LINKED LIST VARIATIONS

COP 3502

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)
  - From Fall 2009 Foundation Exam

```
void delEveryOther(node* head){
      if (head == NULL || head->next == NULL) return;

      node *temp = head->next;

      head->next = temp->next;

      free(temp);

      delEveryOther(head->next);
}
```

Linked List Practice Problem

- Write an iterative 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)
  - From Fall 2009 Foundation Exam

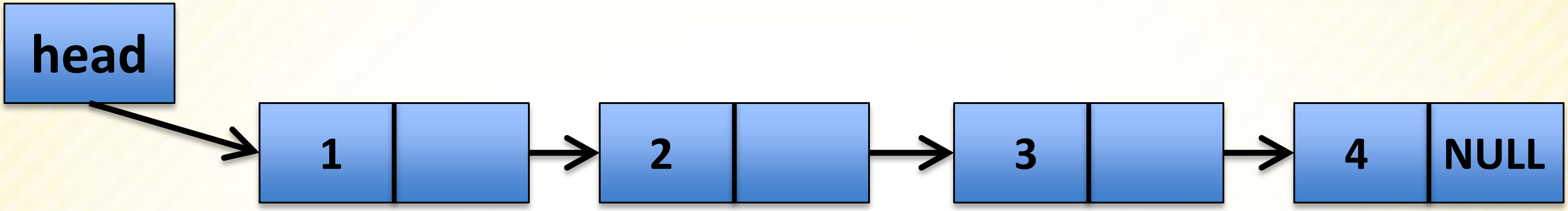
```
void delEveryOther(struct node *head) {
      struct node* curr = head;

      while(curr != NULL && curr->next != NULL) {
            struct ll* temp = curr->next;
            curr->next = temp->next;
            curr=temp->next;
            free(temp);
      }
}
```

Linked List Variations

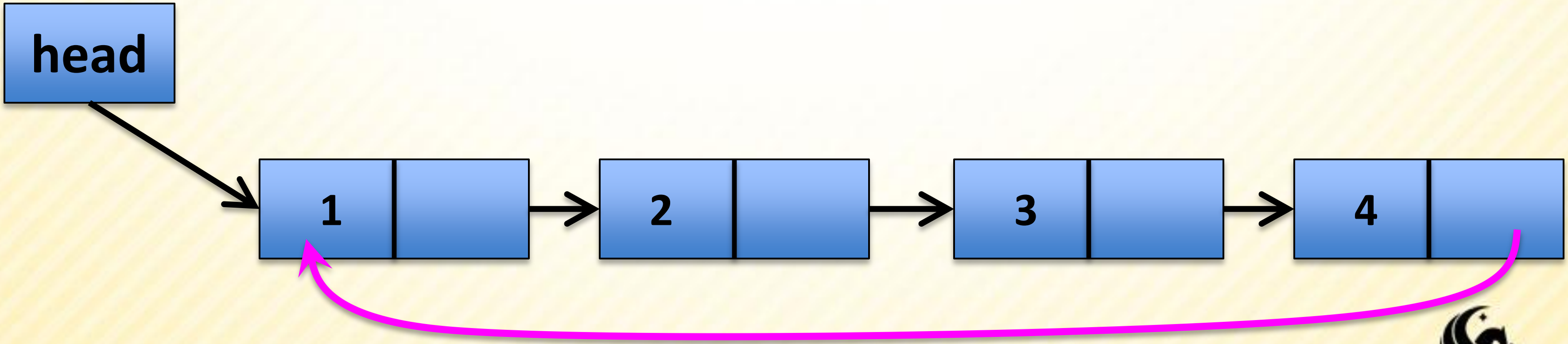
- There are 3 basic types of linked lists:
  - Singly-linked lists
  - Doubly-Linked Lists
  - Circularly-Linked Lists
- We can also have a linked lists of linked lists

Circularly Linked Lists

- Singly Linked List:

