SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220413<A|D>

Replace <A|D> with this section's letter

Linked Lists II

—

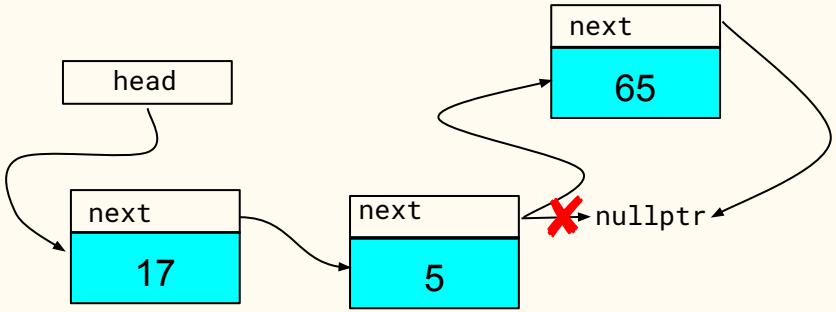
CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- A linked list toolkit (continued)

Building a list

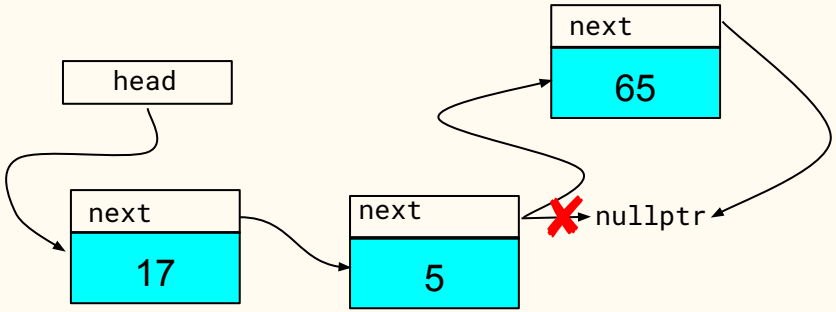
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
    }
}
```



adding a Node to end of list

Building a list

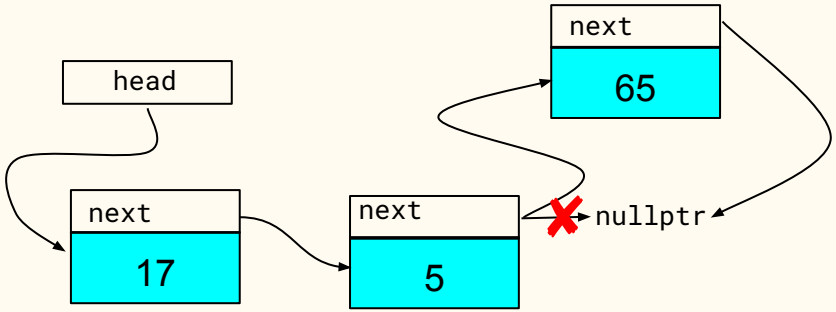
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        ---
    }
}
```



adding a Node to end of list

Building a list

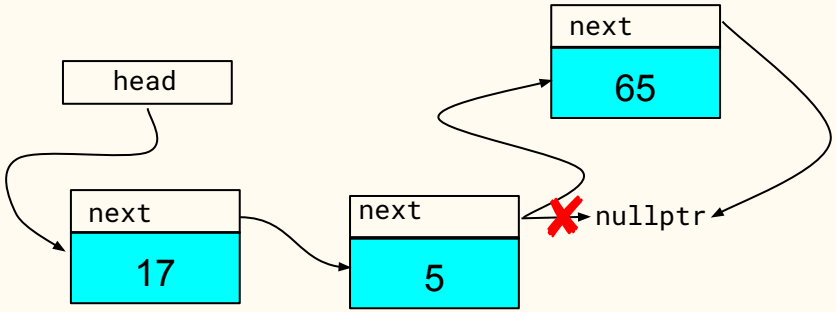
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = ___;
    }
}
```



adding a Node to end of list

Building a list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = _15_;
    }
}
```



adding a Node to end of list

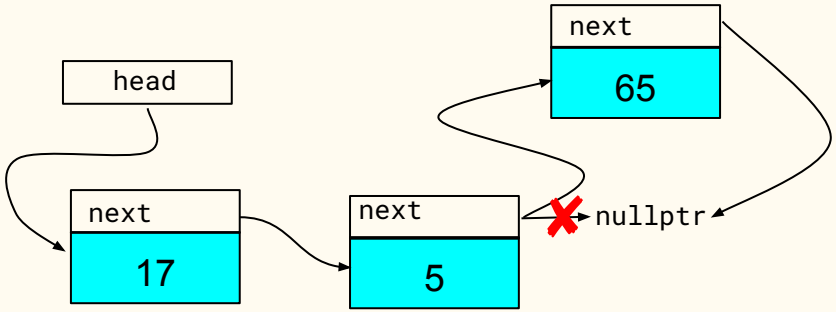
TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220413<A|D>

Replace <A|D> with this section's letter

Which address replaces blank #15 for initializing a pointer used to traverse the list?

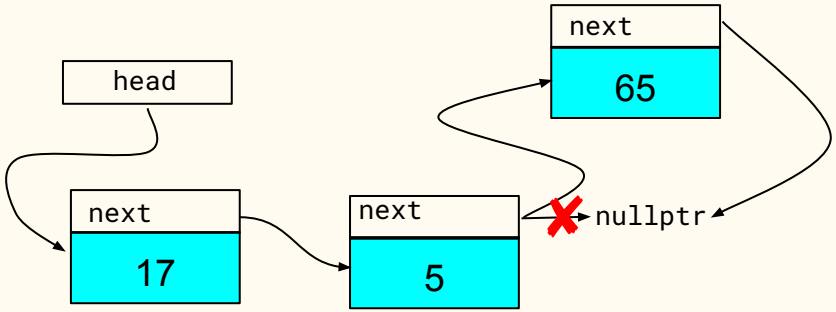
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = _15_;
    }
}
```



adding a Node to end of list

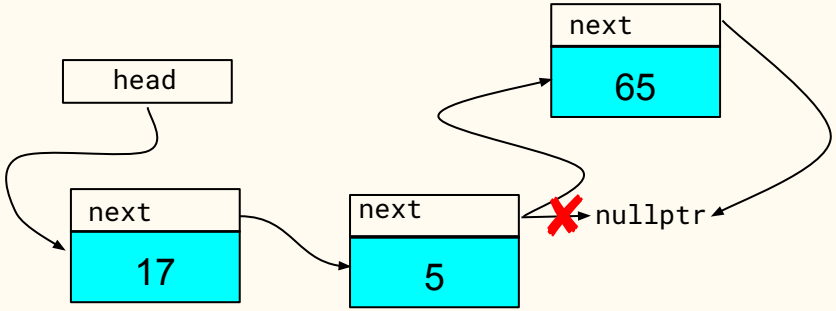
Building a list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
    }
}
```



adding a Node to end of list

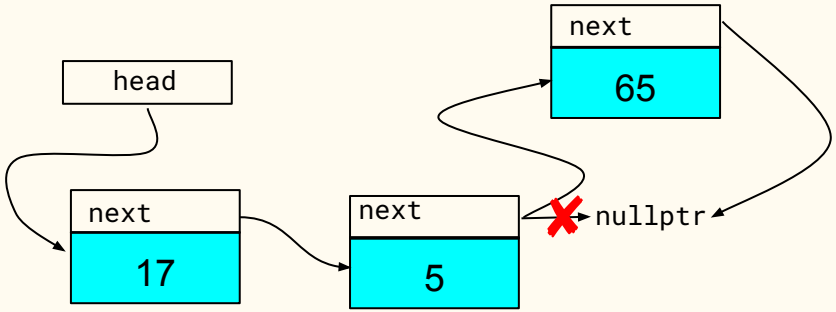
How will we know that we have reached the tail node of our list?



