LINKED LIST OPERATIONS

COP 3502

Linked List Operations

- There are several basic operations that need to be performed on linked lists:
  1) Adding a node.
  2) Deleting a node.
  3) Searching for a node.
  - We will build functions to perform these operations.
- There are of course many other operations you could do:
  - Counting nodes, modifying nodes, reversing the list, and more.
  - We can also build functions for these.

Linked List Operations

- Design
  - Functions that change the contents of lists (i.e. insertion and deletion) will return the head pointer.
    - For, example: head = insertNode(head, 12);
    - Why must we return the head pointer?
      - If the first node in the list has changed inside the insertNode function we need our head pointer to reflect these changes.
      - If the head pointer doesn’t change within the function, then head is just reset to its original address.

Linked List Operations

- Design
  - Functions that do not change the contents of the list, return values according to their purpose.
    - For example, if we want to search for a node and return 0 or 1 if it’s found.
    - Or if we want to count the number of nodes in our list.
  - And some functions that process the entire list are void, such as functions that print the list.

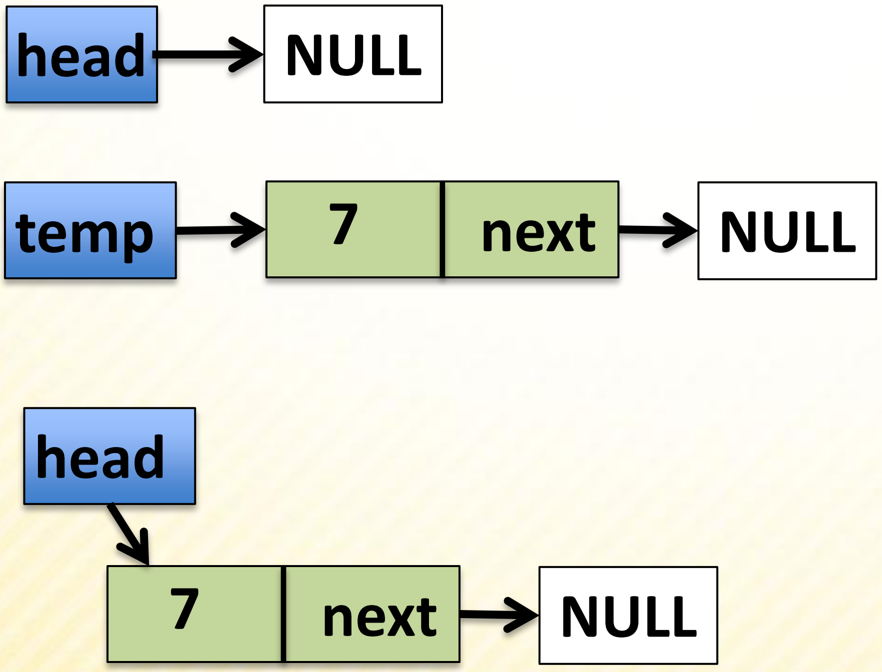
Linked Lists: Insert In Order

- Let's implement a function that will insert a node in order into our linked list.
  - Useful if we want to keep a sorted list (useful for HW#2)
- The cases we will have to check for are:
  1) The list is empty
  2) The element is less than the first node
  3) The element is inserted into the middle of our list
  4) The element is inserted at the end of our list.
     - We already know how to do cases 1,2, and 4!
     - And really we're going to merge case 3 and 4, so this should be pretty easy for us!

Linked Lists: Insert In Order

Case 1) The list is empty:

- Create the new node, and if the list is empty return the new node.
- Simple!

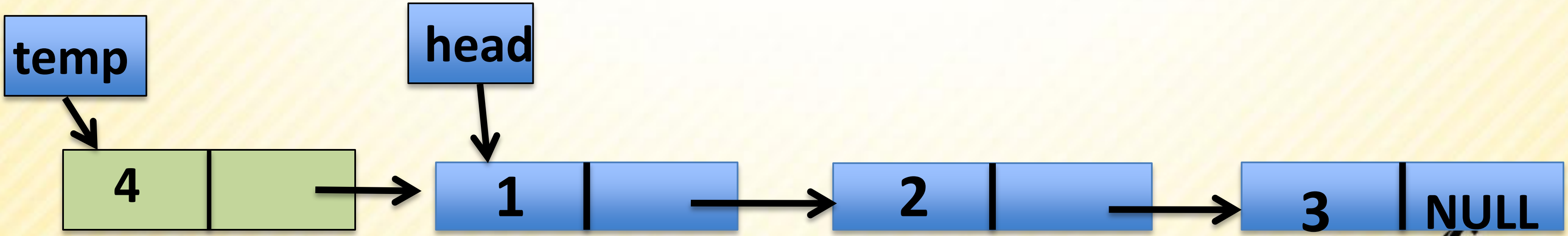


Linked Lists: Insert In Order

Case 2) The element is < the head:

- In this case we want to add the element to the front of the list
- We already know how to do this!