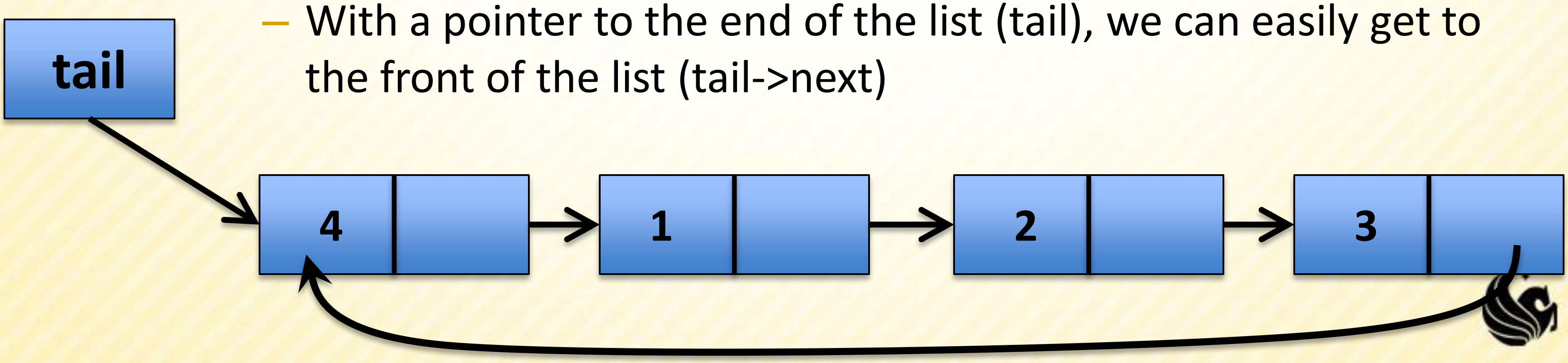


- Circularly-Linked List



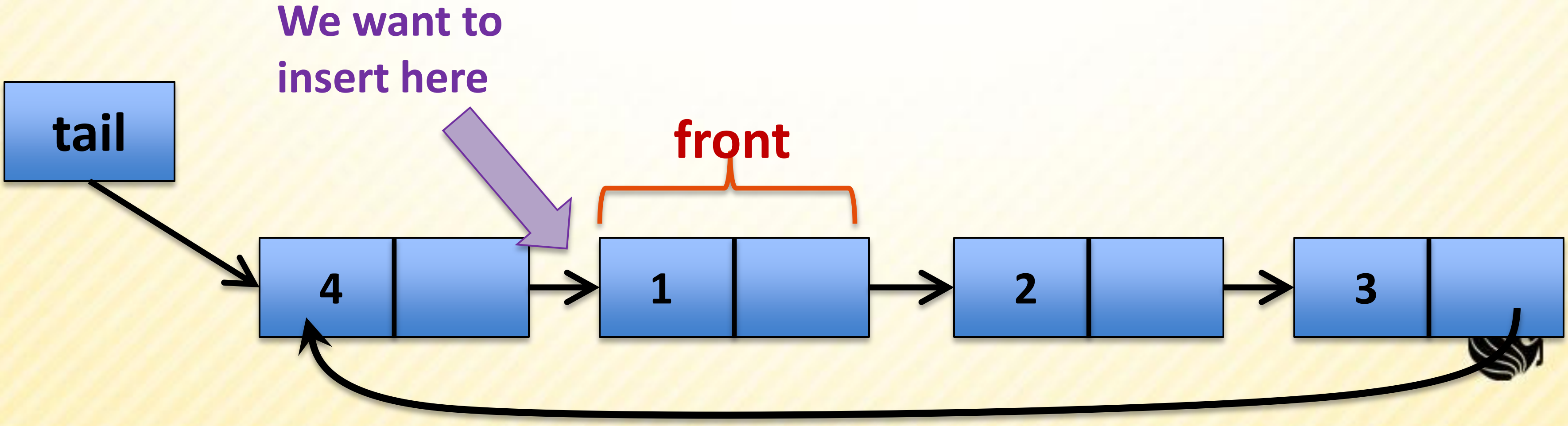
Circularly Linked Lists

- Why use a Circularly Linked List?
  - It may be a natural option for lists that are naturally circular, such as the corners of a polygon
  - OR you may wish to have a queue, where you want easy access to the front and end of your list.
    - For this reason, most circularly linked lists are implemented as follows:
      - With a pointer to the end of the list (tail), we can easily get to the front of the list (tail->next)