adding a Node to end of list

Building a list

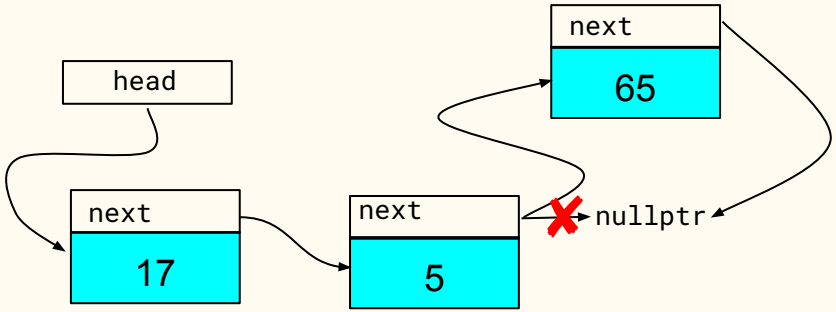
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (___) {

        }
    }
}
```



adding a Node to end of list

Building a list

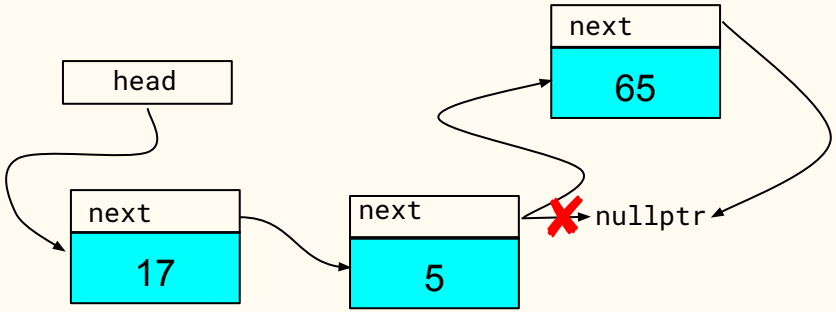
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (_16_) {

        }
    }
}
```



adding a Node to end of list

Which expression replaces blank #16 to terminate the while loop used for traversing the list?

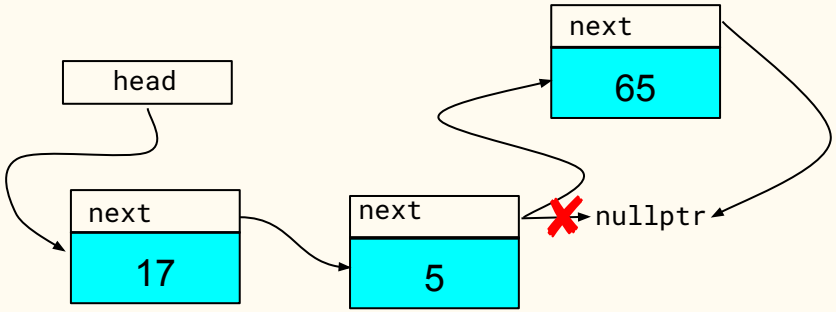
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (_16_) {

        }
    }
}
```



adding a Node to end of list

Building a list

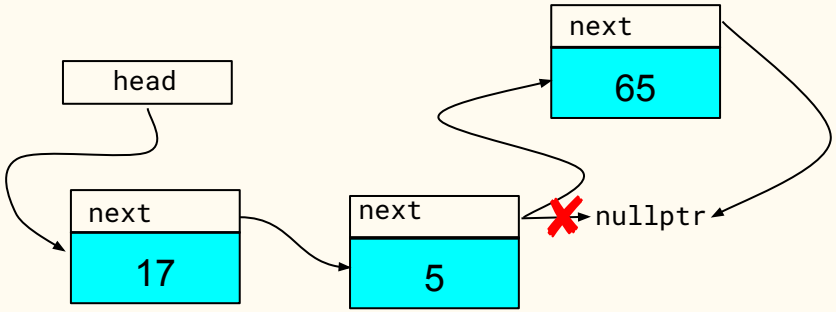
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (curr->next != nullptr) {

        }
    }
}
```



adding a Node to end of list

Building a list

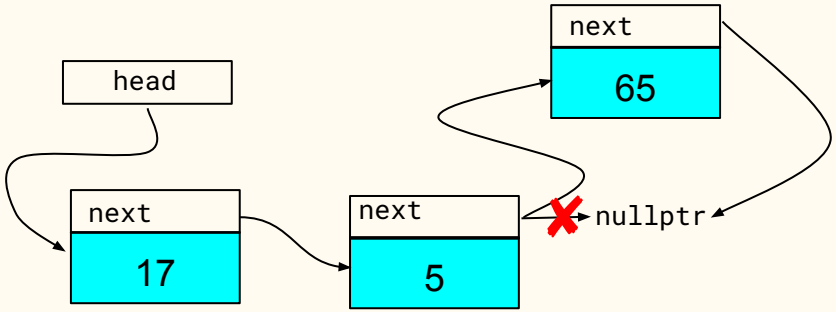
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            ---
        }
    }
}
```



adding a Node to end of list

Building a list

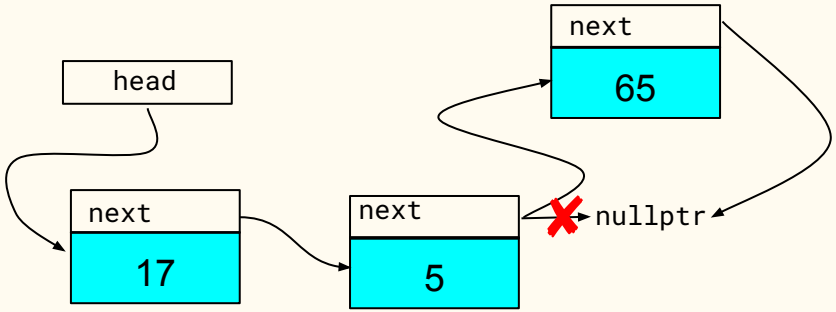
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            _17_
        }
    }
}
```



adding a Node to end of list

Which statement (replacing blank #17) will move the current pointer to the next node in the list on each iteration of the while loop?

