# COMP1511 WEEK 9

Starting 5 minutes past the hour

# Assignment #1 Style Marking Notes

- **Header comments** - don't forget to fill this out with the relevant details (short description of the program)
- **Implement functions** - move some of your logic into helper functions
  - **Overdeep nesting**
  - **Overly complex if statements**
- **Comments above functions**
- Note on **reusing variable names** between functions - YOU CAN!

# Assignment #2 Check In

No longer in draft mode!

- How are you going?
- Do you have any questions?
- Any tips you want to share?

struct node {
int data
}

# Memory Allocation Revision

man malloc ()

void * = pointer

function

struct node * = malloc (sizeof (struct node))

free ( node )

HEAP

- What does malloc do?
  - What are its inputs and output and what do they mean?
  - Describe a function that will allocate memory for a struct and assign a pointer to the result.
- What does free do?
  - What is the input to free and how does it help it do what it needs to do?
- What is a *use after free* error?
  - Give an example.
  - Discuss why these are extremely dangerous, one of the worst causes of bugs in C programs and a major source of security vulnerabilities.

free(node)

node->next

- What is a memory leak?
  - What does dcc --leak-check do?

dcc -lc -o program program.c

# Help me debug?????

```
struct node *new_node(int data) {
    struct node *new = malloc(sizeof(struct node *));
    new->data = data;

    return new;
}
```

# More Linked Lists

When tackling a linked list exercise, it's a good idea to consider the following questions:

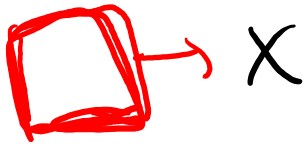
- What cases do I need to consider? Some of the common cases to consider are:
  - Number of nodes (ie empty list, list with one node, list with many nodes)
  - Location in the list (ie, at the start/middle/end of the list)
- Do I need to iterate through a linked list?
  - What loop condition(s) should I use?
  - How many iterators do I need?
- Do I need to malloc/free memory?

