DPST1091 / CPTG1391
Introduction to Programming
Week 11 Lecture 1

Lecturer and Course Convener:

Dr Pantea Aria

UNSW
College

Linked Lists Large Application

unswcollege.edu.au

Agenda

- Last lecture
  - Practice Exam
  - Linked Lists Functions –Delete

- Today
- Large Application Example

MyExperience Survey

Tell us what **worked** (and what **didn't**!) — I genuinely value your feedback

UNSW College

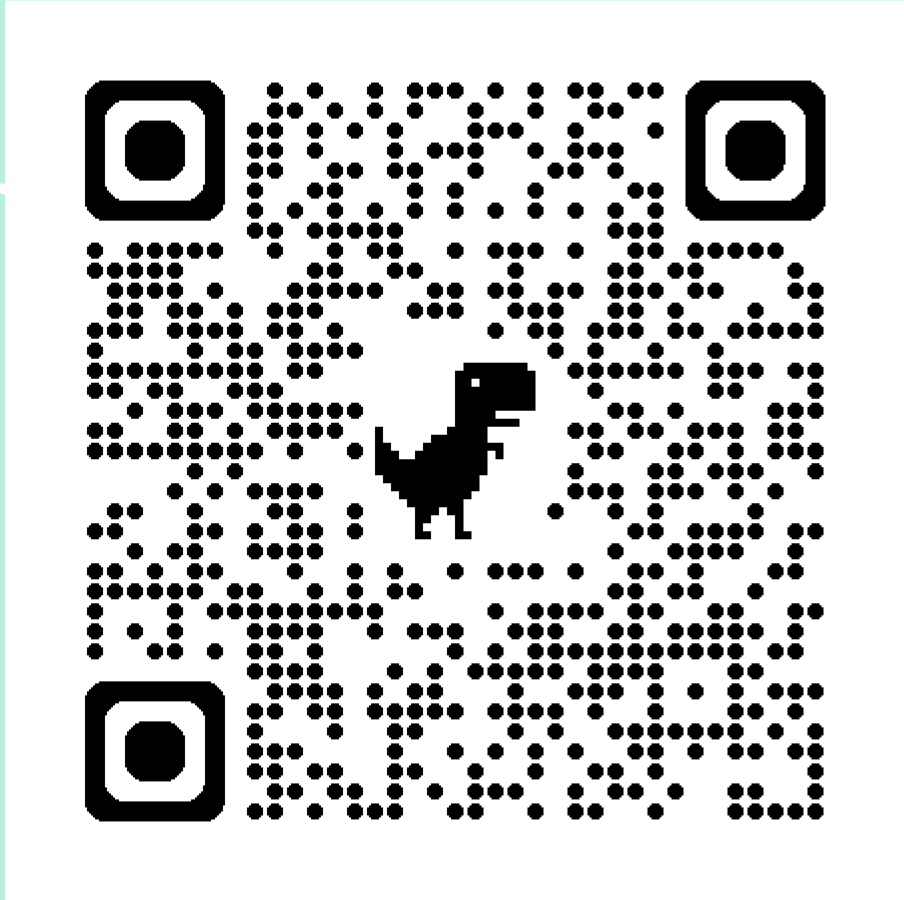
Recap Exercise

How can we print the data stored in the first node?
How can we print the data stored in the second node?
How can we change the next pointer of the first node to NULL?
What effect would that have on the list?

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

Demo

list_exercise.c



Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College

Deletion recap

UNSW College

Deleting the First Node in a Linked List

We must first check whether the list is **empty**

If the list is empty, there is no node to remove.

In that case, we simply return the head pointer, which will be NULL.

```
if (head == NULL) {

    return head;
    //or return NULL;

}
```

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 that we want to delete.

Deleting the First Node in a Linked List

What **issue** would arise if we called free on the head node **before updating the head pointer**?

```
free(head);
```

Deleting the First Node in a Linked List

Once memory **has been freed**, we **can no longer access** it. As a result, we **lose access** to the rest of the list.

```
// This will crash
head = head->next;
```

Deleting the First Node in a Linked List

What problem could occur if we update the head pointer before handling the original first node properly?

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

Deleting the First Node in a Linked List

We no longer have a pointer to the first node, so there is **no way to free** its memory.

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

Deleting the First Node in a Linked List

Let's introduce a separate pointer that refers to the first node.

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

Deleting the First Node in a Linked List

Now we can safely update the head pointer.

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

Deleting the First Node in a Linked List

Now we can safely free the memory allocated to the first node.

```
free(temporary);
```

Deleting the First Node in a Linked List

```
struct node *delete_first_node(struct node *head) {

    if (head == NULL) {
        return head;
    }

    struct node *temporary = head;
    head = head->next;
    free(temporary);
    return head;

}
```