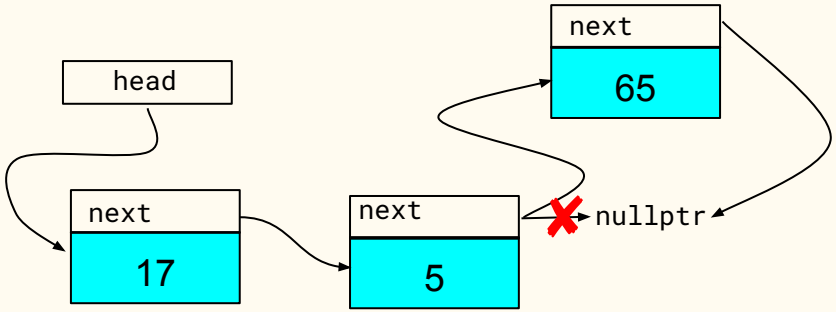
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            _17_
        }
    }
}
```



adding a Node to end of list

Building a list

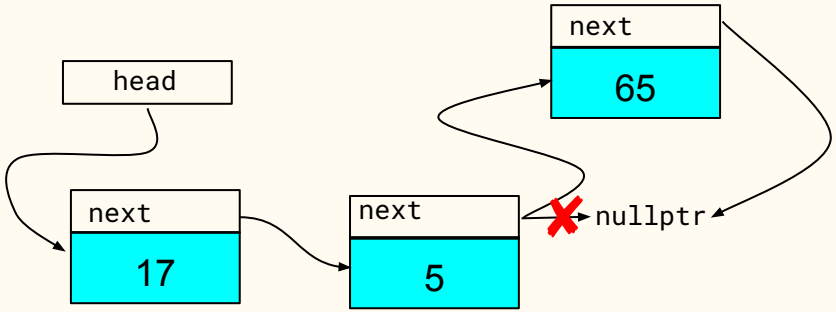
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
        // traverse list to current tail
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            curr = curr->next;
        }
    }
}
```



adding a Node to end of list

Building a list

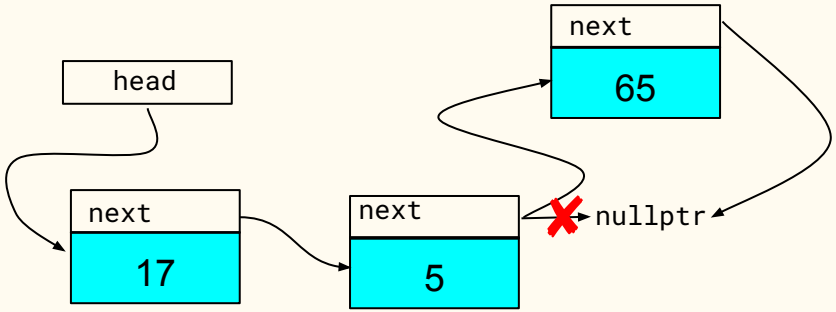
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            curr = curr->next;
        }
        // create a new Node and make tail
        ---
    }
}
```



adding a Node to end of list

Building a list

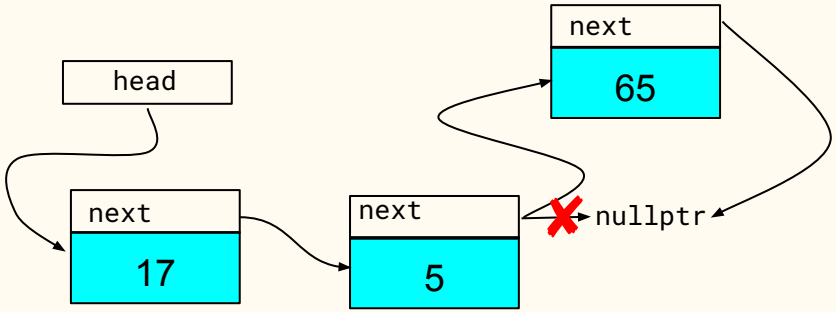
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            curr = curr->next;
        }
        // create a new Node and make tail
        _18_
    }
}
```



adding a Node to end of list

Which statement replaces blank #18 to instantiate a tail `Node` with `data` as the `data` member?

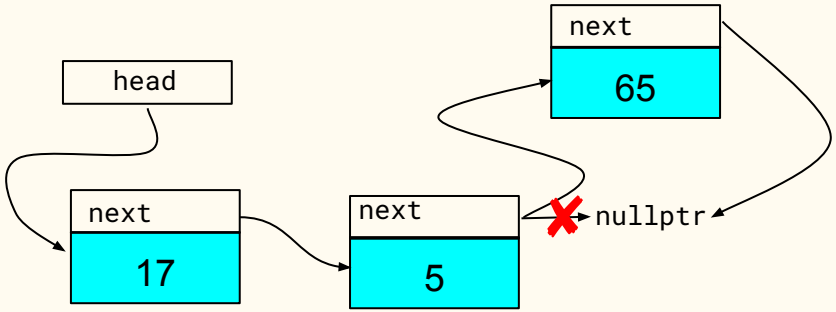
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            curr = curr->next;
        }
        // create a new Node and make tail
        _18_
    }
}
```



adding a Node to end of list

Building a list

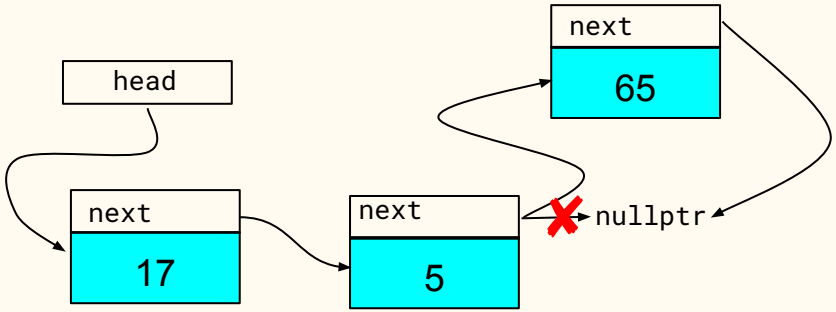
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            curr = curr->next;
        }
        // create a new Node and make tail
        curr->next = new Node(data);
    }
}
```