1) Create the new node
2) Set the new node’s next to head
3) Return temp.



temp

head

4

1

2

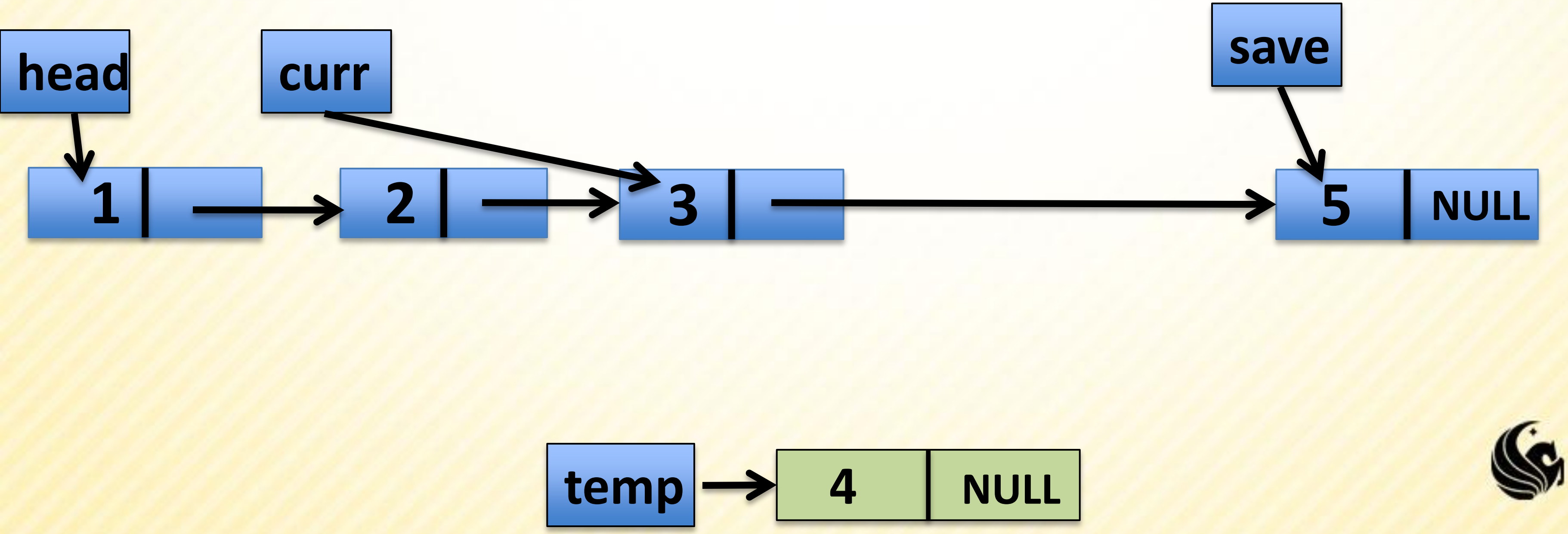
3

NULL

Linked Lists: Insert In Order

Case 3/4) Insert the element in the middle or end of our list.

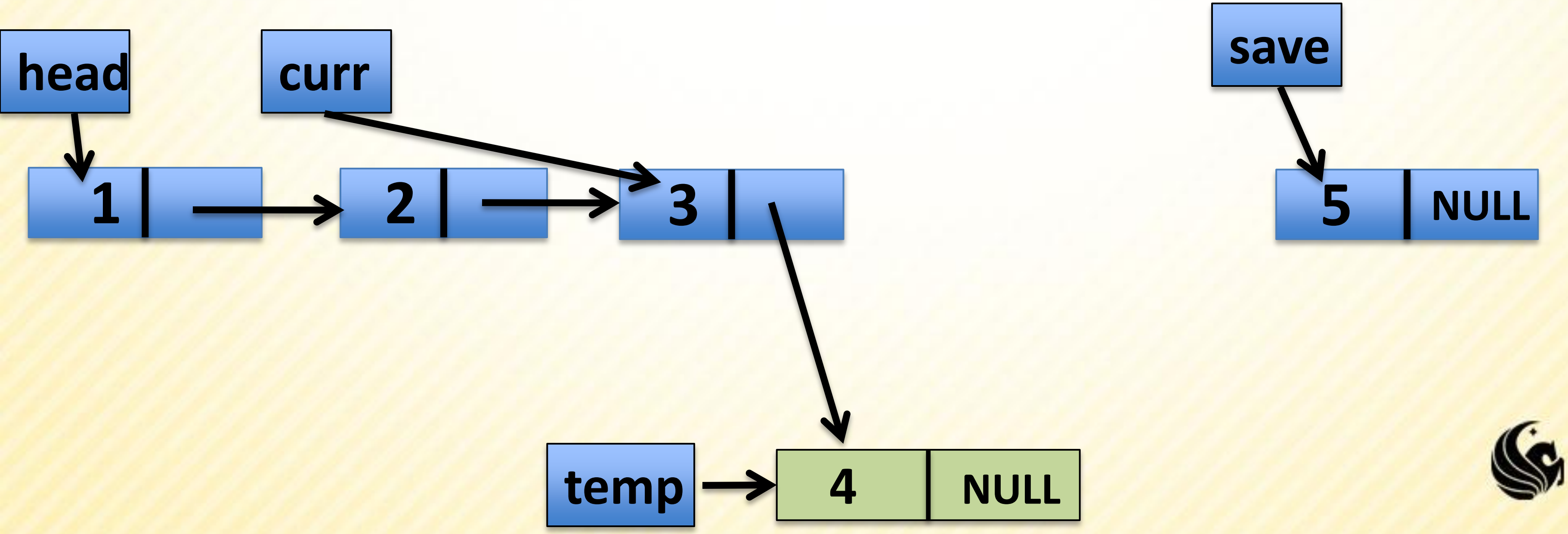
- In this case we need to traverse the list while our element is less than the curr element.
- Then we add the element after the curr and before curr->next;