Deleting All Nodes

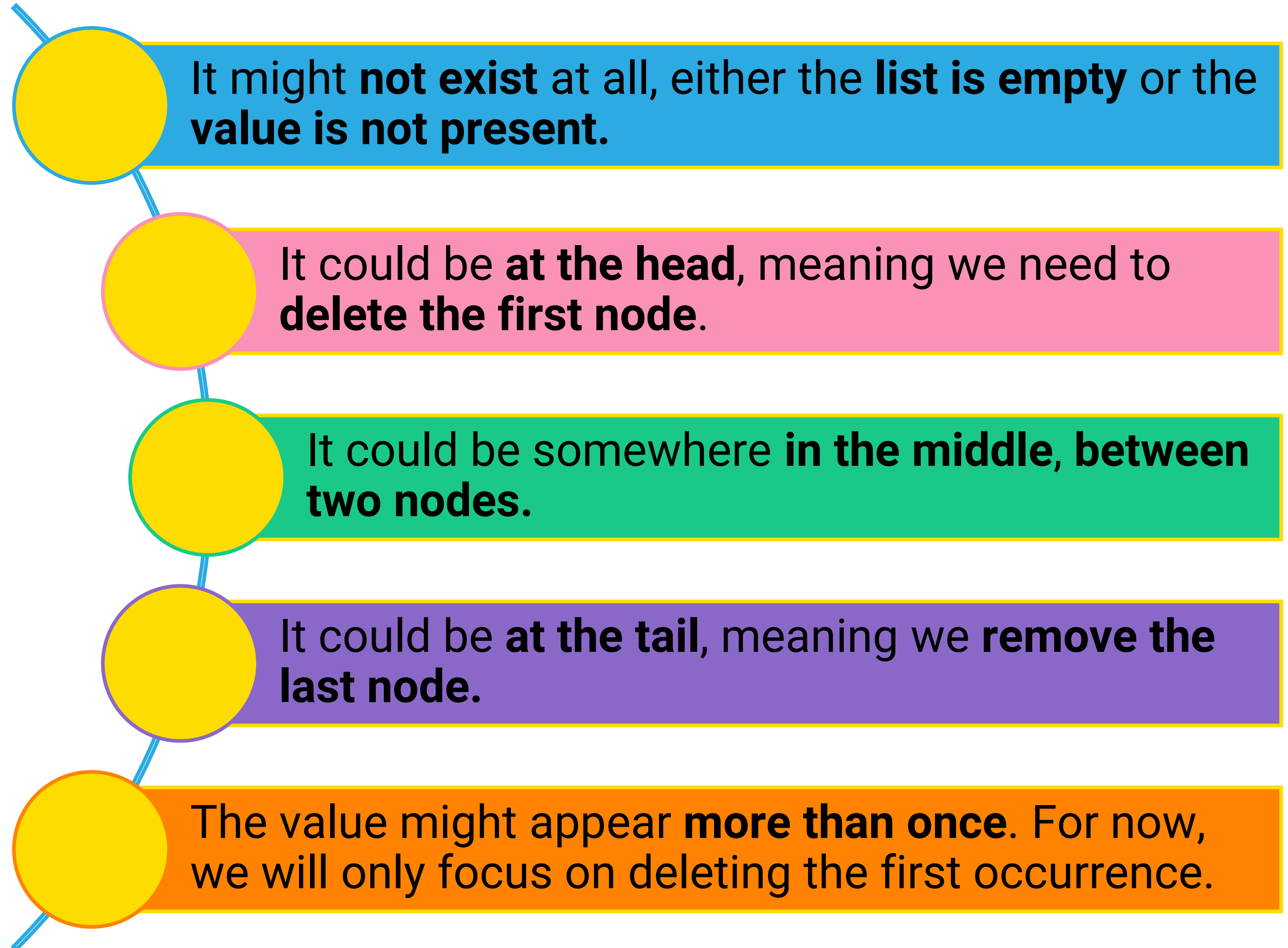
Correct way

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

```
// Delete all nodes from a given list
void delete_all_nodes(struct node *head) {

    struct node *current = head;

    while (current != NULL) {
        head = head->next;
        free(current);
        current = head;
    }

}
```

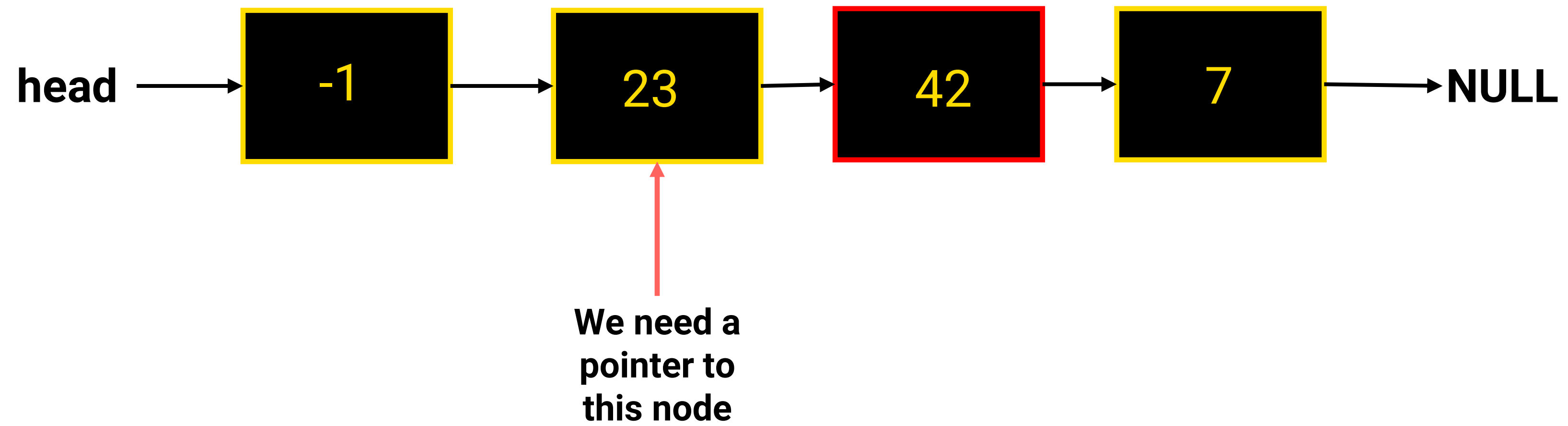
Search and Delete

We want to find a node that contains a specific value and then remove it from the list. Where might this node be located?



Search and delete between 2 nodes

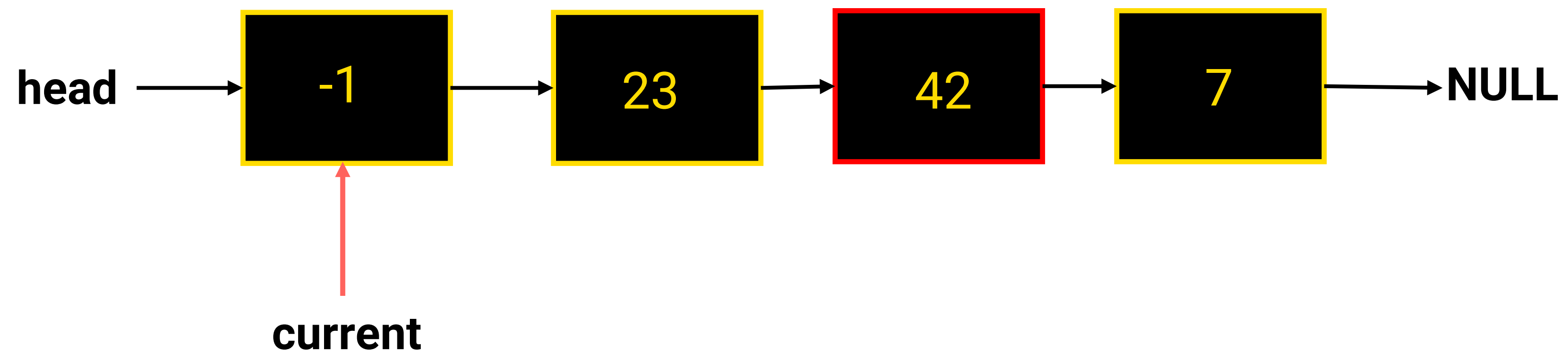
To delete a node, we must **connect the previous node directly to the next node**; so, if we want to remove the node containing 42, we first need to **locate the node that comes before it**.