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

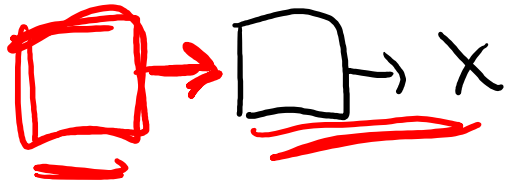
→ X

append to head of the list

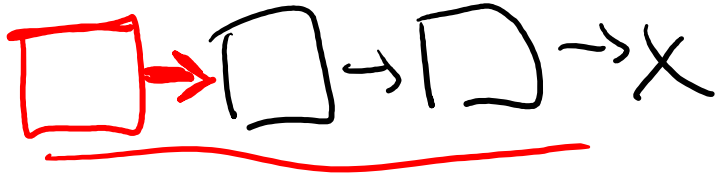
1) nothing in linked list



2) working with the head of the list



3) many nodes



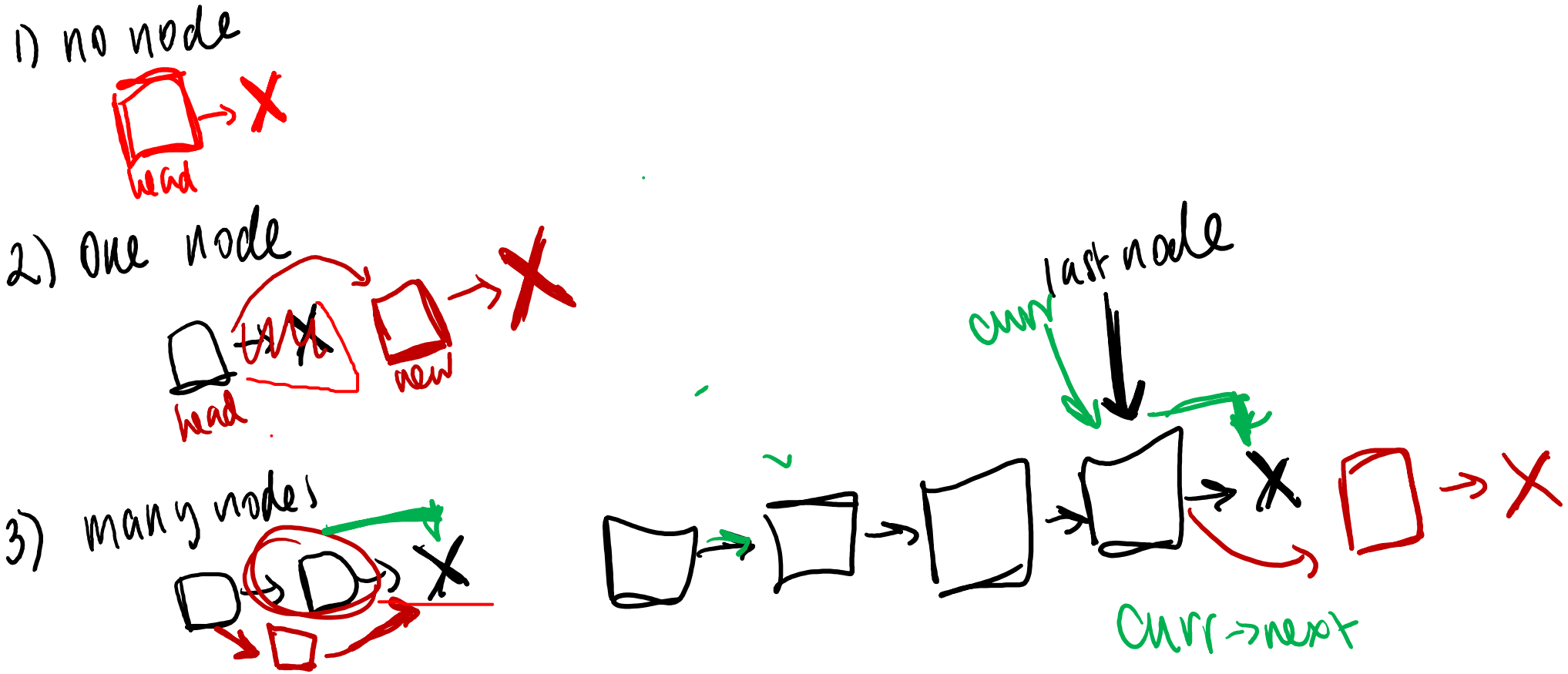
12. Implement a function `add_last` which adds a new node to the end of a given list.

`add_last` should have this prototype:

```
struct node *add_last(struct node *head, int data);
```

`add_last` should call `malloc` to allocate memory for the new node it adds.

`add_last` should return a pointer to the head of the list.



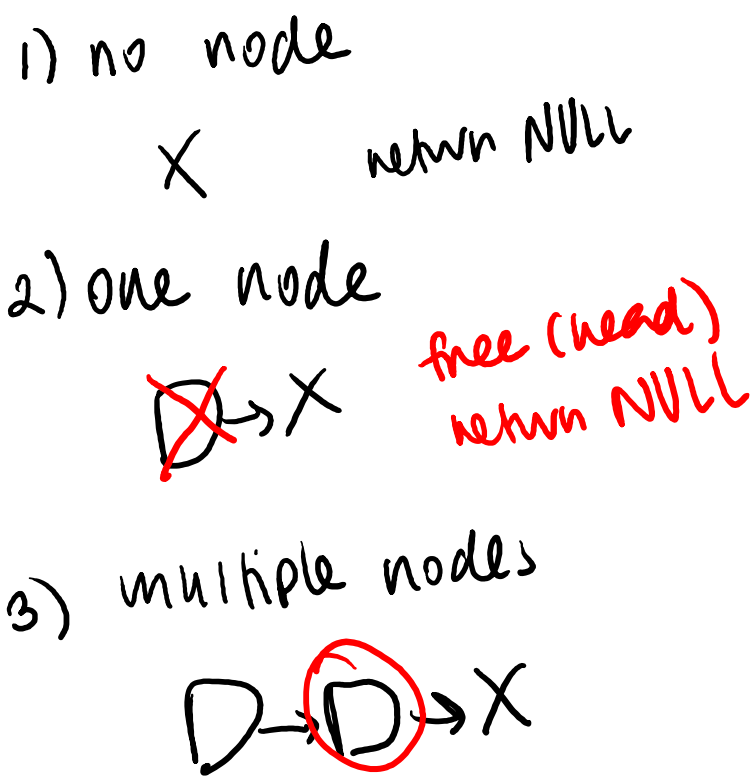
13. Implement a function `delete_last` which deletes the last node from a given list.