Linked Lists: Insert In Order

Case 3/4) Insert the element in the middle or end of our list.

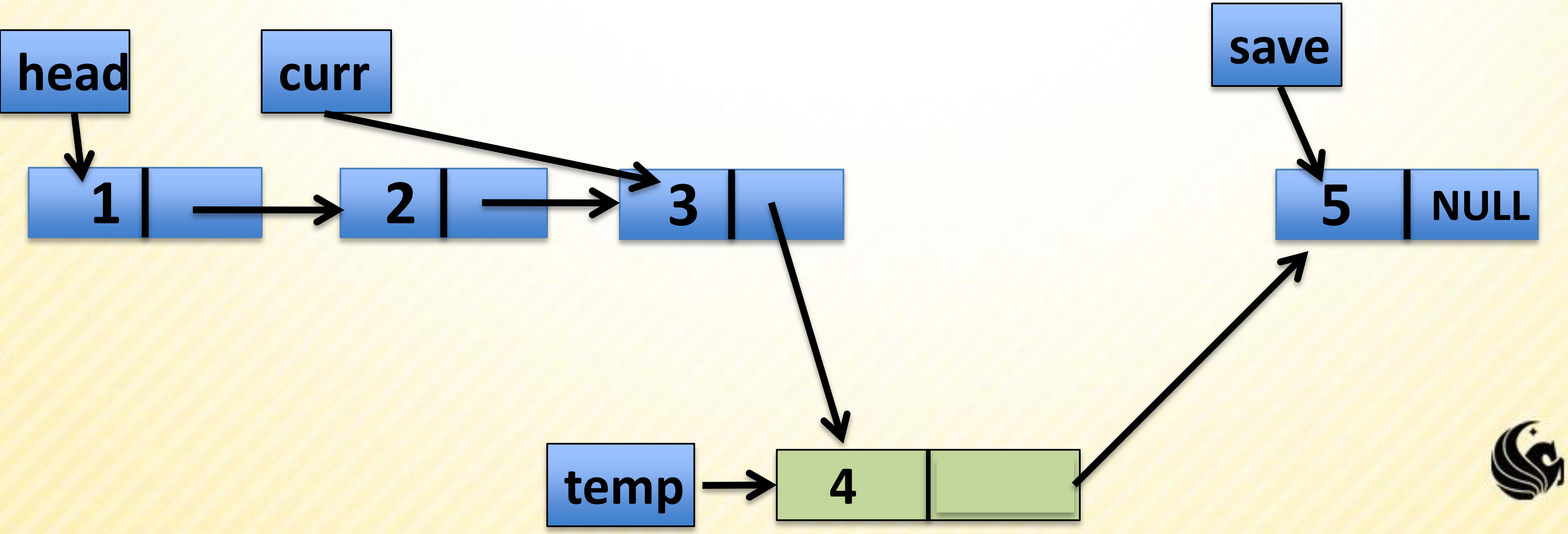
- In this case we need to traverse the list while our element is less than the curr element.
- Then we add the element after the curr and before curr->next;



Linked Lists: Insert In Order

Case 3/4) Insert the element in the middle or end of our list.

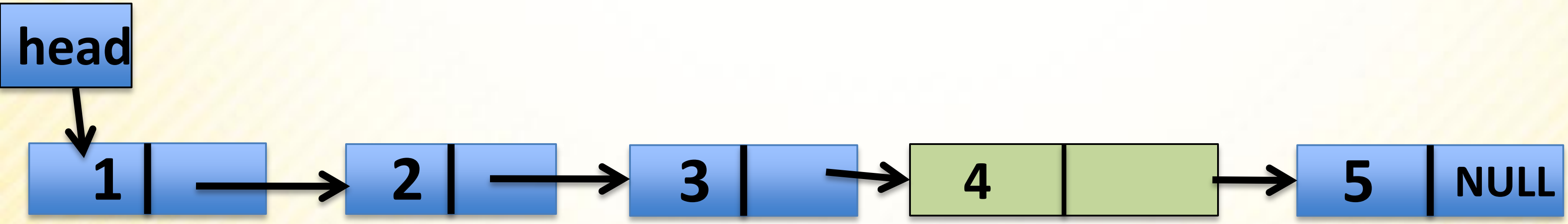
- In this case we need to traverse the list while our element is less than the curr element.
- Then we add the element after the curr and before curr->next;



Linked Lists: Insert In Order

Case 3/4) Insert the element in the middle or end of our list.