Search and delete between 2 nodes

Method 1

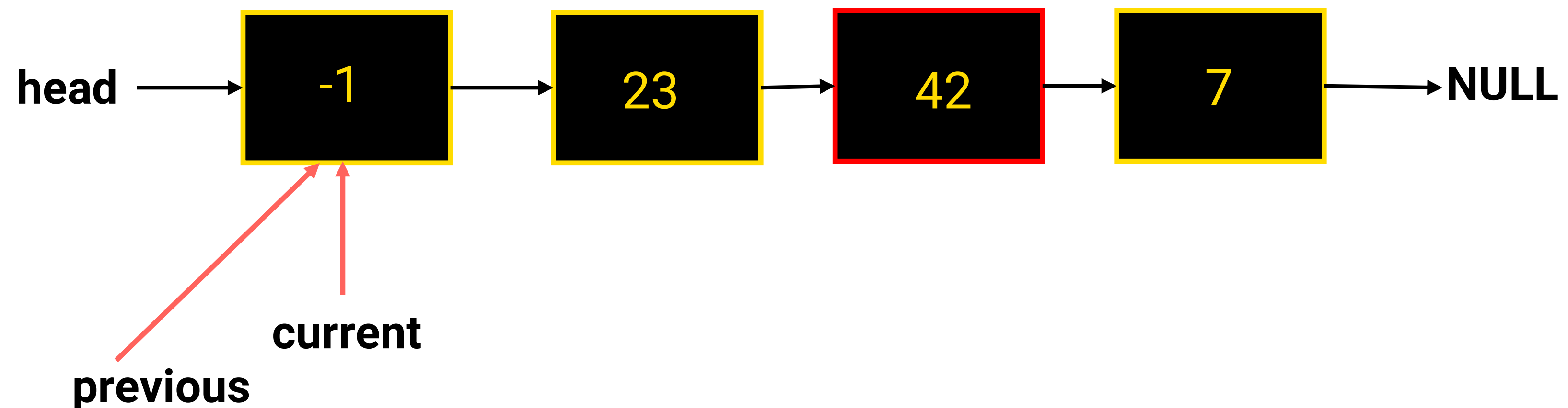
```c
// Method 1:
// Use two pointers while traversing the list:
// - 'current' moves through each node to find the target value.
// - 'previous' keeps track of the node before 'current'.
// This allows us to relink nodes properly when deleting.

struct node *previous = NULL;
struct node *current = head;

// Traverse the list until:
// 1. We reach the end (current == NULL), or
// 2. We find the node containing search_key.

while (current != NULL && current->data != search_key) {

    previous = current;       // Move previous forward
    current = current->next; // Move current forward

}
```



Search and delete between 2 nodes

Method 1

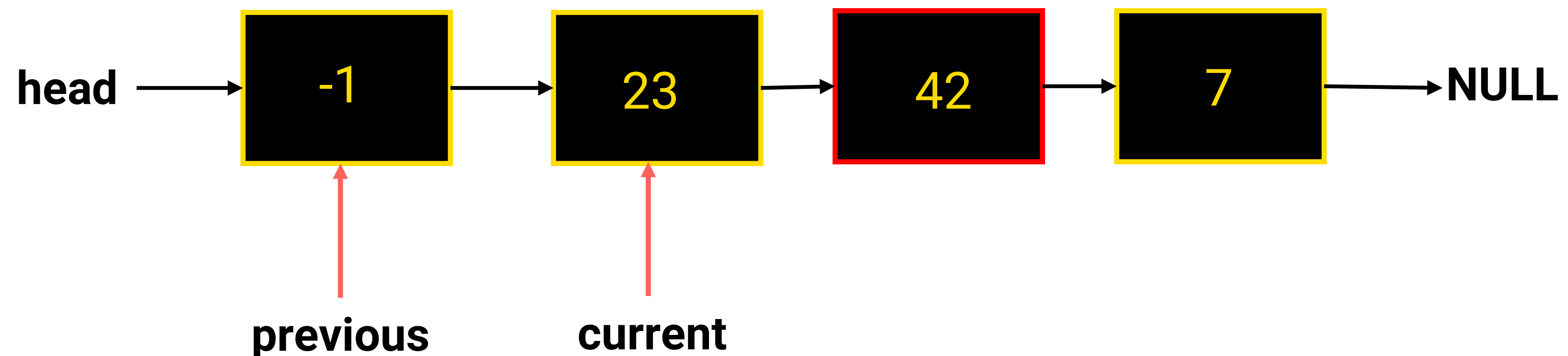
```
// Method 1:
// Use two pointers while traversing the list:
// - 'current' moves through each node to find the target value.
// - 'previous' keeps track of the node before 'current'.
// This allows us to relink nodes properly when deleting.

struct node *previous = NULL;
struct node *current = head;

// Traverse the list until:
// 1. We reach the end (current == NULL), or
// 2. We find the node containing search_key.

while (current != NULL && current->data != search_key) {

    previous = current;       // Move previous forward
    current = current->next; // Move current forward

}
```



Search and delete between 2 nodes

Method 1

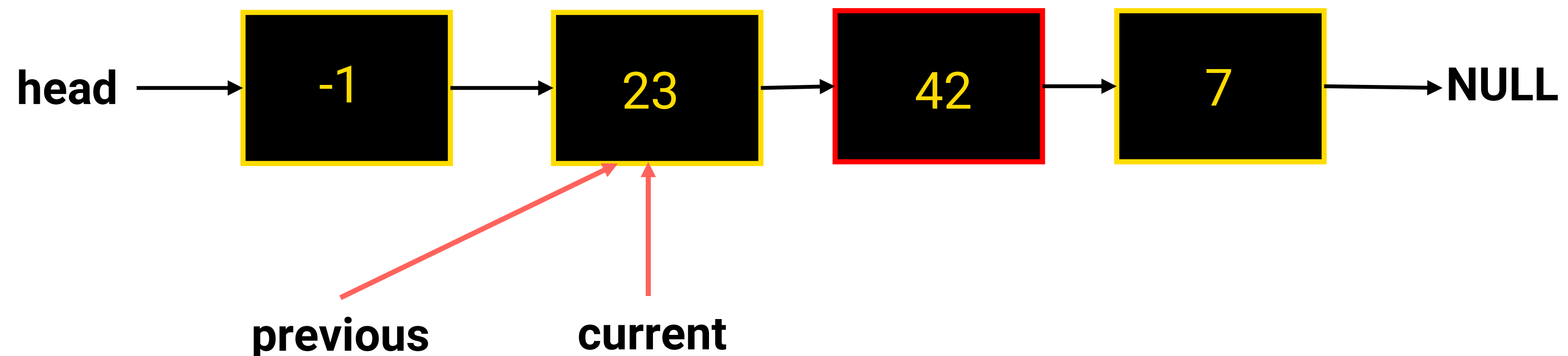
```c
// Method 1:
// Use two pointers while traversing the list:
// - 'current' moves through each node to find the target value.
// - 'previous' keeps track of the node before 'current'.
// This allows us to relink nodes properly when deleting.

struct node *previous = NULL;
struct node *current = head;

// Traverse the list until:
// 1. We reach the end (current == NULL), or
// 2. We find the node containing search_key.

while (current != NULL && current->data != search_key) {

    previous = current;        // Move previous forward
    current = current->next; // Move current forward

}
```