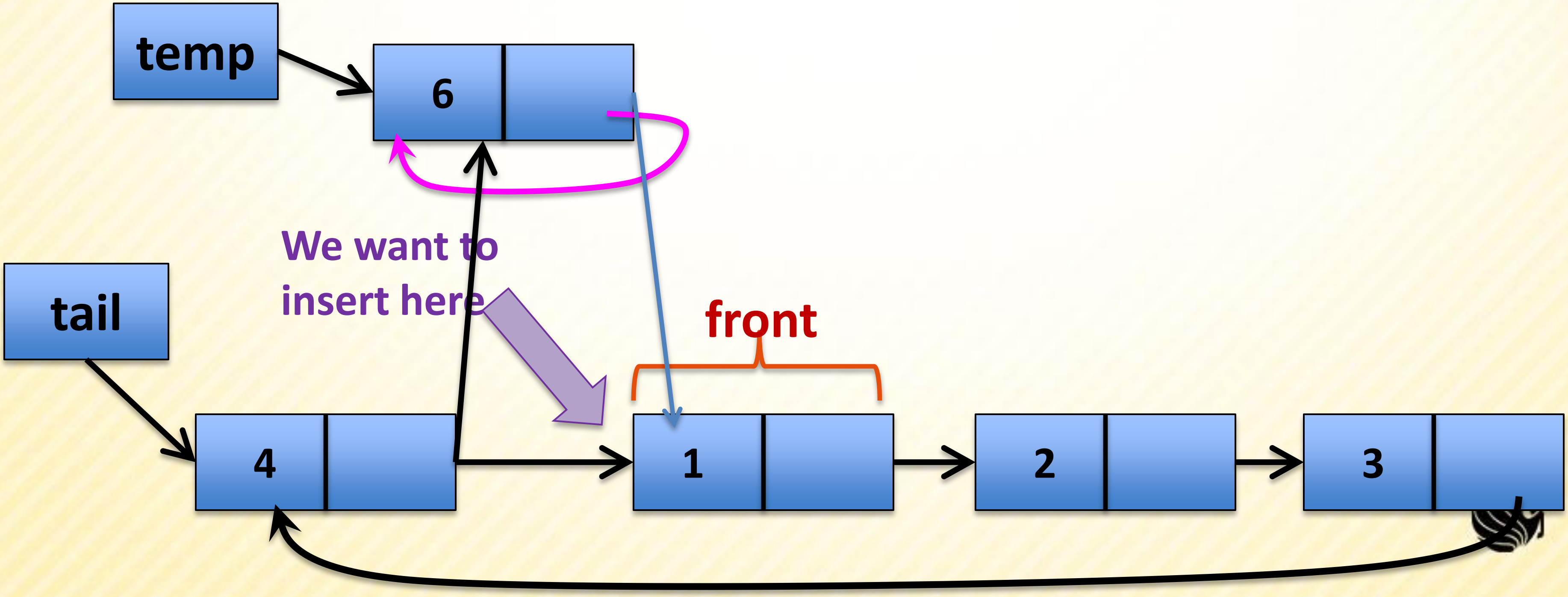
Circularly Linked List

- Consider inserting to the front of a circular linked list:
  - The first node is the node next to the tail node
  - We want to insert the new node between the tail node and the first node.



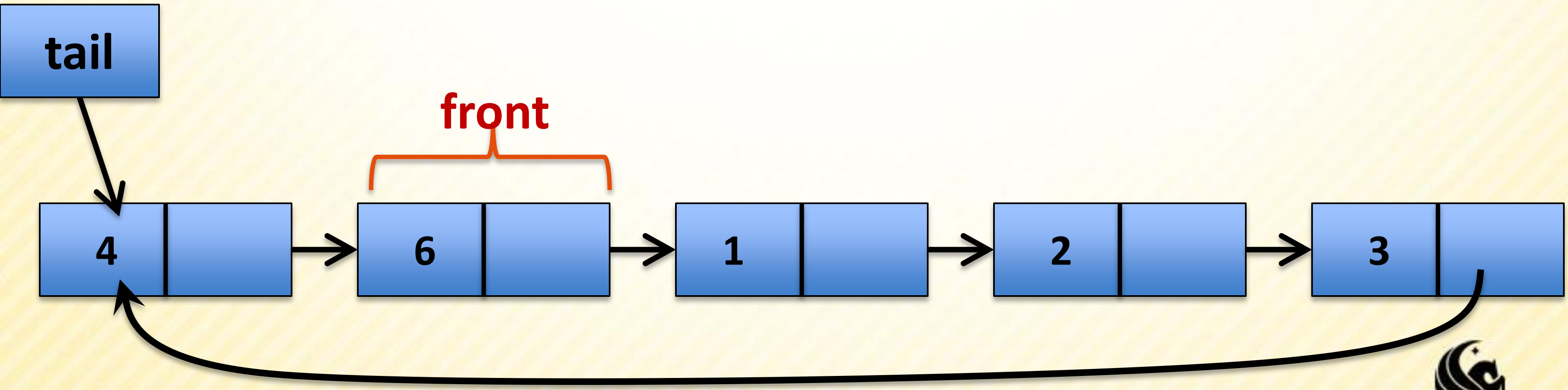
Circularly Linked List

- Steps:
  - Create a new node in memory, set its data to val
  - Make the node point to itself
  - if tail is empty, then return this node, it's the only one in the list
  - If it's not the only node, then it's next is tail->next
  - and tail->next should now point to the new node.



