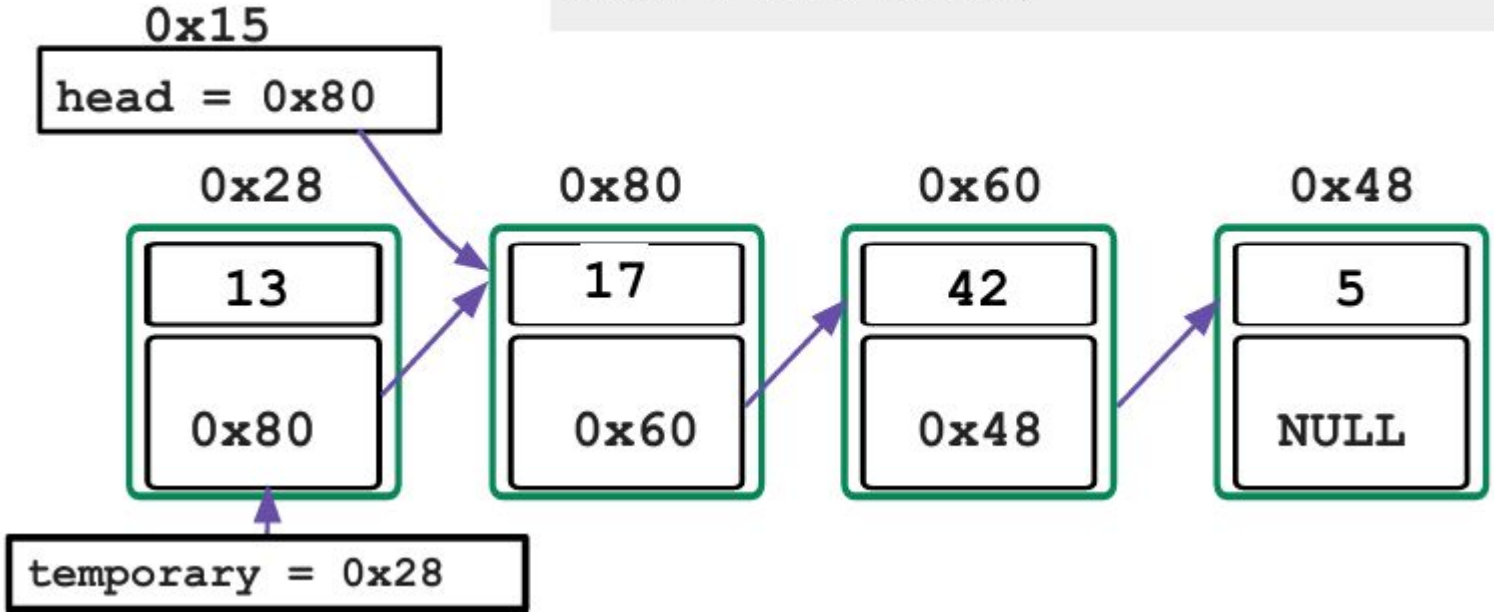
```
struct node *temporary = head;
```



Deleting the First Node in a Linked List

- Now we can update head

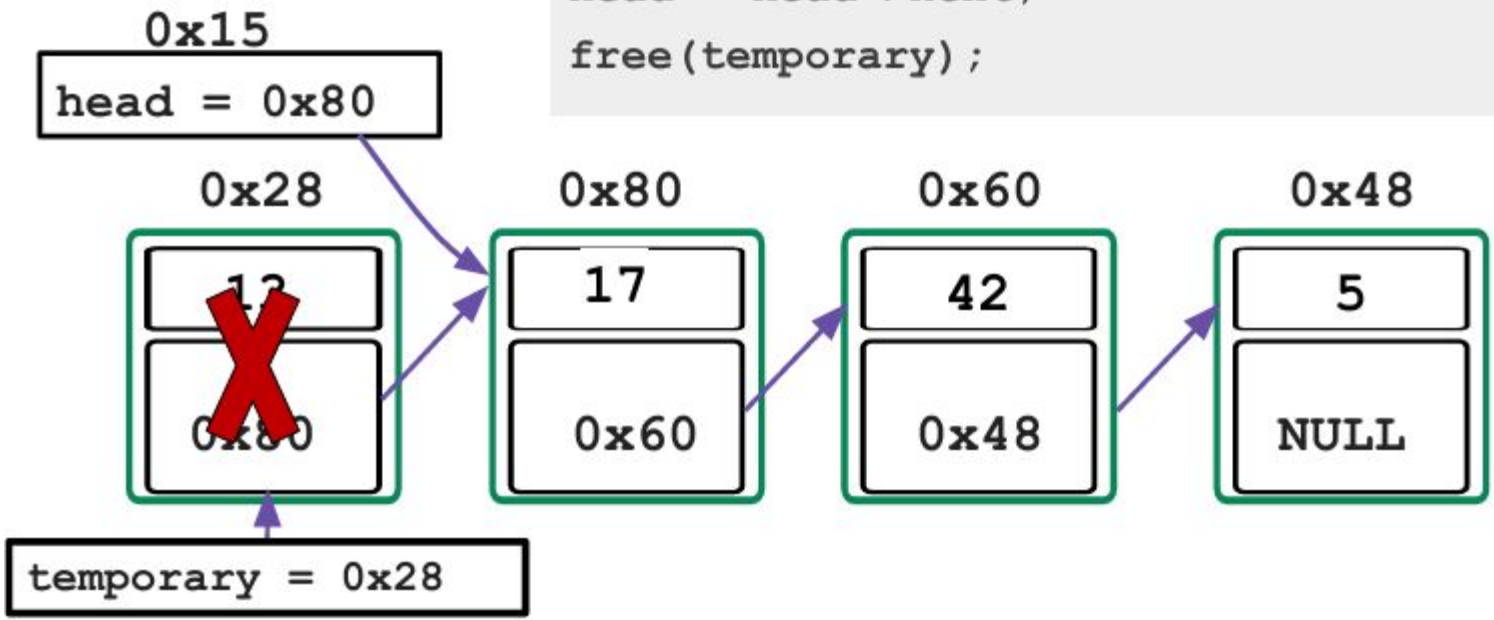
```
head = head->next;
```



Deleting the First Node in a Linked List

- Now we can free the first node

```
head = head->next;
free(temporary);
```



Deleting the First Node in a Linked List

- What should the prototype of the function look like?
- How should we call the function?
- Are there any cases we haven’t considered?

Deleting the First Node in a Linked List

- We need to consider the case where the list is empty
  - If it's empty
    - We can't delete anything
  - We just return the head of the list, which is NULL

```
if (head == NULL) {
    return head; //or return NULL;
}
```

Memory Leaks

- We malloc for every node in our lists
- For small nodes that do not run for a long time, not freeing the nodes may not cause any issues
- For programs that run for a long time
  - For example
    - Web servers (e.g. used by Youtube, or Amazon etc)
    - Game servers (e.g. Minecraft server)
    - Your own web browser or local games
  - Memory leaks can gradually increase memory usage, leading to degraded performance, crashes, restarts and potentially service outages

Free Lists the Wrong Way

- What is wrong with this code?

```
// Free all nodes in a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        free(current);
        current = current->next;
    }
}
```

Free Lists the Wrong Way

- Don’t forget, if you free memory, you can’t use it!

```
// Free all nodes in a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        free(current);
        current = current->next;
    }
}
```

Free Lists the Wrong Way

- Don’t forget, if you free memory, you can’t use it!

```
// Free all nodes in a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        free(current);
        current = current->next;
    }
}
```

Warning: Memory Use After Free

- Using memory after freeing is a serious bug
  - On dcc our program will crash at runtime and will helpfully tell us the problem
  - In production code, it can cause:
    - Crashes
    - Bugs
    - Serious security Vulnerabilities 🚨
      - Real world example: Google Chrome's WebAudio component 2019.
        - It allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page NVD - CVE-2019-13720

Free List the Correct Way

- Let's test it and check it with dcc --leak-check

```c
// Delete all nodes from a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        head = head->next;
        free(current);
        current = head;
    }
}
```