adding a Node to end of list

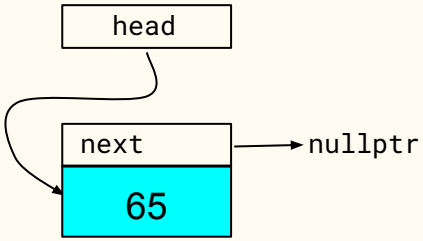
Building a list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    if (head_ptr == nullptr) {
        head_ptr = new Node(data);
    } else {
        Node* curr = head_ptr;
        while (curr->next != nullptr) {
            curr = curr->next;
        }
        curr->next = new Node(data);
    }
}
```

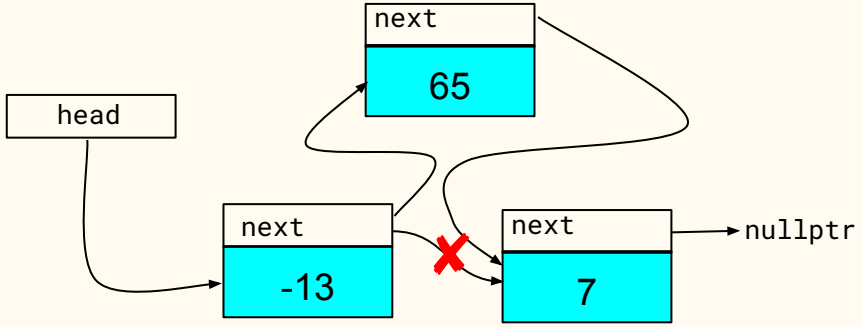


adding a Node to end of list

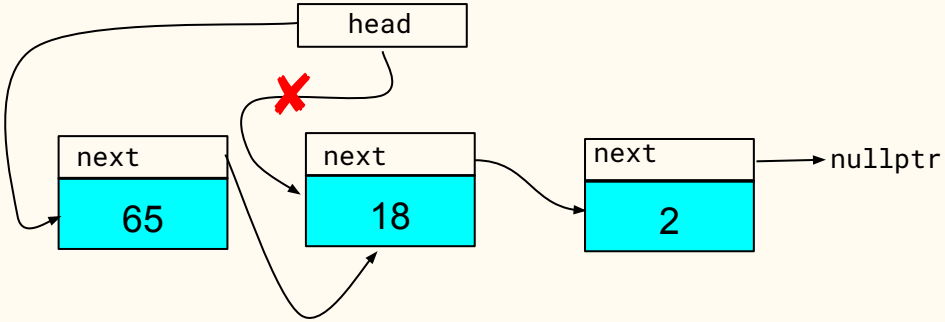
Building a list



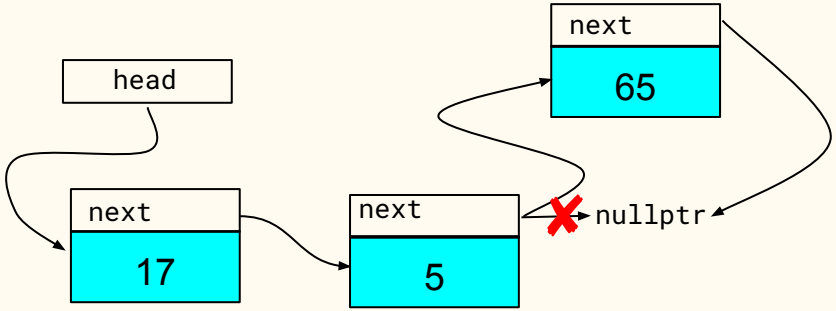
creating a list with one Node



inserting a Node into list

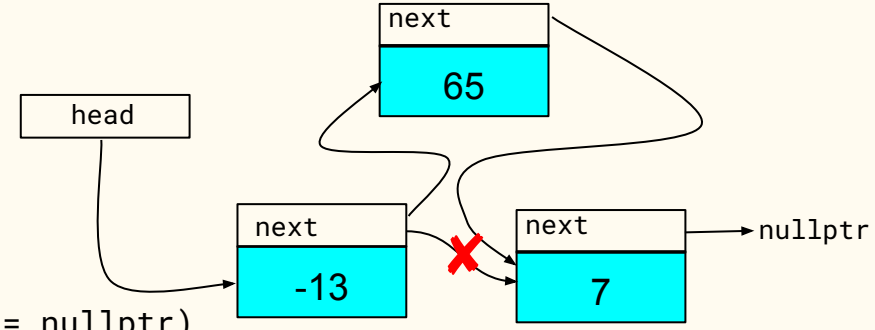


adding a Node to beginning of list



adding a Node to end of list

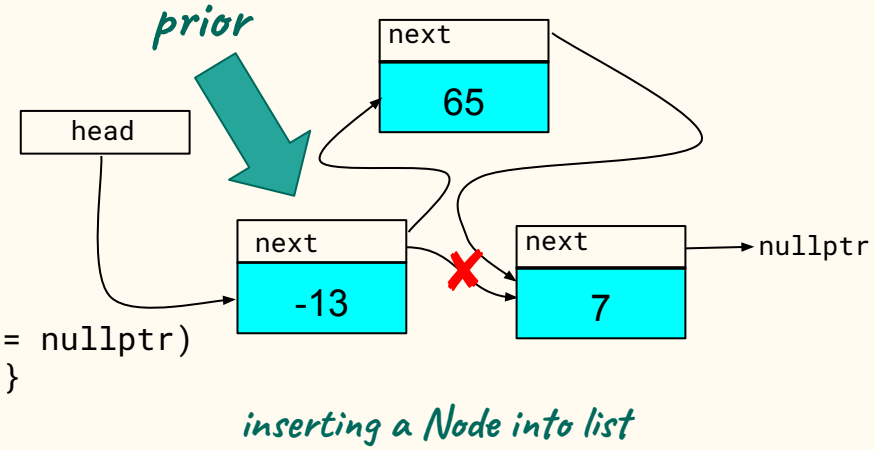
Building a list



inserting a Node into list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};


void add_node_to_list(___) { }
```