Circularly Linked List

- Steps:
  - Create a new node in memory, set its data to val
  - Make the node point to itself
  - if tail is empty, then return this node, it's the only one in the list
  - If it's not the only node, then it's next is tail->next
  - and tail->next should now point to the new node.
- ***Resulting List:***



Circularly Linked List

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

```
node* AddFront(node* tail, int val) {
    // Create the new node

    // Set the new node's next to itself (circular!)

    // If the list is empty, return new node

    // Set our new node's next to the front

    // Set tail's next to our new node

    // Return the end of the list
}
```

Circularly Linked List

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

```
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)

    // If the list is empty, return new node

    // Set our new node's next to the front

    // Set tail's next to our new node

    // Return the end of the list
}
```

Circularly Linked List

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

```
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)
    temp->next = temp;

    // If the list is empty, return new node

    // Set our new node's next to the front

    // Set tail's next to our new node

    // Return the end of the list
}
```

```
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)
    temp->next = temp;

    // If the list is empty, return new node
    if (tail == NULL) return temp;

    // Set our new node's next to the front

    // Set tail's next to our new node

    // Return the end of the list
}
```

```
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)
    temp->next = temp;

    // If the list is empty, return new node
    if (tail == NULL) return temp;

    // Set our new node's next to the front
    temp->next = tail->next;

    // Set tail's next to our new node

    // Return the end of the list
}
```

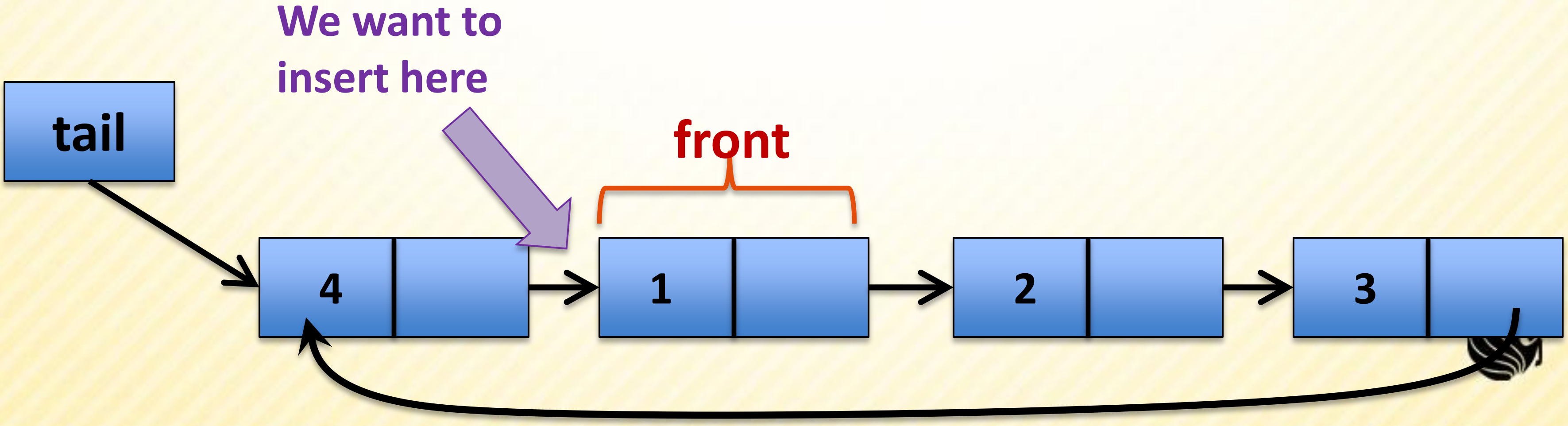
```
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)
    temp->next = temp;

    // If the list is empty, return new node
    if (tail == NULL) return temp;

    // Set our new node's next to the front
    temp->next = tail->next;

    // Set tail's next to our new node
    tail->next = temp;

    // Return the end of the list
}
```

```
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)
    temp->next = temp;

    // If the list is empty, return new node
    if (tail == NULL) return temp;

    // Set our new node's next to the front
    temp->next = tail->next;

    // Set tail's next to our new node
    tail->next = temp;

    // Return the end of the list
}
```

```c
node* AddFront(node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;

    // Set the new node's next to itself (circular!)
    temp->next = temp;

    // If the list is empty, return new node
    if (tail == NULL) return temp;

    // Set our new node's next to the front
    temp->next = tail->next;

    // Set tail's next to our new node
    tail->next = temp;

    // Return the end of the list
    return tail;
}
```

Circularly Linked List

- Inserting a node at the ***End*** of a Circular Linked List
  - The new node will be placed just after the tail node
    - (which is the last node in the list)
  - So again the new node will be inserted between the tail node and the front node.
  - The only difference with AddFront, is that now we need to change where tail points after we add the node.
    - That's the only difference, so the code is pretty similar.