Search and delete between 2 nodes

Method 1

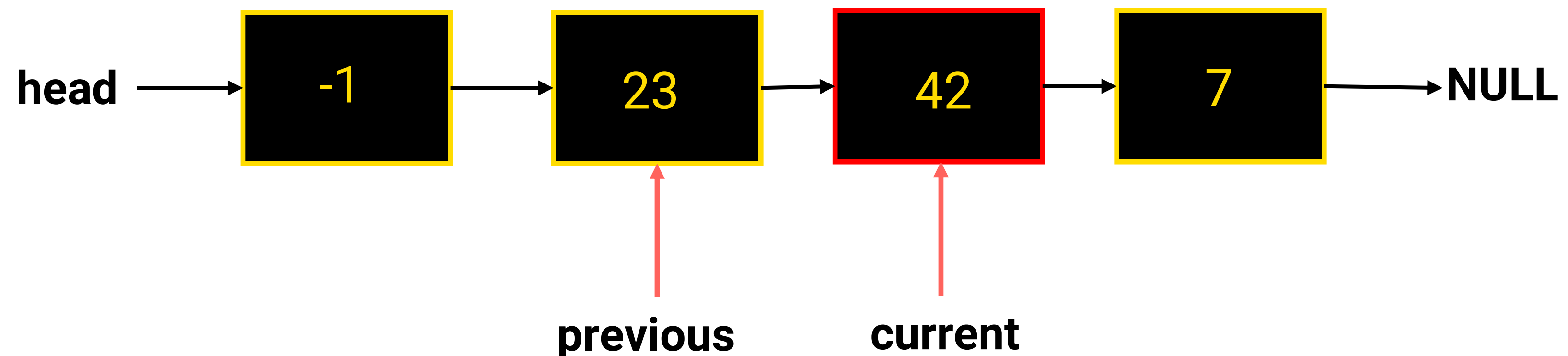
```c
// Method 1:
// Use two pointers while traversing the list:
// - 'current' moves through each node to find the target value.
// - 'previous' keeps track of the node before 'current'.
// This allows us to relink nodes properly when deleting.

struct node *previous = NULL;
struct node *current = head;

// Traverse the list until:
// 1. We reach the end (current == NULL), or
// 2. We find the node containing search_key.

while (current != NULL && current->data != search_key) {

    previous = current;       // Move previous forward
    current = current->next; // Move current forward

}
```



Search and delete between 2 nodes

Method 1

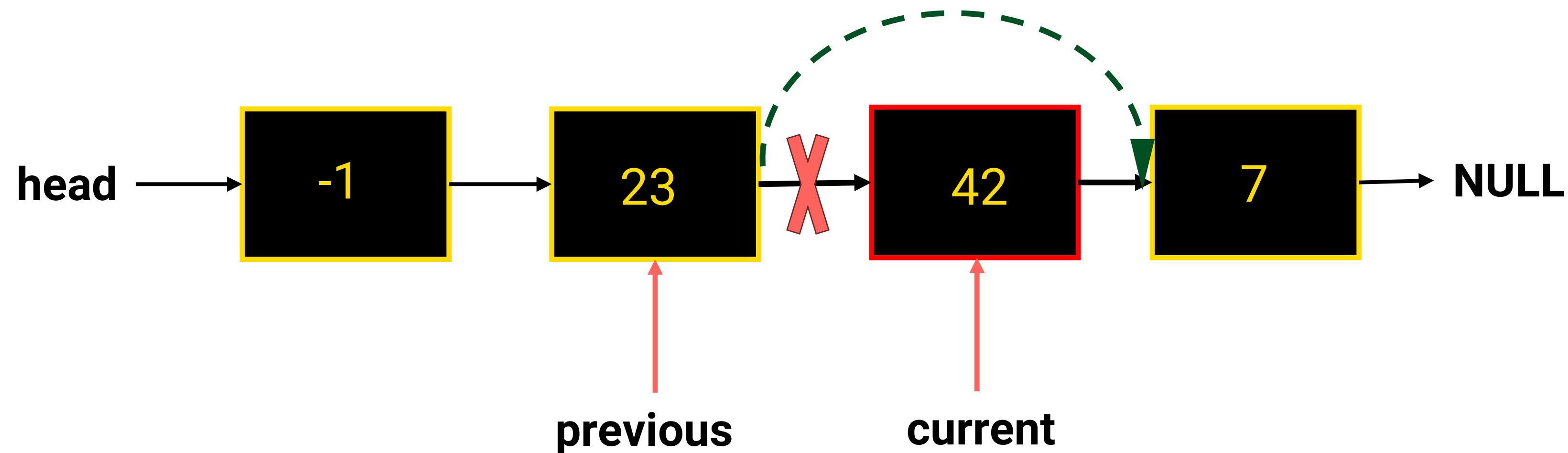
```
// Method 1:
// Use two pointers while traversing the list:
// - 'current' moves through each node to find the target value.
// - 'previous' keeps track of the node before 'current'.
// This allows us to relink nodes properly when deleting.

struct node *previous = NULL;
struct node *current = head;

// Traverse the list until:
// 1. We reach the end (current == NULL), or
// 2. We find the node containing search_key.

while (current != NULL && current->data != search_key) {

    previous = current;       // Move previous forward
    current = current->next; // Move current forward

}
```



Search and delete between 2 nodes

Method 1

Now, we must **update the links** so that the **previous node points to the node that comes after the one being removed (current)**.