Revision Exercise

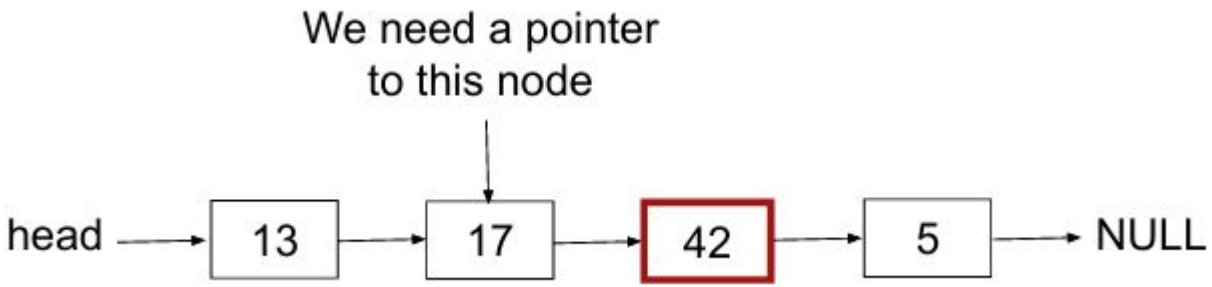
- Write a function that searches for a given value in a linked list
  - Return 1 if it exists
  - Return 0 otherwise
- What should the prototype be?
- What cases should we make sure we test?

Search and Delete

- We want to search for a node with a particular value in it and then delete it
  - Where could the item be
    - Nowhere - if it is an empty list or the list does not contain the value
    - At the head (deleting the first node in the list)
    - Between any 2 nodes in the list
    - At the tail (deleting the last node in the list)
    - There could be multiple occurrences!
      - For now let’s just consider the first occurrence

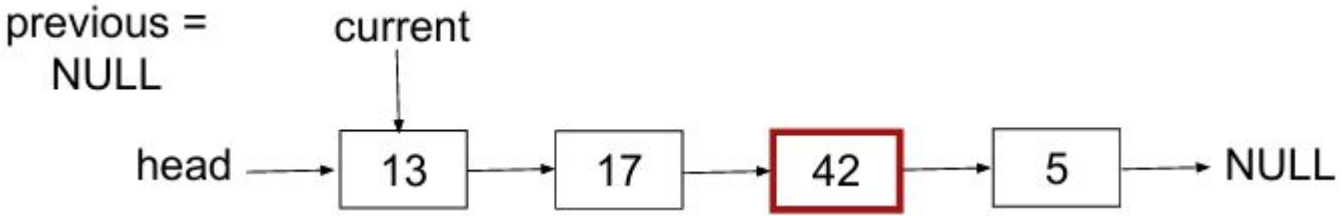
Search and Delete: Between Two Nodes

- To delete a node we need to link the previous node to the next node
  - E.g. if we want to delete the node with 42, we need to find the node before it



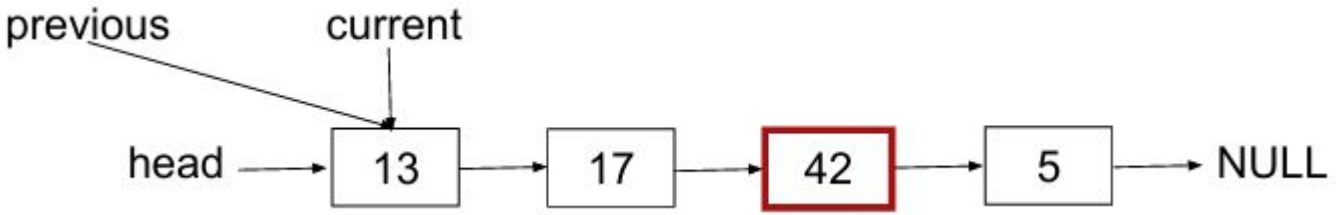
Search and Delete: Between Two Nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



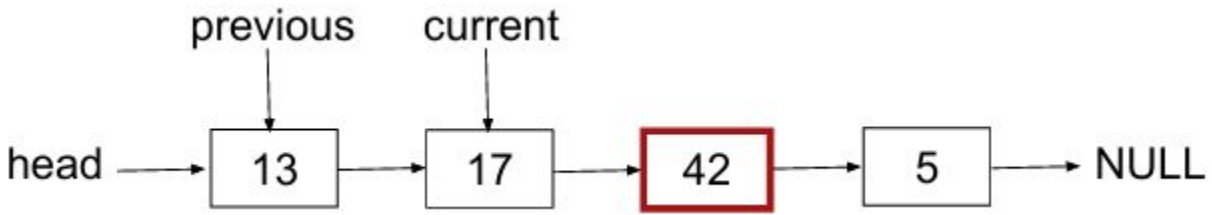
Search and Delete: Between Two Nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