Circularly Linked List

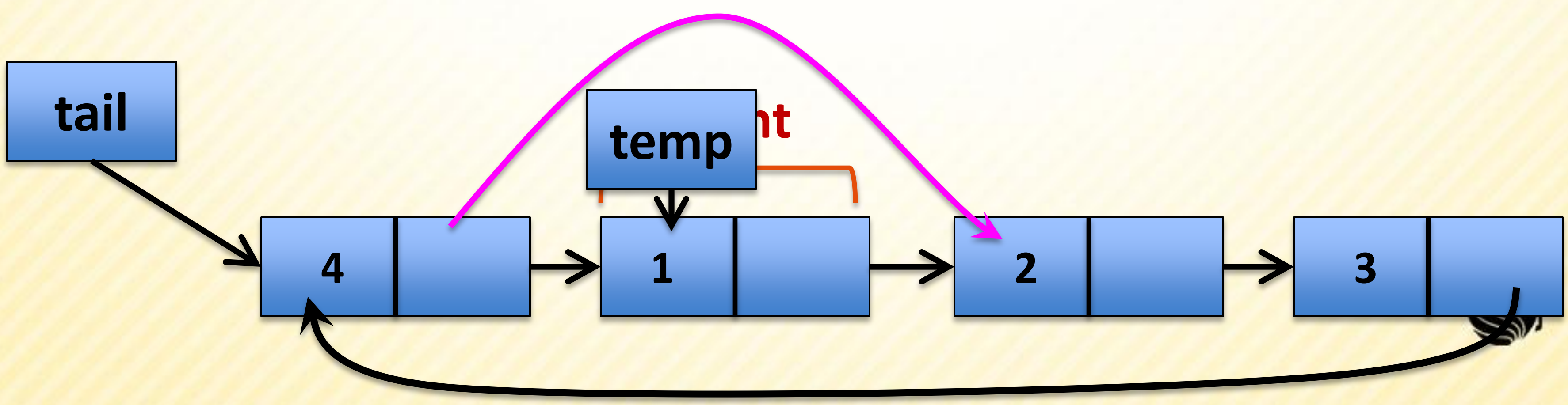
```
typedef struct node {
   int data;
   node *next;
} node;
```

```
struct node* AddEnd(struct node* tail, int val) {
    // Create the new node
    node *temp = (node*)malloc(sizeof(node));
    temp->data = val;
    // Set the new node's next to itself (circular!)
    temp->next = temp;
    // If the list is empty, return new node
    if (tail == NULL) return temp;

    // Set our new node's next to the front
    temp->next = tail->next;
    // Set tail's next to our new node
    tail->next = temp;

    // Return the new end of the list
    return temp;
}
```

The only line of code that's different

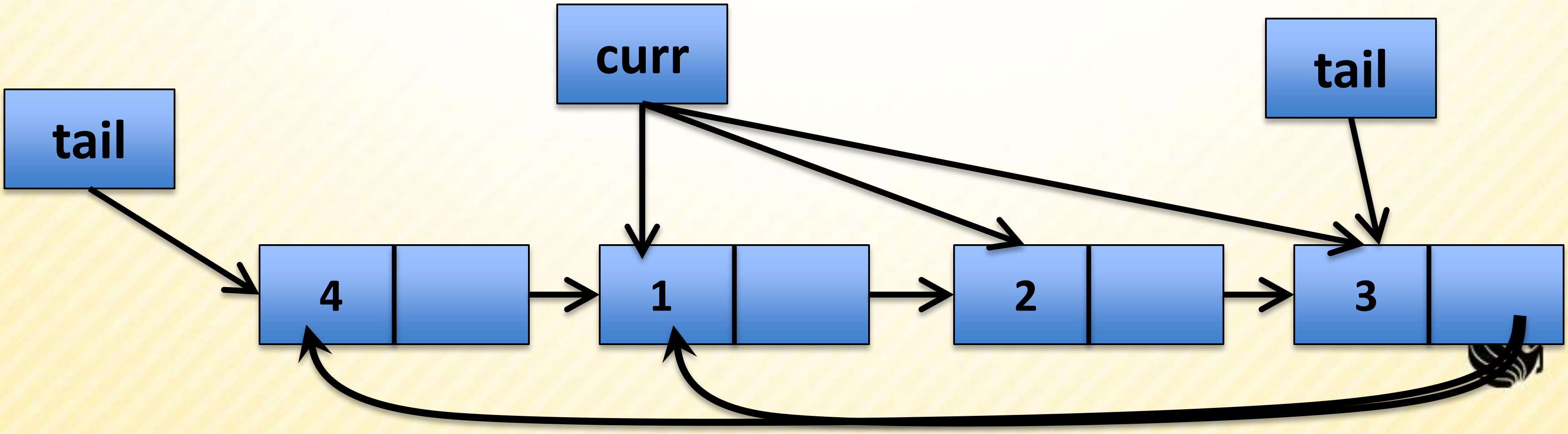
Circularly Linked List

- Deleting the First Node in a Circular Linked List
  - The first node can be deleted by simply replacing the next field of tail node with the next filed of the first node:
    - temp = tail->next; // This is the front
    - tail->next = temp->next; // This is the node after front
    - free(temp);



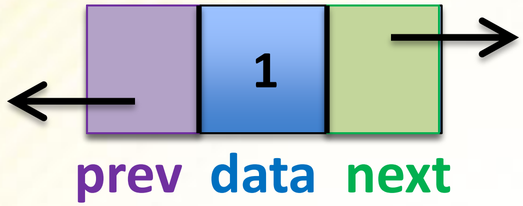
Circularly Linked List

- Deleting the Last Node in a Circular Linked List
  - This is a little more complicated
    - ➢ The list has to be traversed to reach the second to last node.
    - ➢ This had to become the tail node, and its next field has to point to the first node.