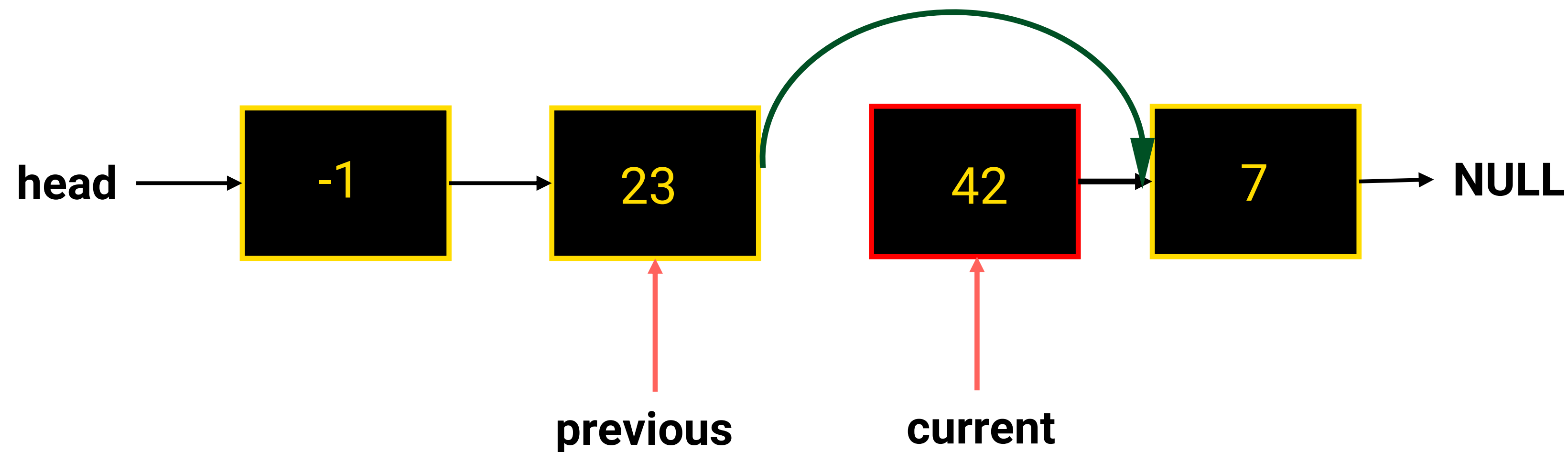


Search and delete between 2 nodes

Method 1

To connect **previous node to the node that comes after the one being removed (current)**.

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

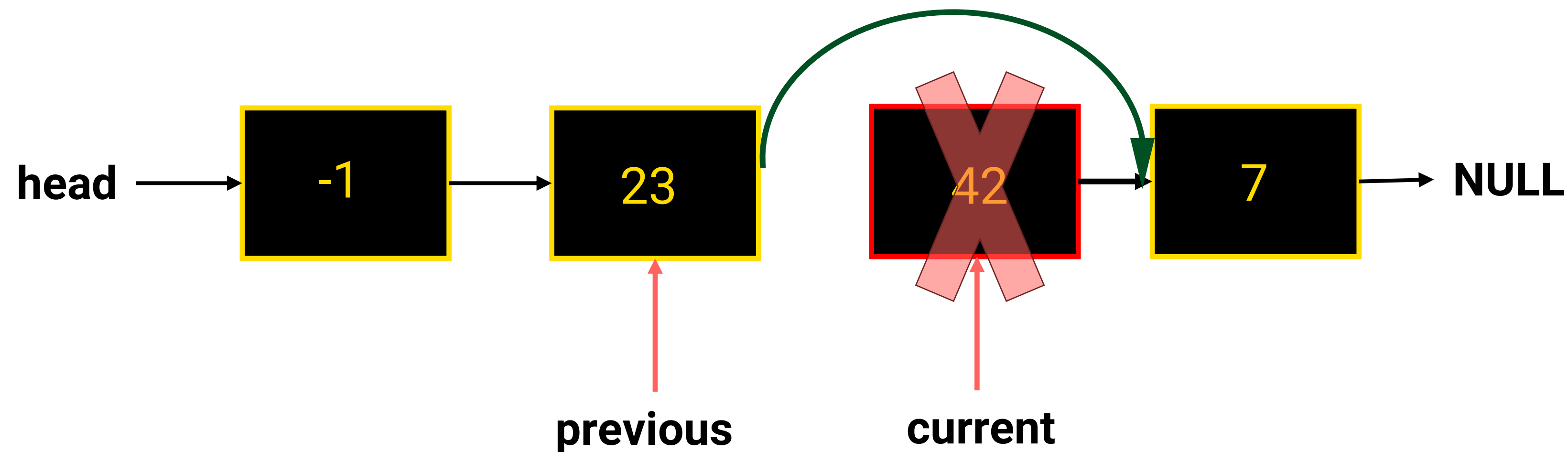


Search and delete between 2 nodes

Method 1

Now we can **free** the node we wanted to **remove (current)**

```
free(current);
```

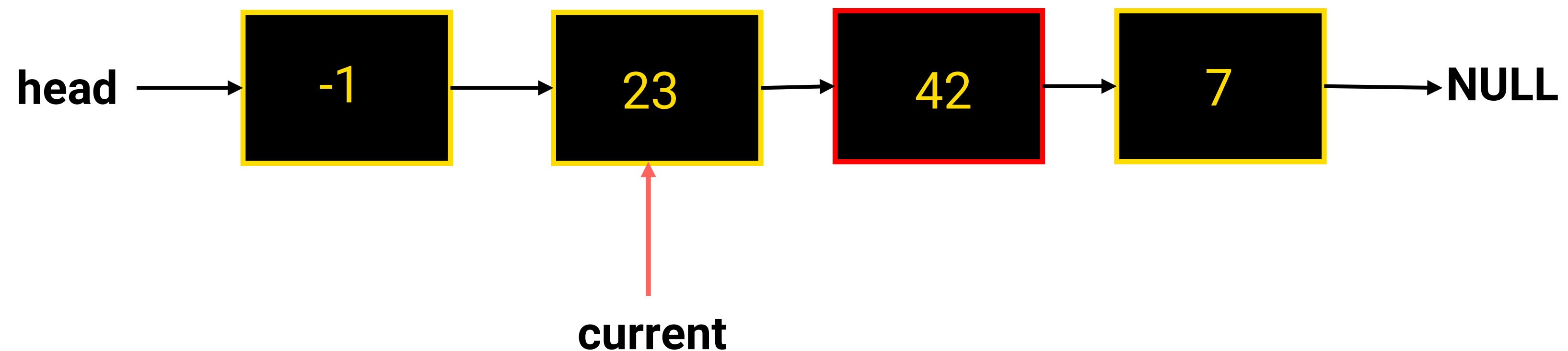


Search and delete between 2 nodes

Method 2

Use only one pointer (current) to move through the list.
Instead of looking at the current node's data,
we examine the NEXT node's data (current->next).

Why? Because if current->next contains the value
we want to delete, then 'current' is the node
right before the one we need to remove.