Building a list

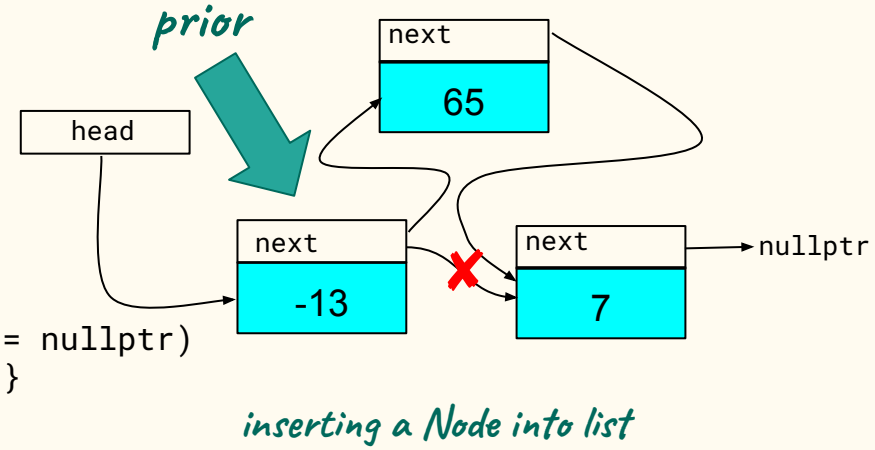



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) { }
```

```
add_node_to_list(head, 65);
```

Building a list

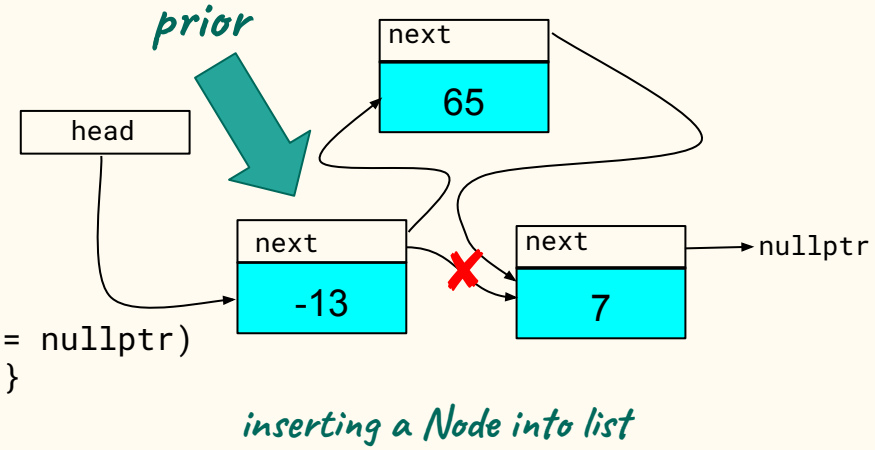



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ---
}
```

```
add_node_to_list(head, 65);
```

Building a list

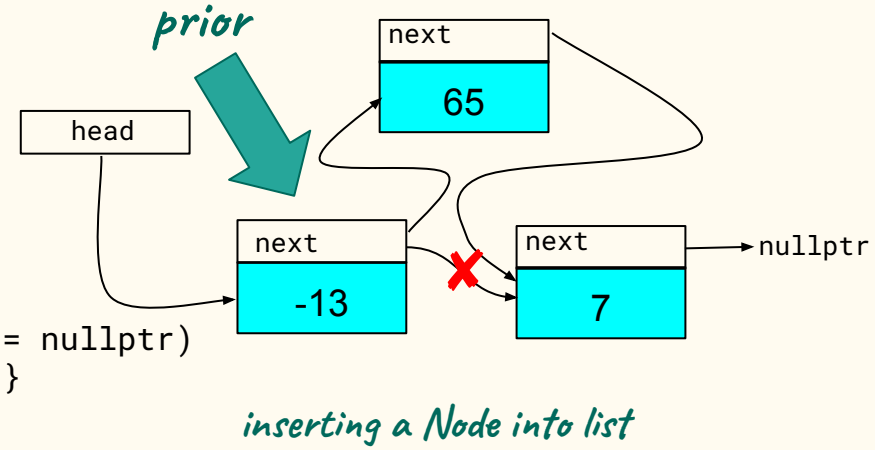



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = ___;
}
```

```
add_node_to_list(head, 65);
```

Building a list

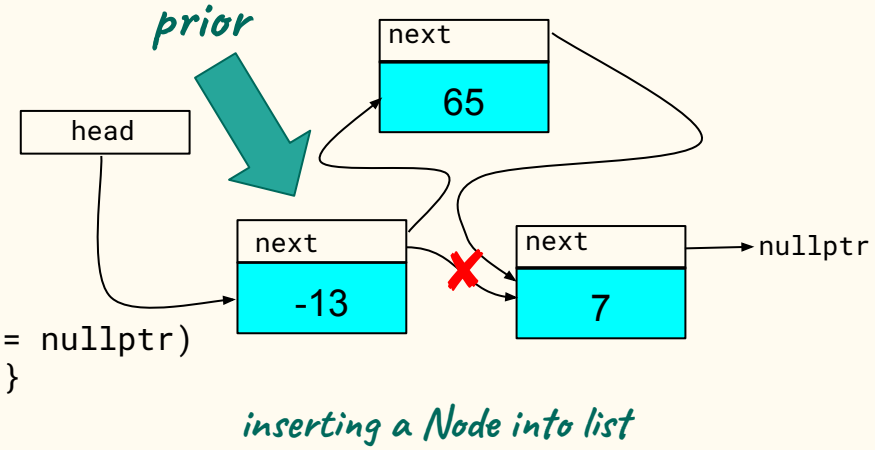



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(___, ___);
}
```

```
add_node_to_list(head, 65);
```

Building a list

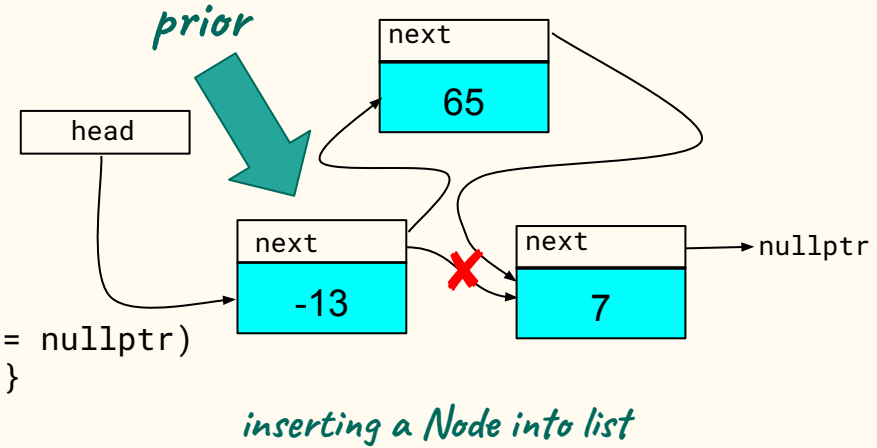



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(_19_, ___);
}
```

```
add_node_to_list(head, 65);
```

What replaces blank #19 when invoking the constructor for instantiating a Node?

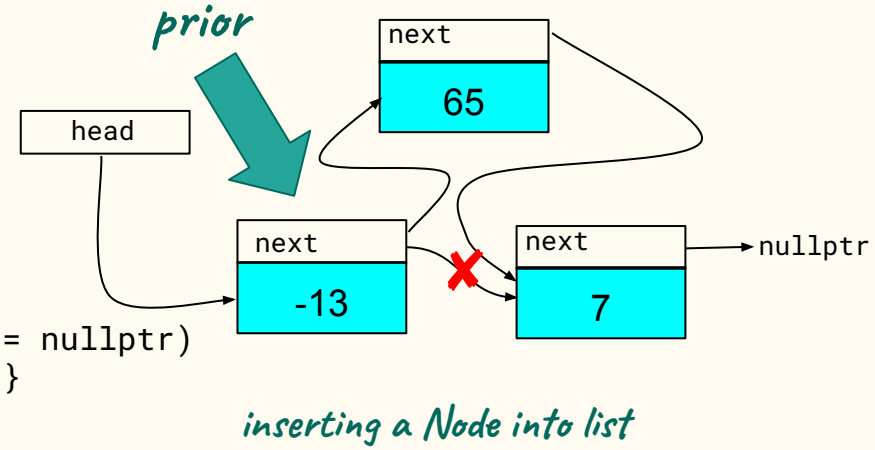



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(_19_, ___);
}
```

```
add_node_to_list(head, 65);
```