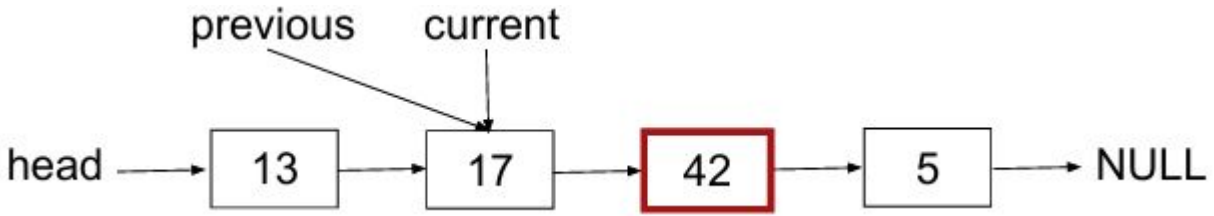
Search and Delete: Between Two Nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



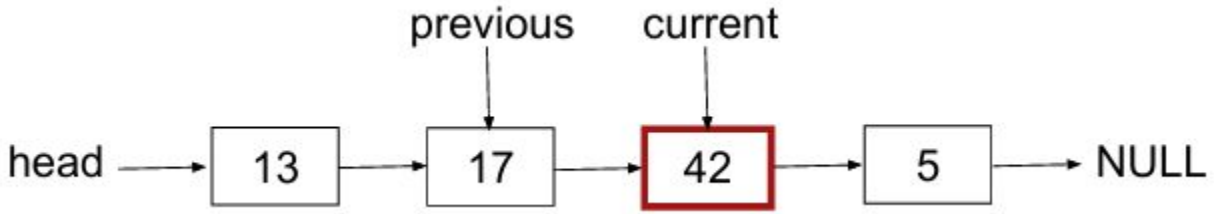
Search and Delete: Between Two Nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



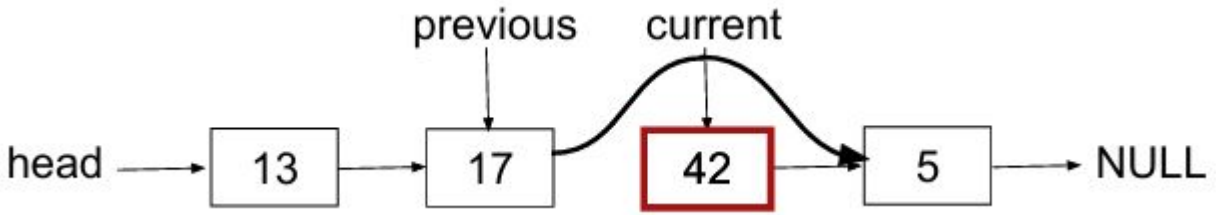
Search and Delete: Between Two Nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



Search and Delete Approach 1

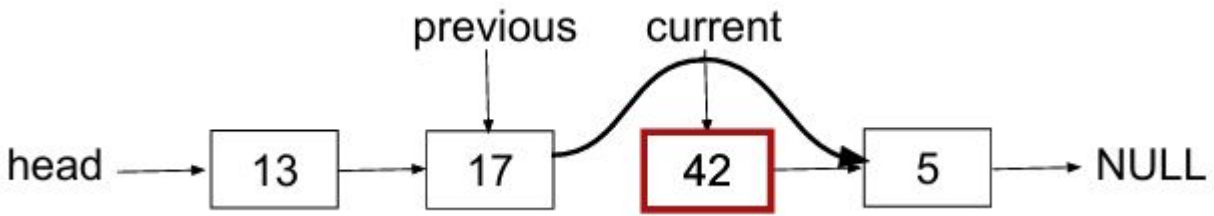
- Then we need to connect previous node to the one after the one we are deleting