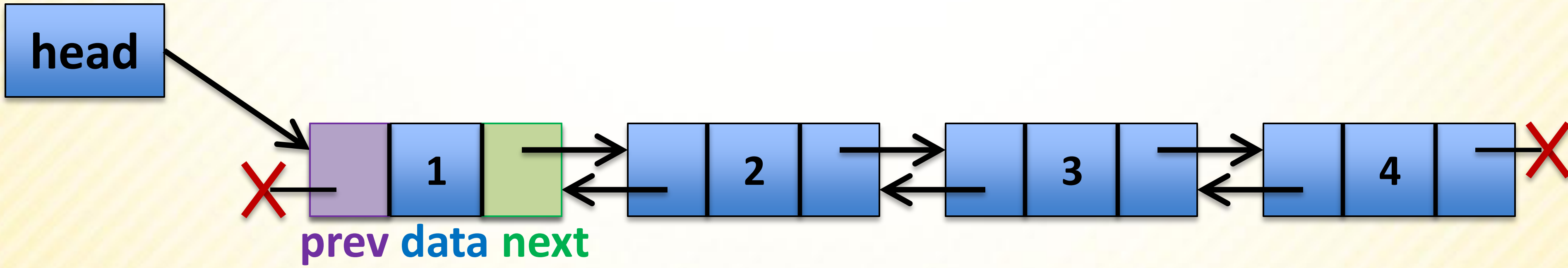
Doubly Linked List

- Doubly Linked List Node:




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

- DoublyLinkedList:

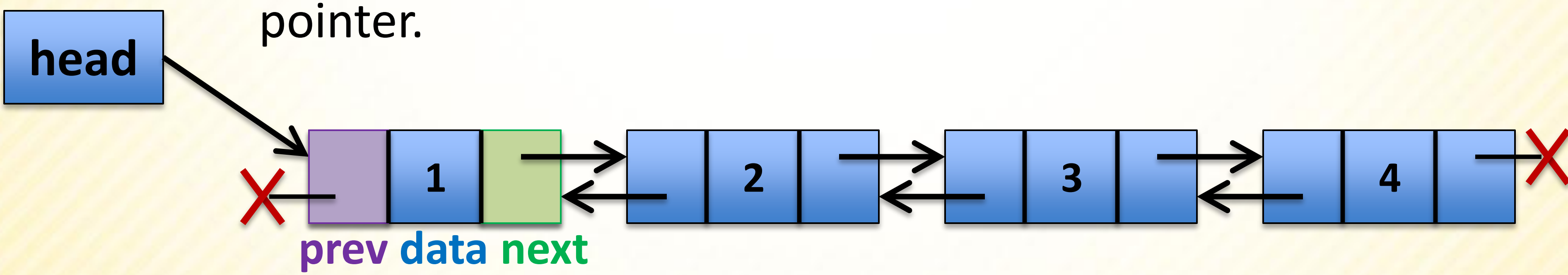


- Each node in the list contains a reference to both:
  - the node which immediately precedes it AND
  - to the node which follows it in the list.

Doubly Linked List

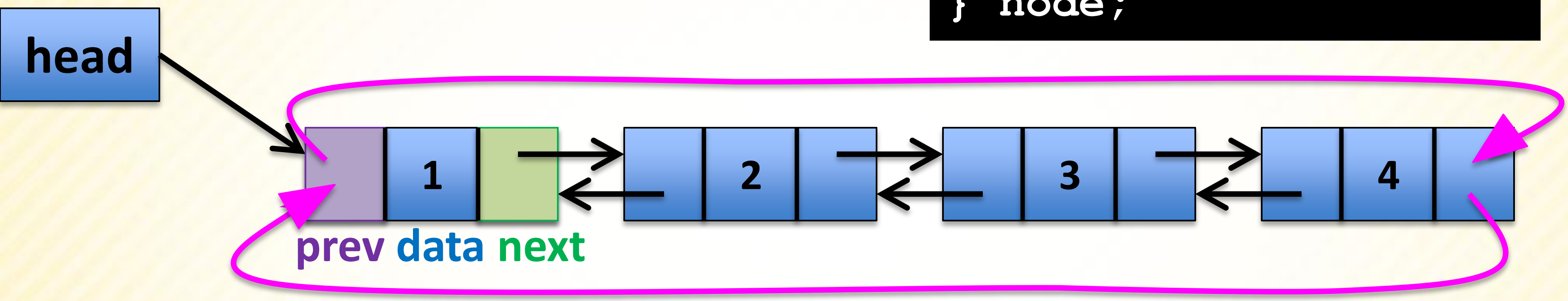
```
typedef struct node {
    int data;
    node *next;
    node *prev;
} node;
```

- DoublyLinkedList:
  - Advantages:
    - Allows searching in BOTH directions
    - Insertion and Deletion can be easily done with a single pointer.