- In this case we need to traverse the list while our element is less than the curr element.
- Then we add the element after the curr and before curr->next;



```
struct node* InsertInorder(node *head, int num) {
       // Create the new node

       // Case 1: Inserting into an empty list.

       // Case 2: Element is < the front

       // Case 3/4: Insert element in the middle/end
       // Use curr to traverse to the right spot
       // to insert temp.
       // Save the node to temp should point to.
       // Insert temp.
       // Return a pointer to the front of the list.
}
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      node *temp = (node*)malloc(sizeof(node));
      temp->data = num;
      temp->next = NULL;

      // Case 1: Inserting into an empty list.

      // Case 2: Element is < the front

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      // Save the node to temp should point to.
      // Insert temp.
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      node *temp = (node*)malloc(sizeof(node));
      temp->data = num;
      temp->next = NULL;

      // Case 1: Inserting into an empty list.
      if (front == NULL)    return temp;

      // Case 2: Element is < the front

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      // Save the node to temp should point to.
      // Insert temp.
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      node *temp = (node*)malloc(sizeof(node));
      temp->data = num;
      temp->next = NULL;

      // Case 1: Inserting into an empty list.
      if (front == NULL)    return temp;

      // Case 2: Element is < the front
      if (num < front->data) {
            temp->next = front;
            return temp;
      }
      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      // Save the node to temp should point to.
      // Insert temp.
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      // Case 1: Inserting into an empty list.
      // Case 2: Element is < the front

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      // Save the node to temp should point to.
      // Insert temp.
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      // Case 1: Inserting into an empty list.
      // Case 2:

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      node *curr = head;
      while(curr->next != NULL &&
             curr->data < temp->data)
            curr = curr->next;

      // Save the node to temp should point to.
      // Insert temp.
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      // Case 1: Inserting into an empty list.
      // Case 2: Element is < the front

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      node *curr = head;
      while(curr->next != NULL &&
            curr->data < temp->data)
           curr = curr->next;

      // Save the node to temp should point to.
       node *save = curr->next;
      // Insert temp.
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      // Case 1: Inserting into an empty list.
      // Case 2: Element is < the front

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      node *curr = head;
      while(curr->next != NULL &&
            curr->data < temp->data)
           curr = curr->next;

      // Save the node to temp should point to.
       node *save = curr->next;
      // Insert temp.
      curr->next = temp;
      temp->next = save;
      // Return a pointer to the front of the list.
```

```
struct node* InsertInorder(node *head, int num) {
      // Create the new node
      // Case 1: Inserting into an empty list.
      // Case 2: Element is < the front

      // Case 3/4: Insert element in the middle/end
      // Use curr to traverse to the right spot
      // to insert temp.
      node *curr = head;
      while(curr->next != NULL &&
            curr->data < temp->data)
           curr = curr->next;

      // Save the node to temp should point to.
       node *save = curr->next;
      // Insert temp.
      curr->next = temp;
      temp->next = save;
      // Return a pointer to the front of the list.
      return head;
}
```

Deleting Nodes

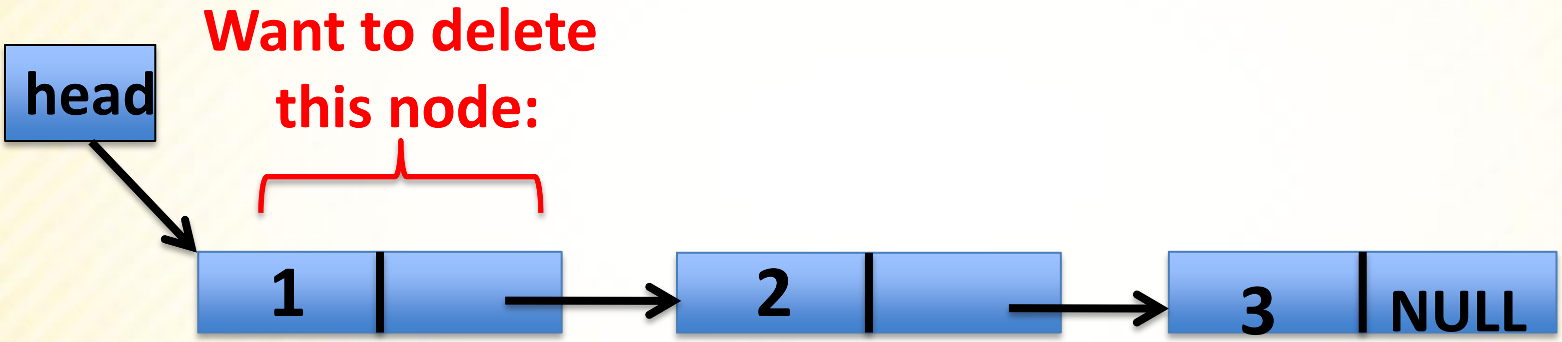
- General Approach:
  1) Search for the node you want to delete
  2) If found, delete the node from the list
  3) To delete, you must make sure:
     - The **predecessor** of the deleted node points to the deleted node's **successor**
  4) Finally, free the node
     - e.g. the node is physically removed from the heap memory.