Search and Delete Approach 1

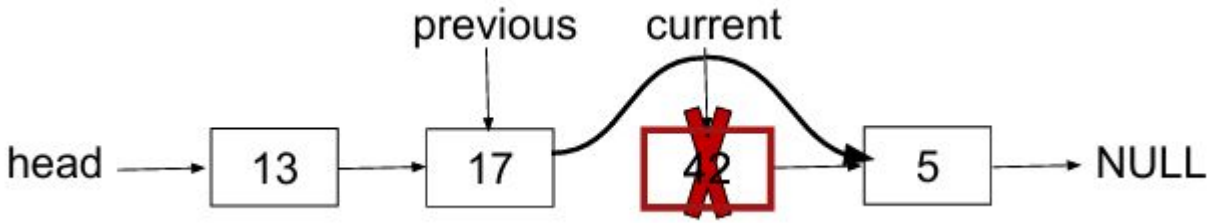
- Then we need to connect previous node to the one after the one we are deleting

```
previous->next = current->next;
```



Search and Delete Approach 1

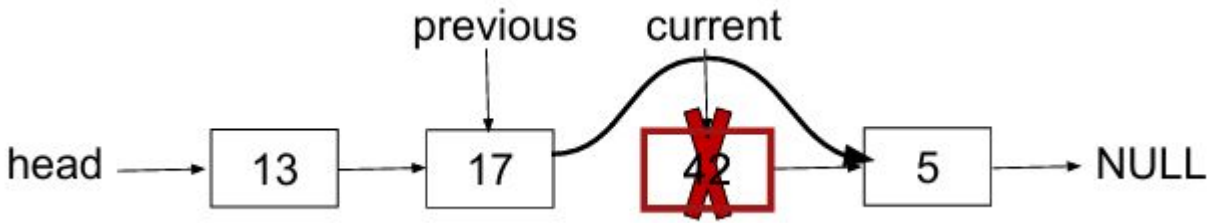
- Now we can free the node we want to delete



Search and Delete Approach 1

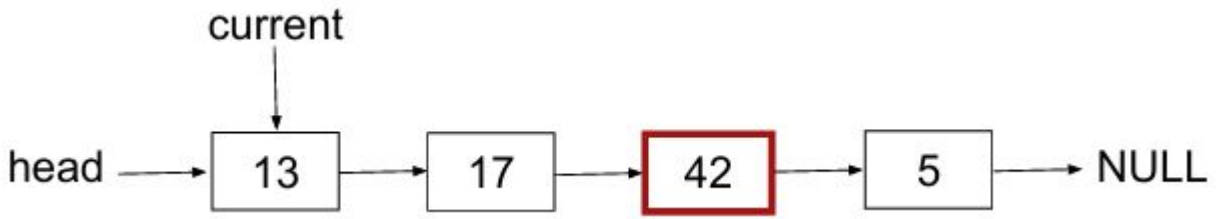
- Now we can free the node we want to delete

```
free(current);
```



Search and Delete Approach 2

```
// Approach 2: Just use 1 pointer to traverse
// but check the next node
struct node *current = head;
while (current->next != NULL &&
       current->next->data != value) {
    current = current->next;
}
```



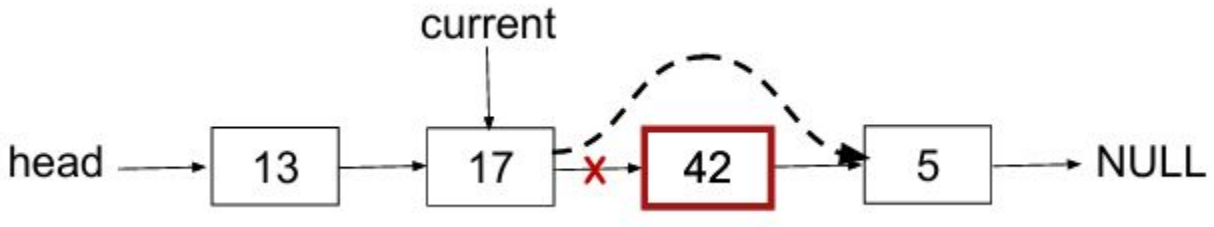
Search and Delete Approach 2: General Case

```
// Approach 2: Just use 1 pointer to traverse
// but check the next node
struct node *current = head;
while (current->next != NULL &&
       current->next->data != value) {
    current = current->next;
}
```