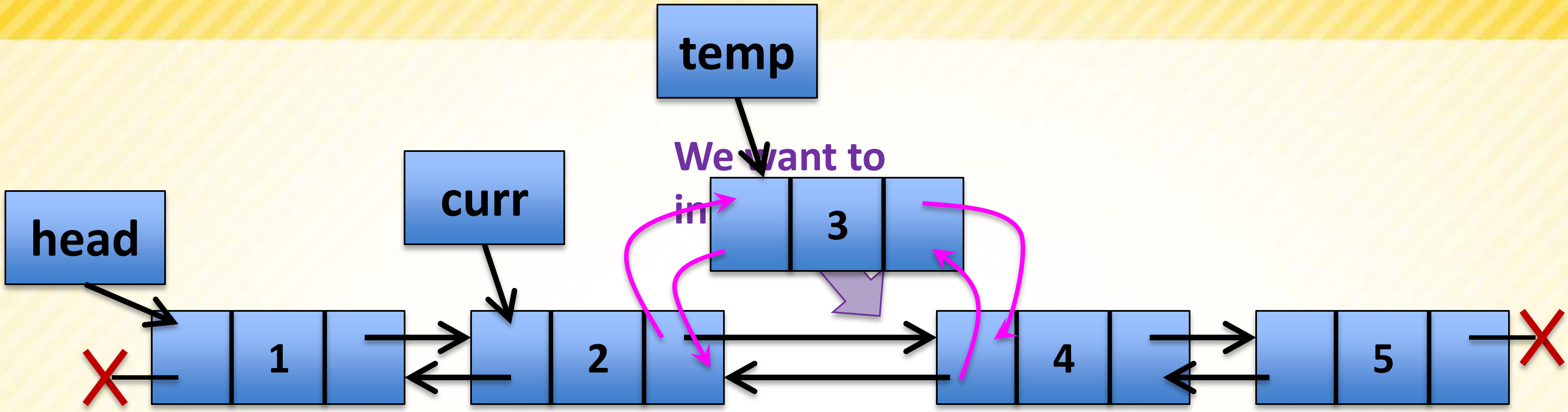
Doubly Linked List

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



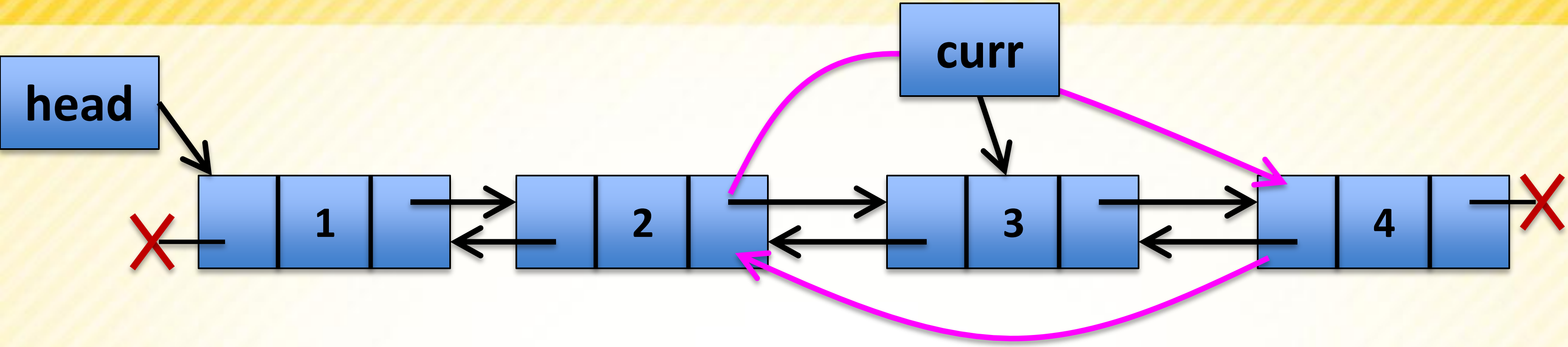
- Circular Doubly Linked List
  - Same as a circular doubly linked list
  - BUT the nodes in the list are doubly linked, so the last node connects to the front AND the first node connects to the last.