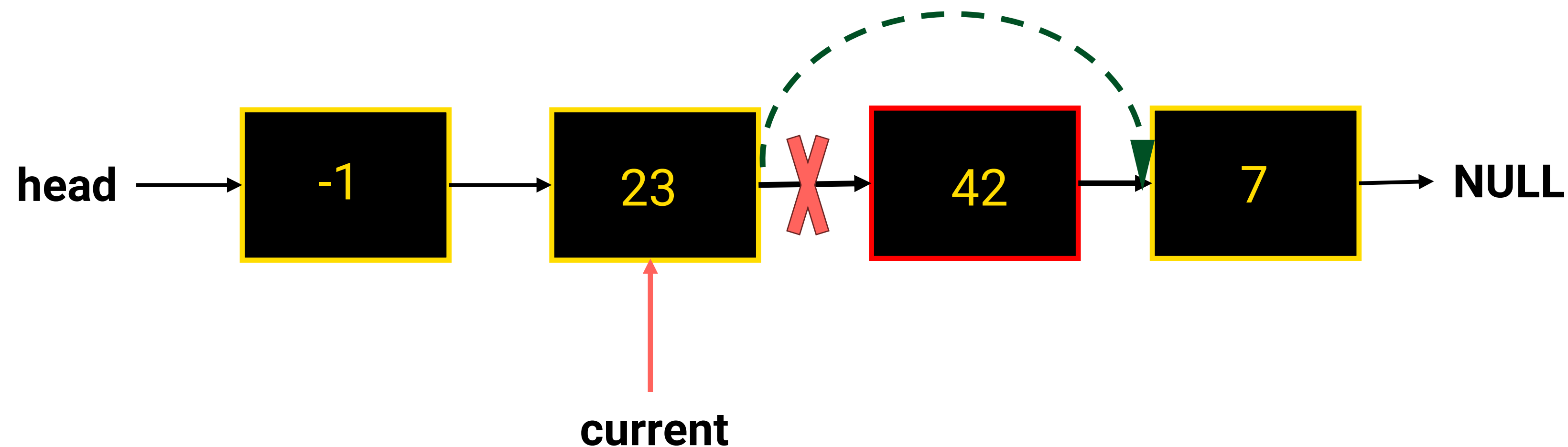


Search and delete between 2 nodes

Method 2

Next, we must update the link **so that the current node points to the node after the one being removed**. However, we still **need a separate pointer** to the node we intend to **free**. **Otherwise, we lose it!!!!!**

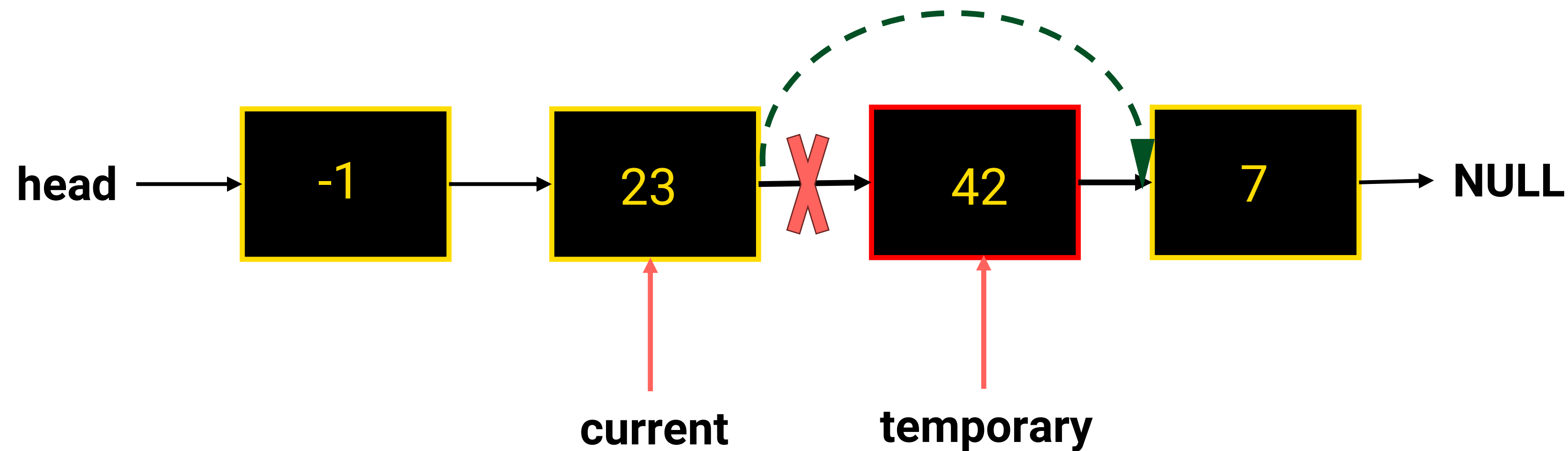
.



Search and delete between 2 nodes

Method 2

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

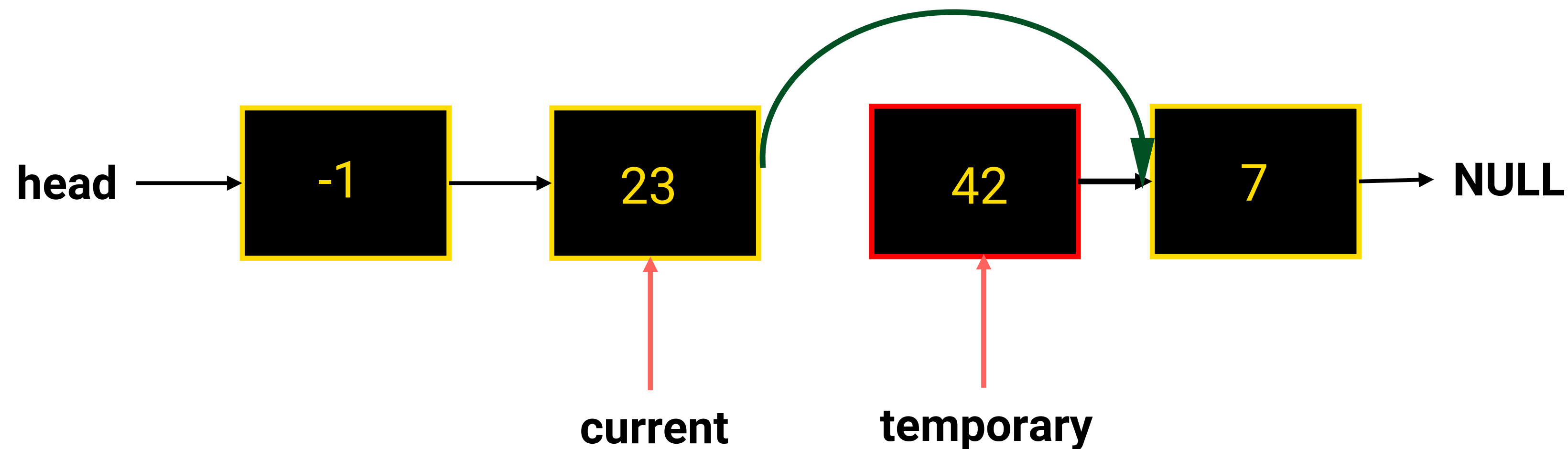


Search and delete between 2 nodes

Method 2

To connect **current node to the node that comes after the one being removed (temporary):**

```
current->next = temporary->next;
```