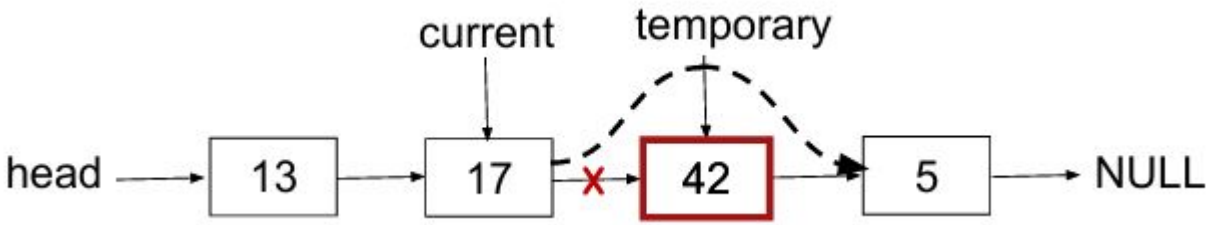
Search and Delete Approach 2

- Then we need to connect current node to the one after the one we're deleting
  - But we still need a pointer to the node we want to free, how can we do this?



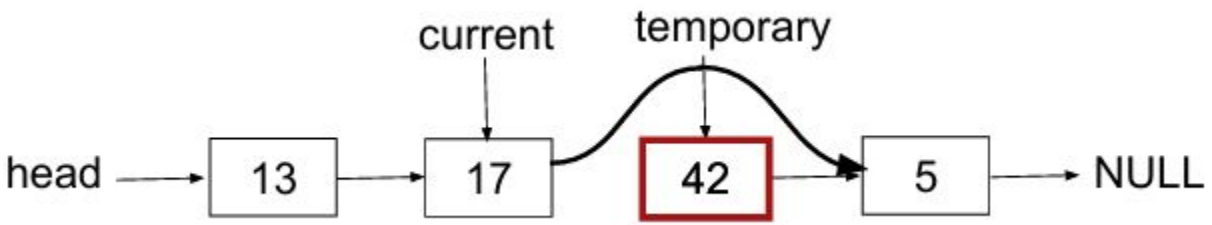
Search and Delete: Approach 2

```
struct node *temporary = current->next;
```



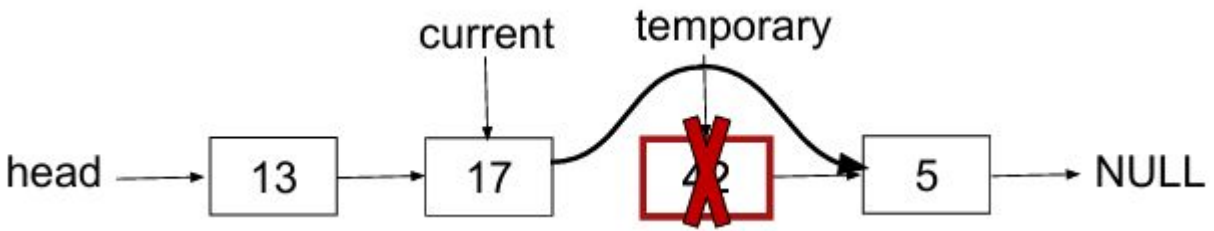
Search and Delete: Approach 2

```
struct node *temporary = current->next;
current->next = temporary->next;
```



Search and Delete: Approach 2

- Now we can free the node we want to delete



Coding

- Let’s code up these approaches
- Extension
  - Make them delete all occurrences

What We Learnt

- Recap on linked lists basic
- Recap on inserting linked lists into a position
- Deleting elements
  - First node
  - Freeing all nodes
  - Search and delete
  - Issues
    - Memory leaks
    - Accessing memory after being freed

Next Time

- More searching and deleting
  - Deleting from anywhere in a linked list recapped/continued
- Linked Lists - A Larger Application
  - Linked lists as members in other structs
  - Linked lists with more complex data types
  - Helpful for Assignment 2!

Reach Out

- Check the course forum for questions!
  - Post there too!
- Admin questions
  - cs1511@unsw.edu.au

Thank You

Questions?