Building a list

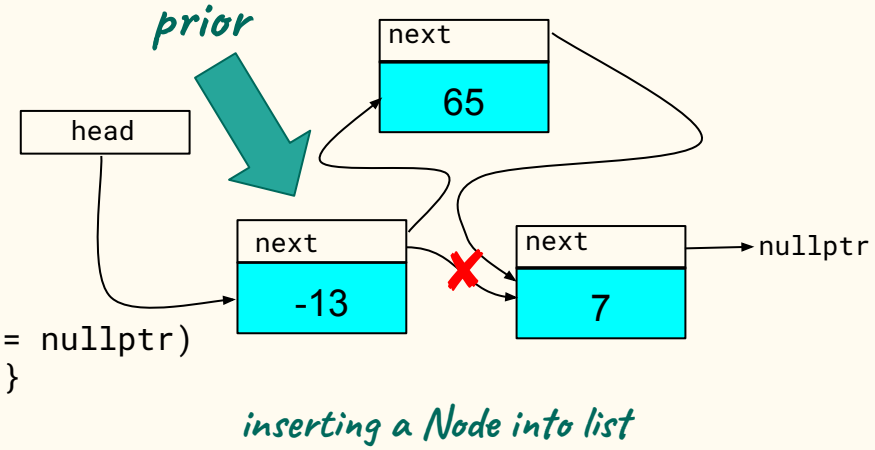



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(data, ___);
}
```

```
add_node_to_list(head, 65);
```

Building a list

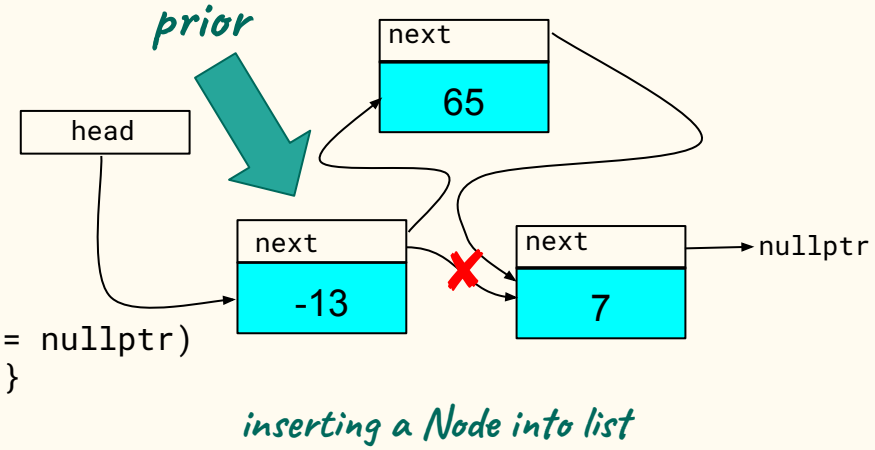



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(data, ___);
}
```

```
add_node_to_list(head, 65);
```

Building a list

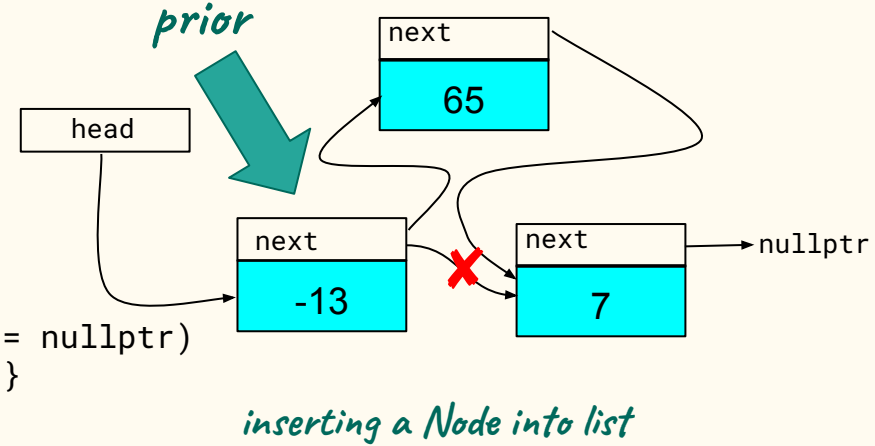



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(data, _20_);
}
```

```
add_node_to_list(head, 65);
```

Which expression replaces blank #20 for providing the address for the `next` pointer of the `Node` instance?

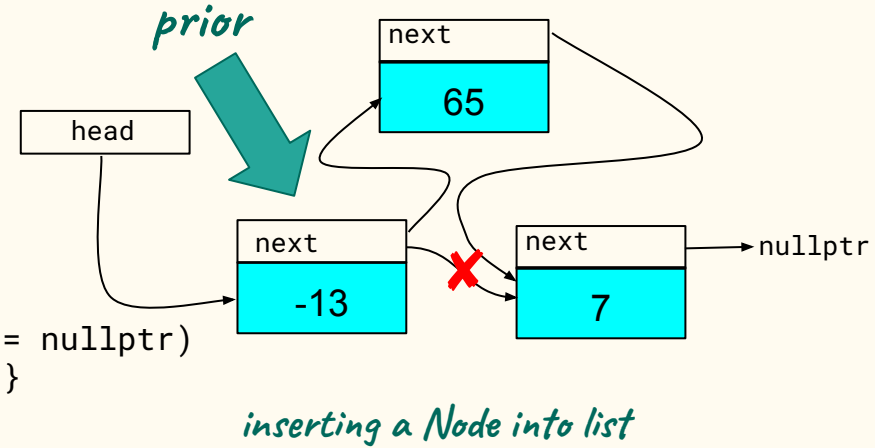



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(data, _20_);
}
```

```
add_node_to_list(head, 65);
```

Building a list

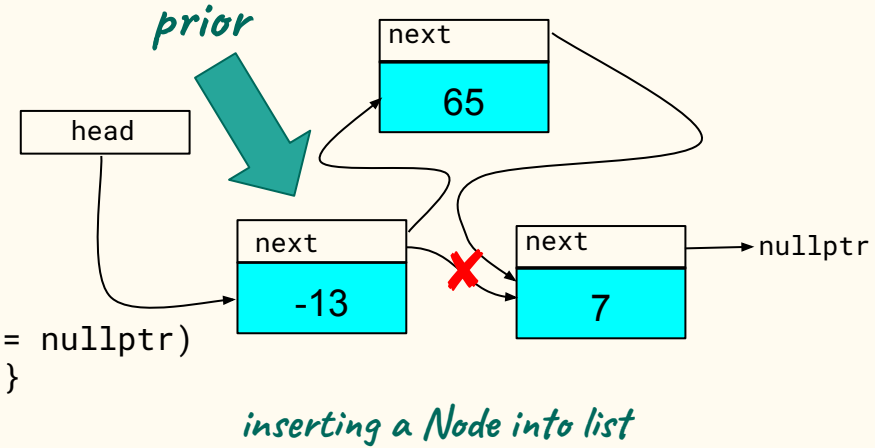



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    ___ = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