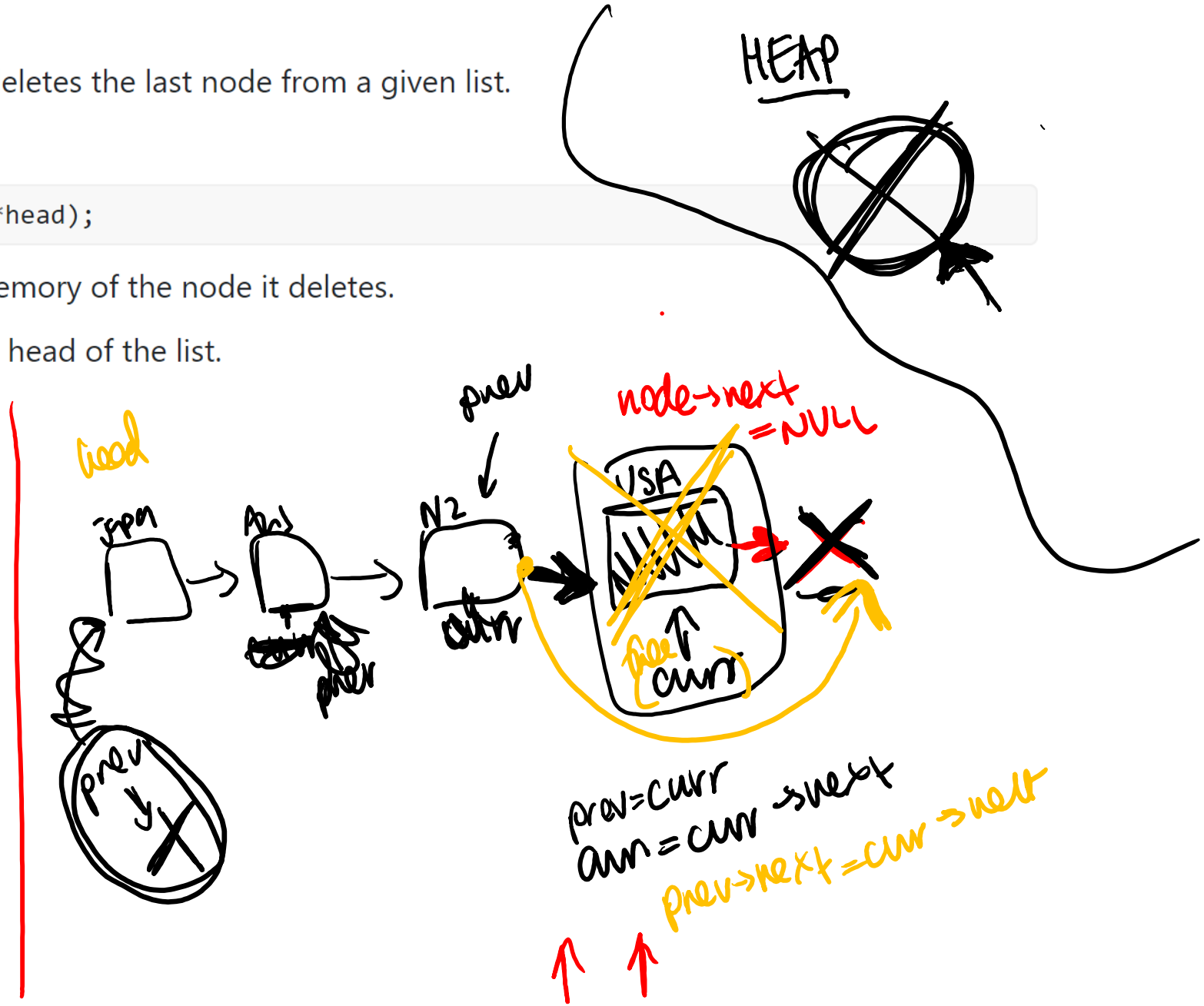
`delete_last` should have this prototype:

```
struct node *delete_last(struct node *head);
```