Deleting Nodes

- There are 4 cases we need to deal with:
  1) Delete the 1st node of a list.
  2) Delete any middle node of a list (not the first or the last)
  3) Delete the last node of the list.
  4) We delete the ONLY node in the list.
     - The resulting list is then empty.

Deleting Nodes

- Case 1) Delete the 1st node of a list



Want to delete this node:

head

1

2

3 NULL

head

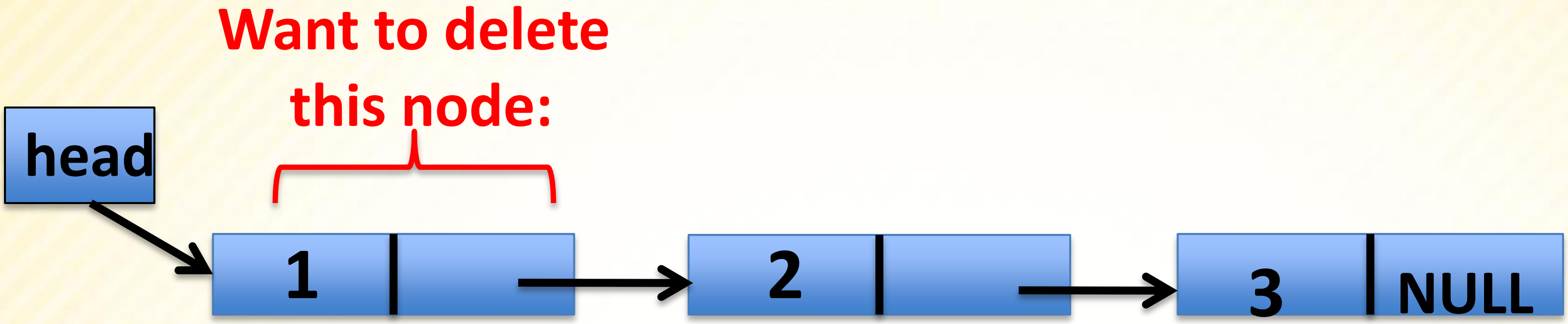
Resulting List:

2

3 NULL

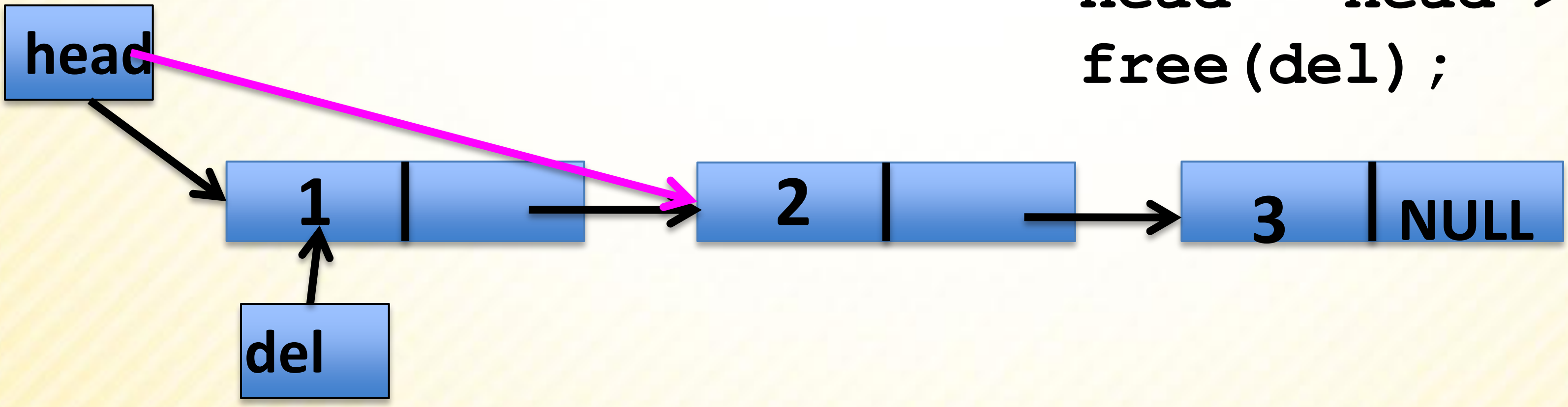
Deleting Nodes

- Case 1) Delete the 1st node of a list

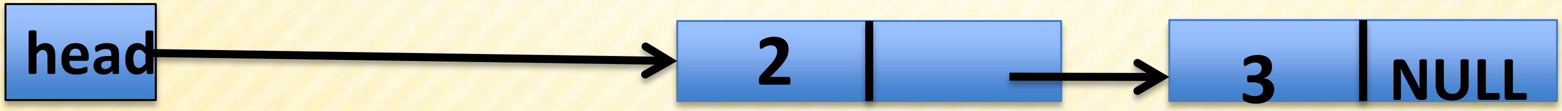


Want to delete this node:

```
node *del = head;
head = head->next;
free(del);
```

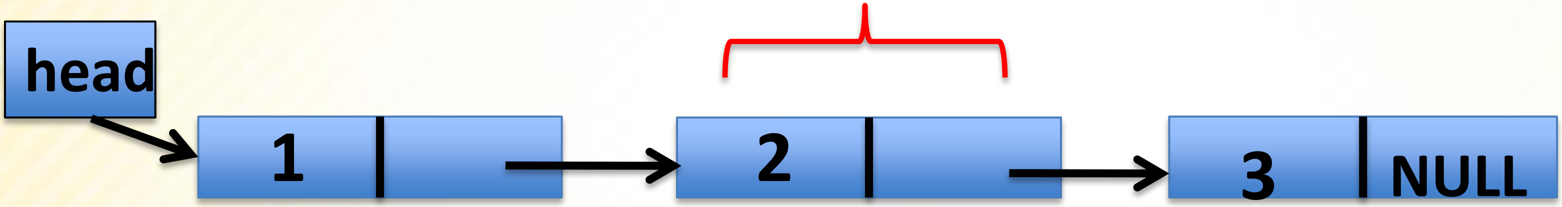


Resulting List:



Deleting Nodes

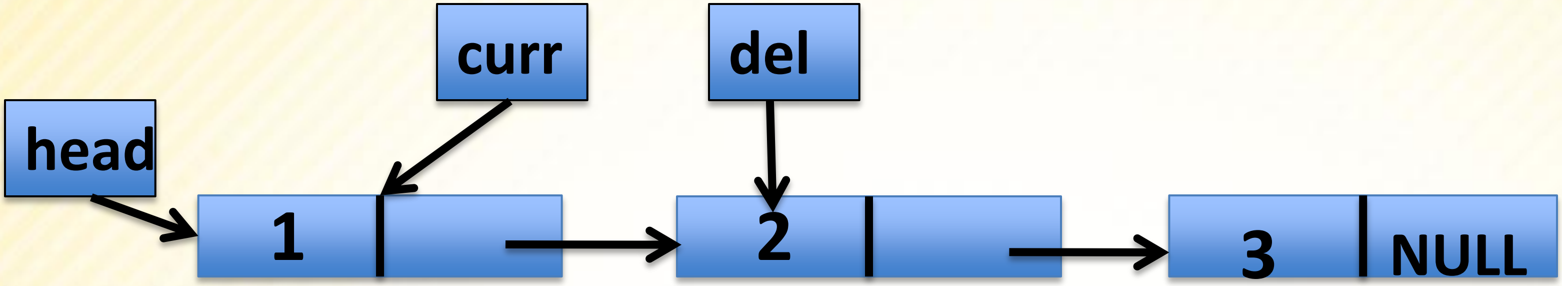
- Case 2) Delete the middle node of a list



Want to delete this node:

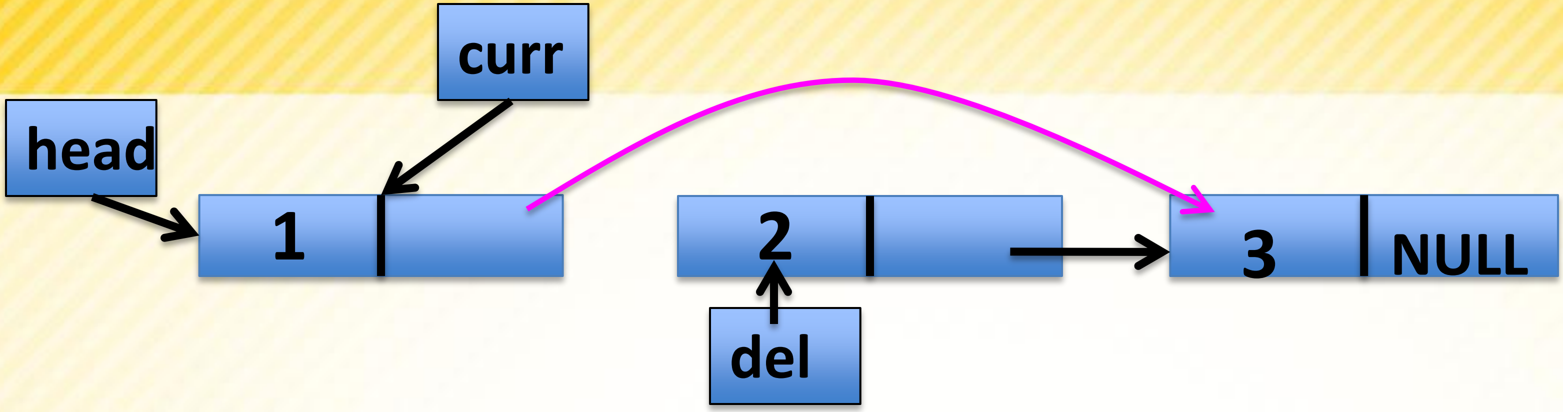
Deleting Nodes

- Case 2) Delete the middle node of a list



```
node *curr = head;
// Traverse the list until curr->next == val
while (curr->next != NULL) {
     if (curr->next->data == val) {
          node *del = curr->next;
          curr->next = curr->next->next;
          free(del);
          return head;
     }
     curr = curr->next;
}
```

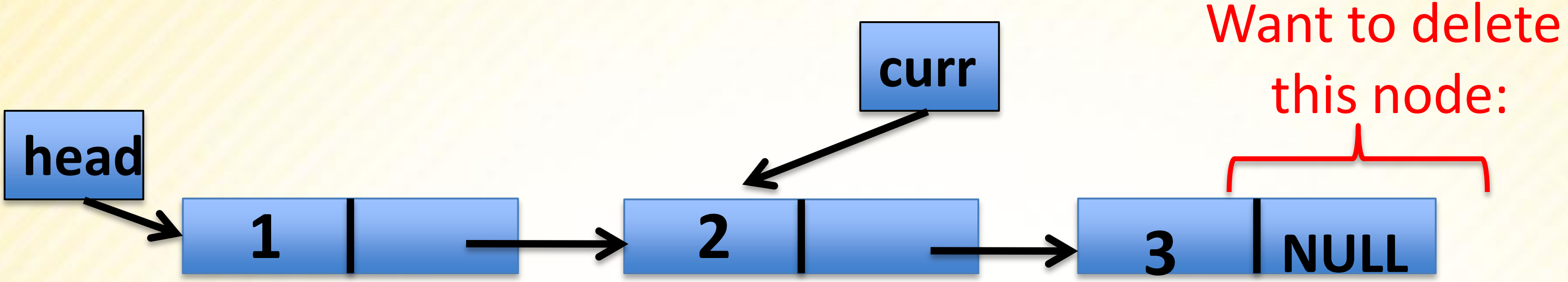
Case 2) Delete the middle node of a list


curr
head
1
2
3
NULL
del

```
node *curr = head;
// Traverse the list until curr->next == val
while (curr->next != NULL) {
     if (curr->next->data == val) {
          node *del = curr->next;
          curr->next = curr->next->next;
          free(del);
          return head;
     }
     curr = curr->next;
}
```

Deleting Nodes

- Case 3) Delete the last node of a list



Want to delete this node:

curr

head

1

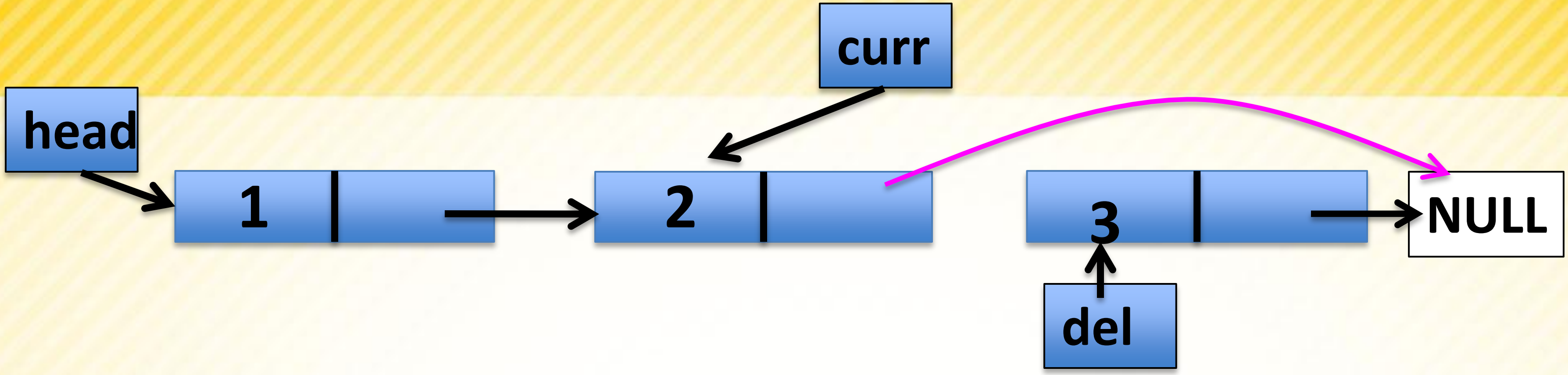
2

3

NULL

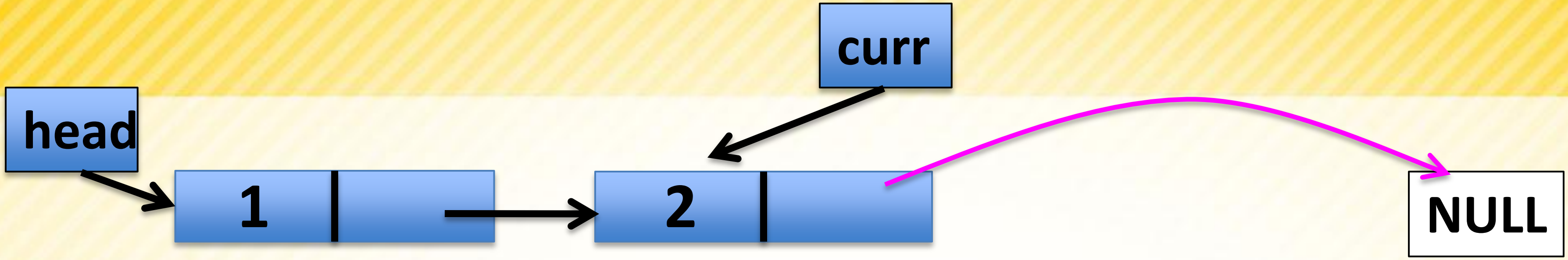
```
node *curr = head;
// Traverse the list until curr->next == val
while (curr->next != NULL) {
     if (curr->next->data == val) {
          node *del = curr->next;
          curr->next = curr->next->next;
          free(del);
          return head;
     }
     curr = curr->next;
}
```

Case 3) Delete the last node of a list



```
node *curr = head;
// Traverse the list until curr->next == val
while (curr->next != NULL) {
     if (curr->next->data == val) {
          node *del = curr->next;
          curr->next = curr->next->next;
          free(del);
          return head;
     }
     curr = curr->next;
}
```

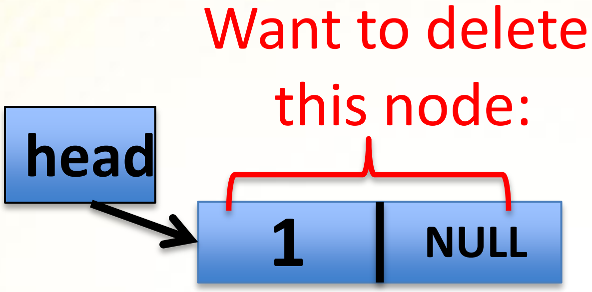
- Case 3) Delete the last node of a list



```
node *curr = head;
// Traverse the list until curr->next == val
while (curr->next != NULL) {
     if (curr->next->data == val) {
          node *del = curr->next;
          curr->next = curr->next->next;
          free(del);
          return head;
     }
     curr = curr->next;
}
```

Deleting Nodes

- Case 4) Delete the ONLY node of a list



```
// We want to:
free(head);
return NULL;
```

But this will fit in with case #1;

```
node* delete(node *head, int num) {
   if (head == NULL) return head;

   // Case 1/4: Delete 1st node, or ONLY node

   // Case 2/3: Delete middle/last node

   // Loop until you find node to delete

   // We didn’t find it, so return original head
   return head;
}
```

```
node* delete(node *head, int num) {
    if (head == NULL) return head;

    // Case 1/4: Delete 1st node, or ONLY node
    node *curr = head;
    if (curr->data == num) {
        node *temp = curr->next;
        free(curr);
        return temp;
    }

    // Case 2/3: Delete middle/last node

    // Loop until you find node to delete

    // We didn't find it, so return original head
    return head;
}
```

```
node* delete(node *head, int num) {
   if (head == NULL) return head;

   // Case 1/4: Delete 1st node, or ONLY node
   // …

   // Case 2/3: Delete middle/last node
   // Loop until you find node to delete
   node *curr = head;
   while (curr >next != NULL) {
         if (curr ->next->data == num) {
            node *del = curr -> next;
            curr >next = curr ->next->next;
            free(del);
            return front;
         }
         curr = curr->next;
   }
   // We didn't find it, so return original head
   return head;
}
```

Deleting the Entire List

- head = freeList(head);

```
node* freeList(node *head) {
   node *curr = head;
   while (curr != NULL) {
      node *temp = curr;
      curr = curr->next;
      free(temp);
   }
   return NULL;
}
```

Linked List Practice Problem

- Write a recursive function that deletes every other node in the linked list pointed to by the input parameter *head*. (Specifically, the 2nd 4th 6th etc. nodes are deleted)

```
void delEveryOther(node* head){









}
```