Building a list

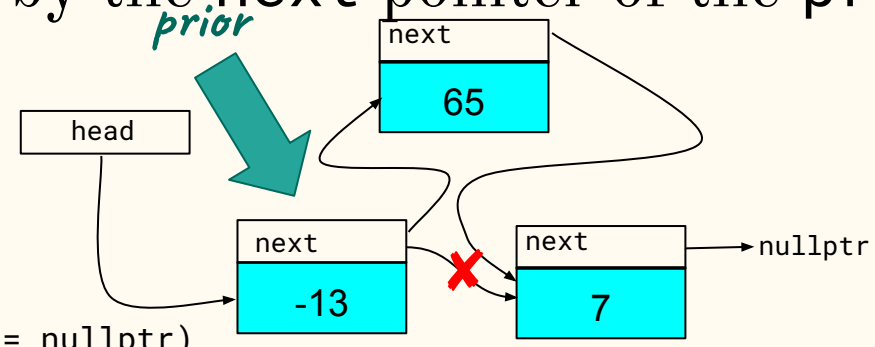



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    _21_ = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

Which expression replaces blank #21 for updating the address of the `Node` "pointed to" by the `next` pointer of the `prior` `Node`?



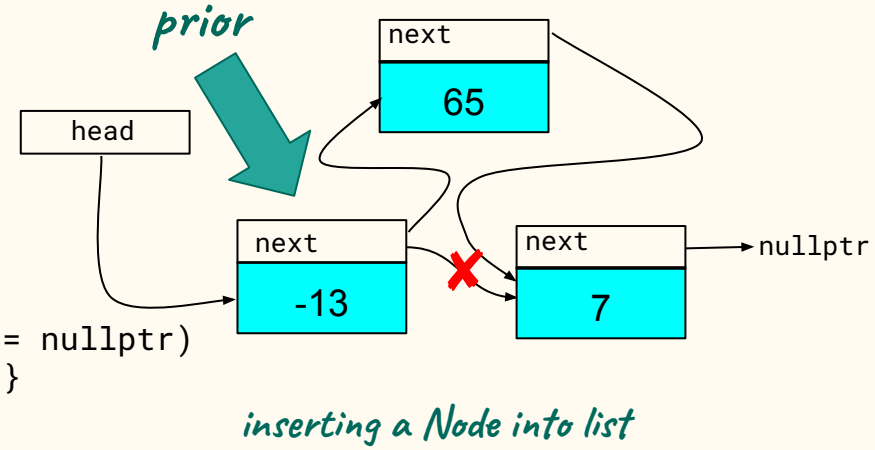
inserting a Node into list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    _21_ = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

Building a list

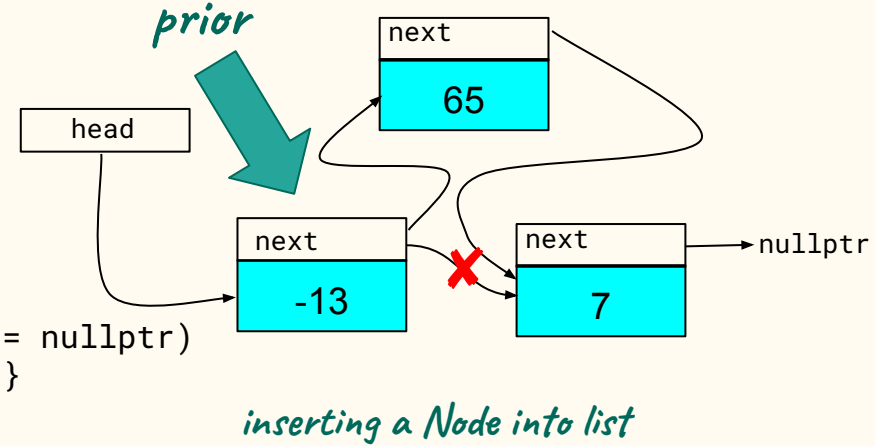



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    prior->next = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

Are there any restrictions on the argument that can be passed to the `prior` parameter of the `add_node_to_list()` function?



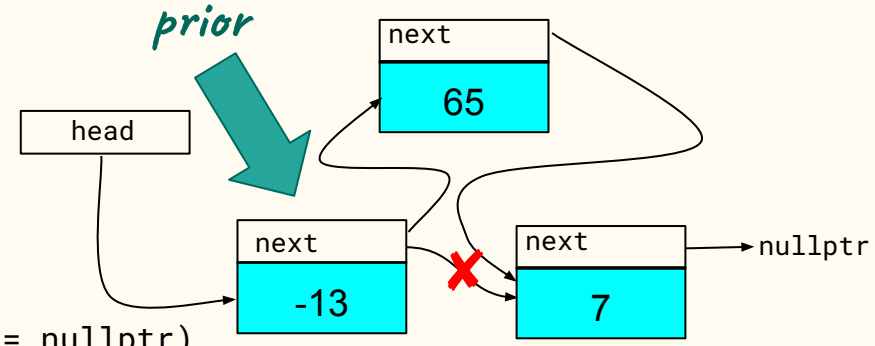
inserting a Node into list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_node_to_list(Node* prior, int data) {
    prior->next = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

Building a list



inserting a Node into list

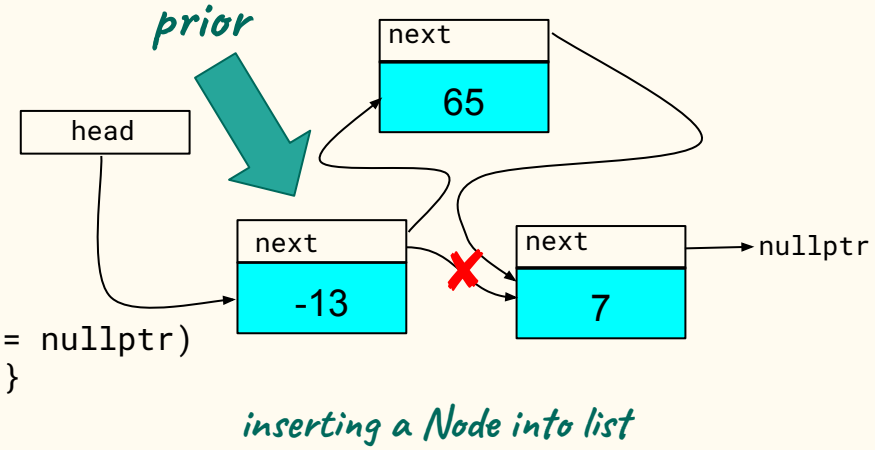
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

prior argument can never be nullptr

```
void add_node_to_list(Node* prior, int data) {
    prior->next = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

Building a list

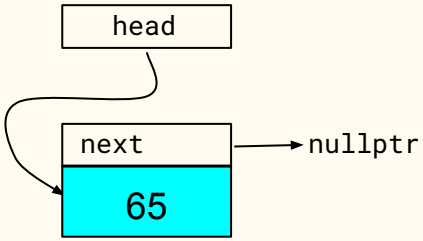