Doubly Linked List - Insertion



- Code:
  - temp->prev = curr;
  - temp->next = curr->next;
  - curr->next->prev = temp;
  - curr->next = temp;
- Disadvantage of Doubly Linked Lists:
  - extra space for extra link fields
  - maintaining the extra link during insertion and deletion

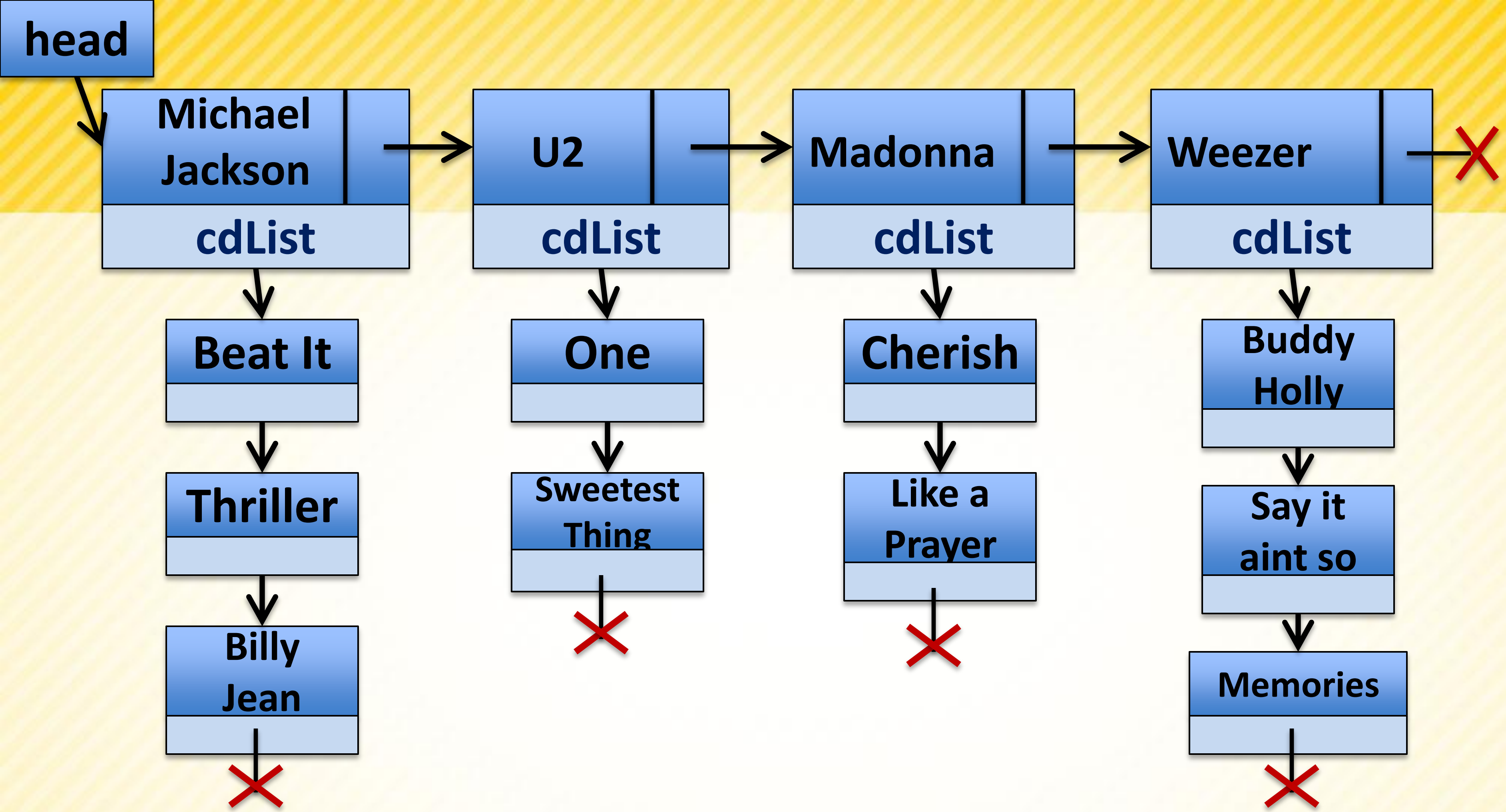
Doubly Linked List - Deletion



- Code:
  - curr->prev->next = curr->next;
  - curr->next->prev = curr->prev;
  - free(curr);
  - (Assuming curr->prev and curr->next are NOT NULL)

A Linked List of Linked Lists

- Linked Lists can be part of more complicated data structures.
  - Consider the situation where we have a linked list of musical artists
  - It might also make sense that each artist has a linked list of songs (or albums) stored




```
struct CDNode {
    char title[50];
    struct CDNode *next;
} ;
```

```
struct ArtistNode {
    char first[30];
    char last[30];
    struct CDNode *cdList;
    struct ArtistNode *next;
} ;
```

Practice Problem

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

- Write a function which accepts a linear linked list J and converts it to a circular linked list.
  - Note this means: J is a pointer to the front of the list.

```
node *convert (node *J) {









}
```