`delete_last` should call `free` to free the memory of the node it deletes.

`delete_last` should return a pointer to the head of the list.



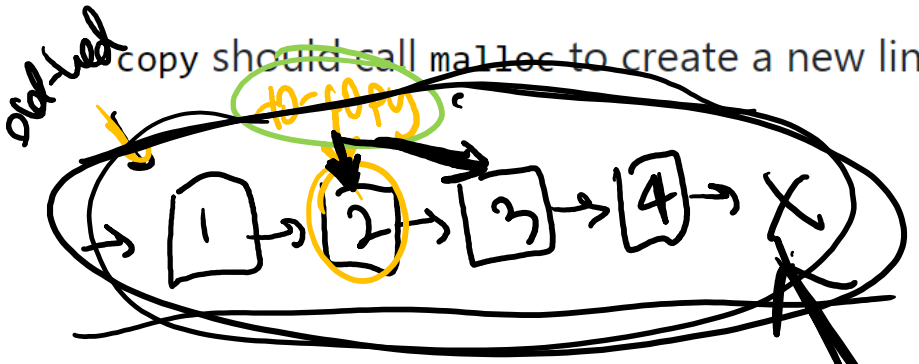



14. Implement a function copy which returns a copy of a linked list.

copy should have this prototype.

```
struct node *copy(struct node *old_head);
```

copy should call malloc to create a new linked list of the same length and which contains the same data.




1) no node in old list

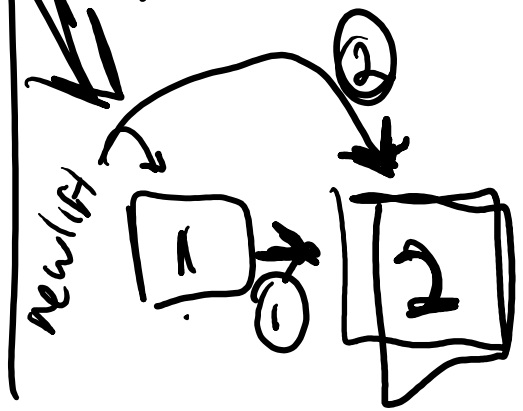
2) one node in old list

3) multiple nodes

X

```
if (old_head == NULL){
    return NULL
}
```

```
struct node *new_head = create_node(old_head->data)
struct node *new_last = new_head
struct node *to_copy = old_head->next
```



15. Implement a function `list_append` which creates a new list by appending the second list to the first.

    `list_append` should have this prototype:

    ```
    struct node *list_append(struct node *first_list, struct node *second_list);
    ```

    Why do we need to make sure it is a new list? Why can't we just change the first list's final node's next pointer to the second list's head?

16. Implement a function `identical` that returns 1 if the contents of the two linked lists are identical (same length, same values in data fields) and otherwise returns 0.

    `identical` should have this prototype:

    ```
    int identical(struct node *first_list, struct node *second_list);
    ```

    `identical` should not create (`malloc`) any new list elements.

17. Implement a function `set_intersection` which given two linked lists in strictly increasing order returns a new linked list containing a copy of the elements found in both lists.

    `set_intersection` should have this prototype:

    ```
    struct node *set_intersection(struct node *set1, struct node *set2);
    ```

    The new linked list should also be in strictly increasing order. It should include only elements found in both lists.

    `set_intersection` should call `malloc` to create the nodes of the new linked list.