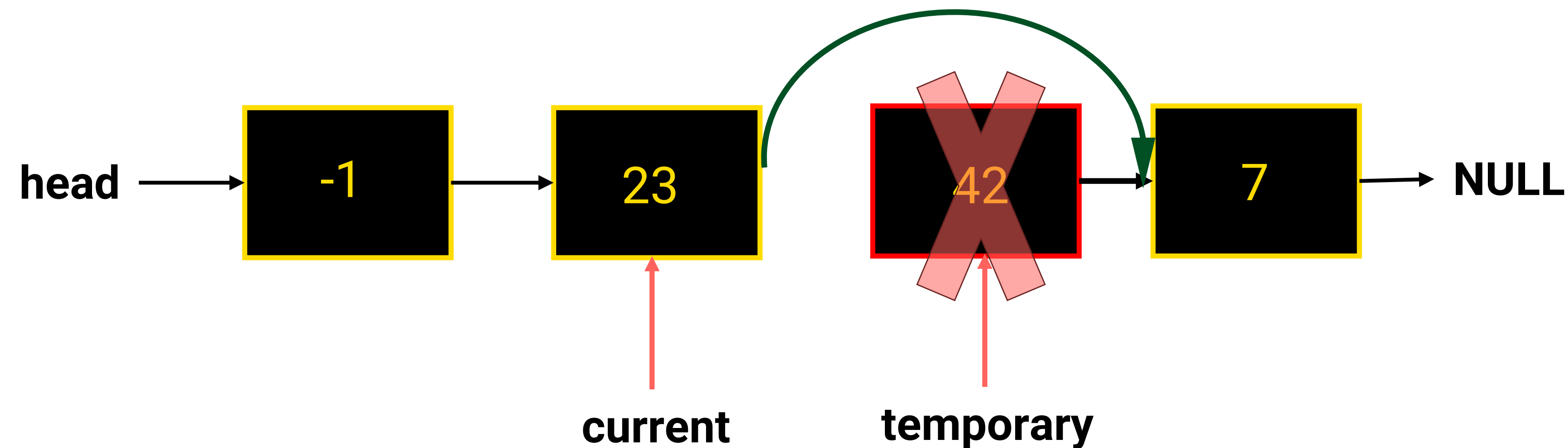


Search and delete between 2 nodes

Method 2

Now we can **free** the node we wanted to **remove** **(current)**

```
free(temporary);
```



Large Application Example with Linked List

UNSW College

Package Delivery Management System

(Linked Lists)

You will implement a package delivery management system in C using linked lists.

Each delivery centre contains a linked list of packages.

- Each package stores information such as:
  - sender
  - receiver
  - weight
  - prioity
  - delivery status

Complete the functions that:

- Create centres
- Insert packages
- Search for packages
- Delete packages
- Merge centres & split centres
- Clear memory

- package_delivery_system.c (TODO – implement functions)
- package_delivery_system.h (provided structs & prototypes)
- main.c (interactive program)
- test_main.c (testing program)

Image is made by AI

Files provided:

File	Description
`package_delivery_system.c`	Implement the TODO functions
`package_delivery_system.h`	Provided structs and prototypes
`main.c`	Interactive program
`test_main.c`	Testing program

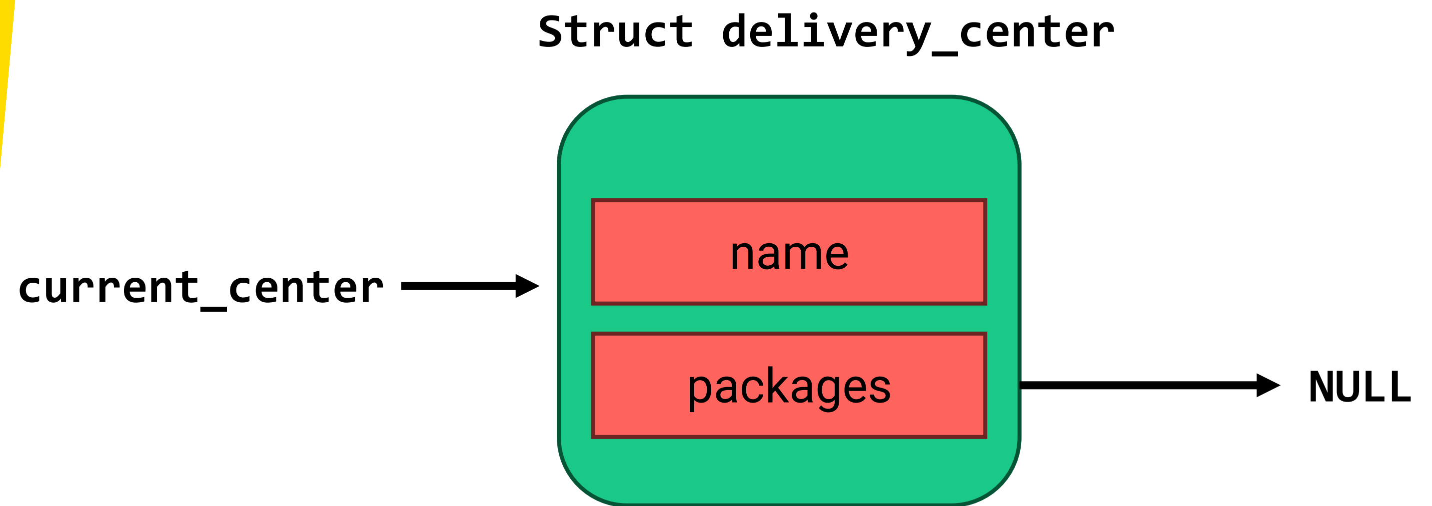
Things to do before you start coding

Step	What to Do
Understand the Problem	Identify what the provided code does, understand how the parts work together, and know how to compile and run the program.
Draw Diagrams	Sketch diagrams to visualise the data structures and program flow before and while implementing each function.
Consider Test Cases	Think about different scenarios to test your functions before and during implementation.

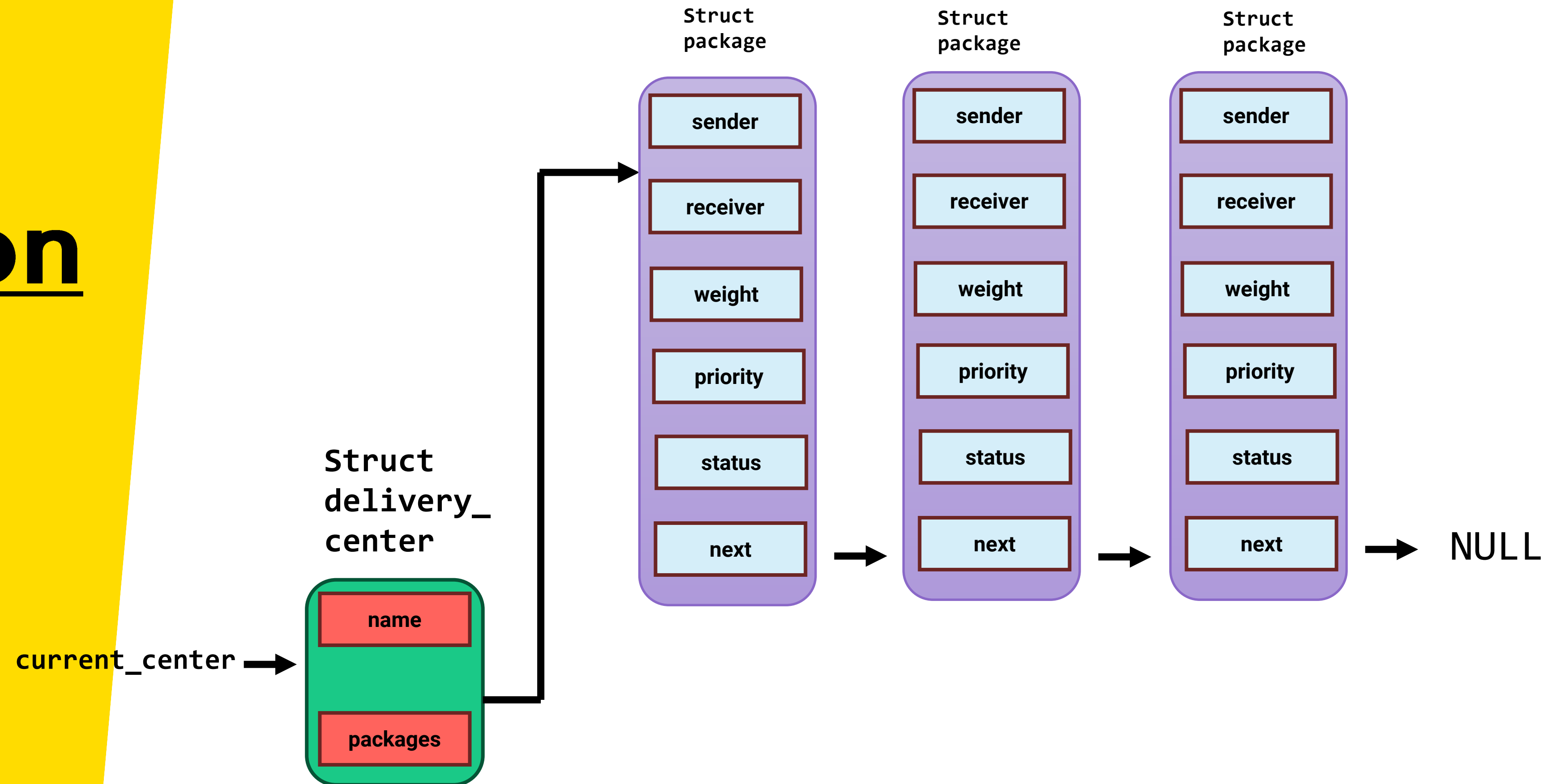
Structs and enums

```c
#define MAX_LEN 100

enum priority_type {
    STANDARD,
    EXPRESS,
    URGENT
};

enum status_type {
    PENDING,
    IN_TRANSIT,
    DELIVERED
};

struct package {
    char sender[MAX_LEN];
    char receiver[MAX_LEN];
    double weight;
    enum priority_type priority;
    enum status_type status;
    struct package *next;
};

struct delivery_centre {
    char name[MAX_LEN];
    struct package *packages;
};
```

Visualisation of System



Visualisation of System



How to Compile and Run this project

Program	Compile Command	Run Command
Testing program	`dcc -o test_main test_main.c package_delivery_system.c`	`./test_main`
Interactive program	`dcc -o main main.c package_delivery_system.c`	`./main`

Functions on each stage

Stages

Stage 1

- create_centre
- insert_package_at_head
- search_package

Stage 2

- clear_centre
- delete_package_of_priority

Stage 3

- merge_centres
- split_centre

Stage 4

Stage 4

Keep track of the **number of packages in a centre** to make count_packages more efficient.

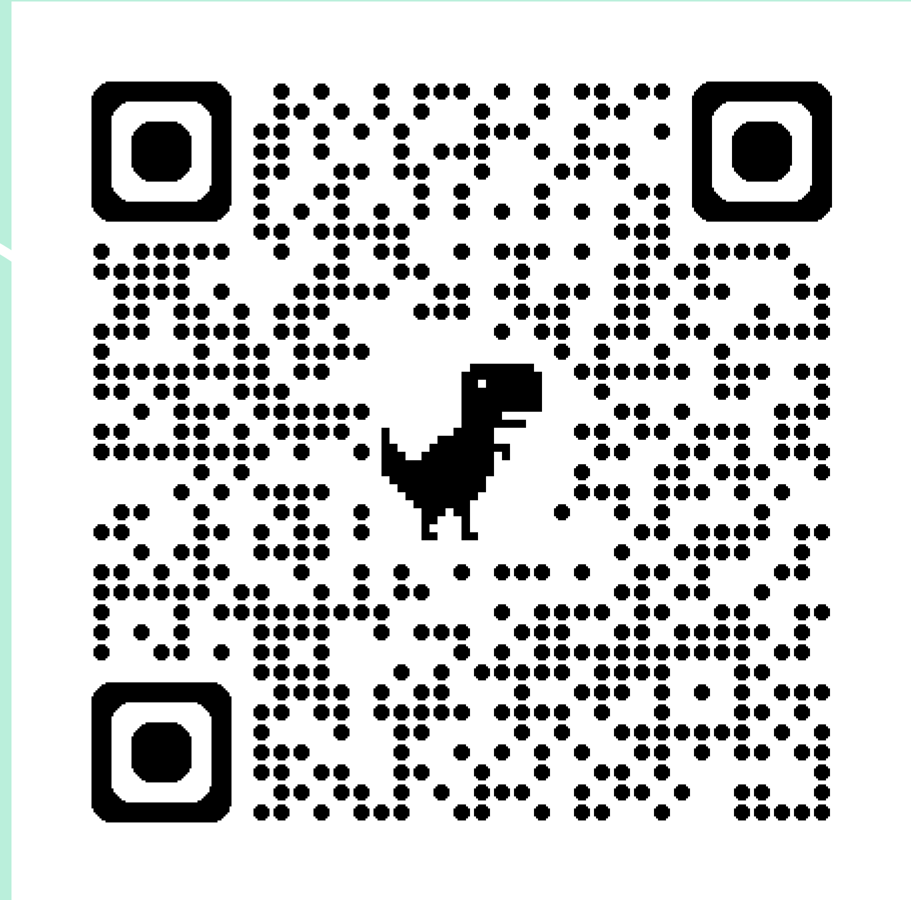
Create an **account system struct** that contains:
- a linked list of **delivery centres**
- an **account name**

Write code to **print all delivery centres**.

Write code to **print all packages in all centres**.

Demo

Package_delivery_system.c



Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me

26T1 Voice of the Student

26T1 Voice of the Student – Fill out form

See you soon …

UNSW College