```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

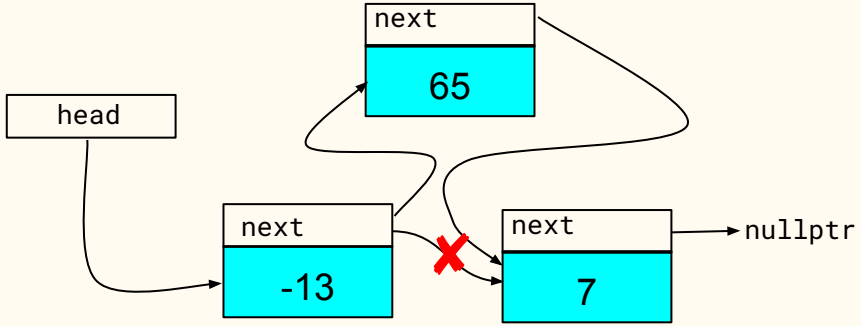
```
void add_node_to_list(Node* prior, int data) {
    prior->next = new Node(data, prior->next);
}
```

```
add_node_to_list(head, 65);
```

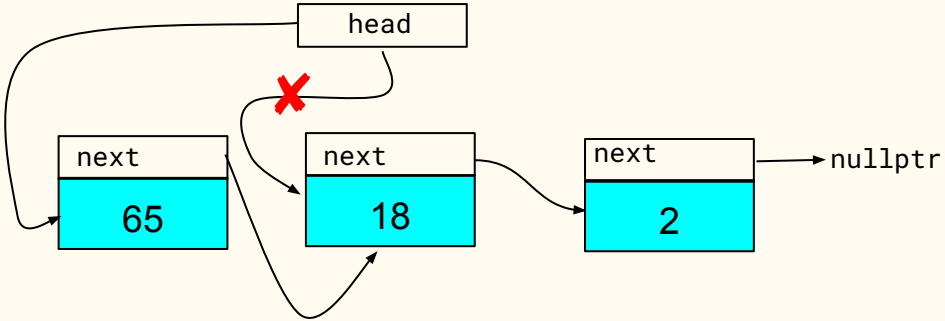
Building a list



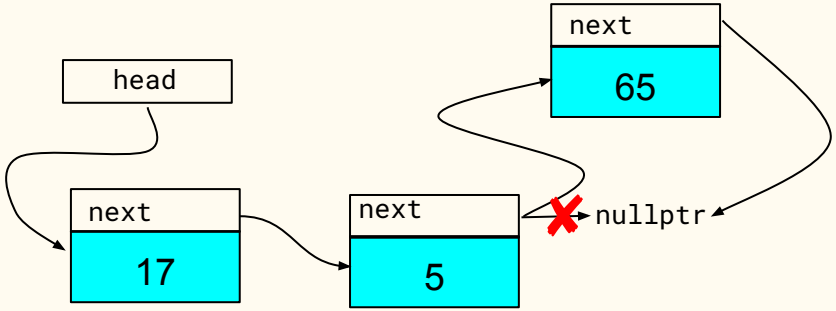
creating a list with one Node



inserting a Node into list



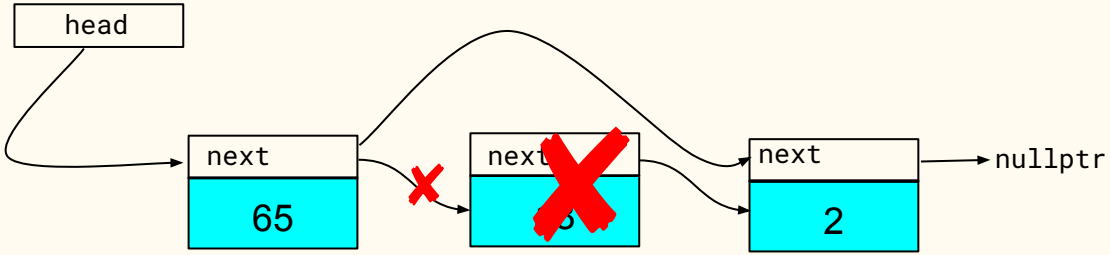
adding a Node to beginning of list



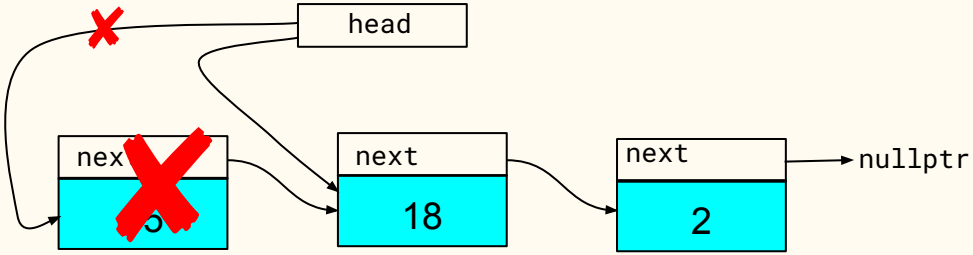
adding a Node to end of list

Removing list nodes

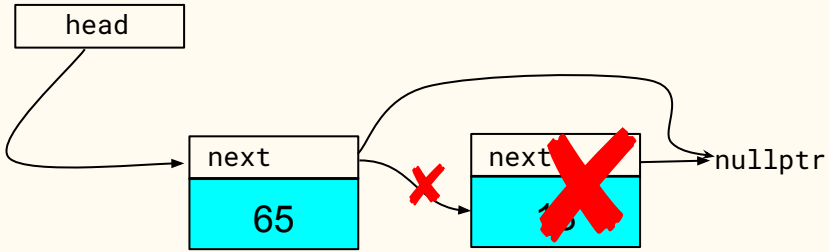
Removing a `Node` from list



removing interior Node



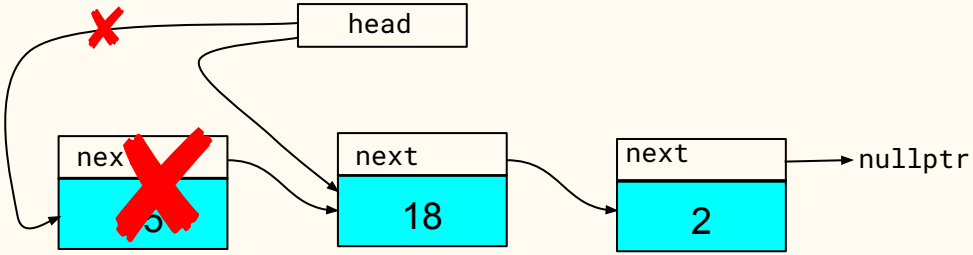
removing the head Node



removing the tail Node

Removing a `Node` from list

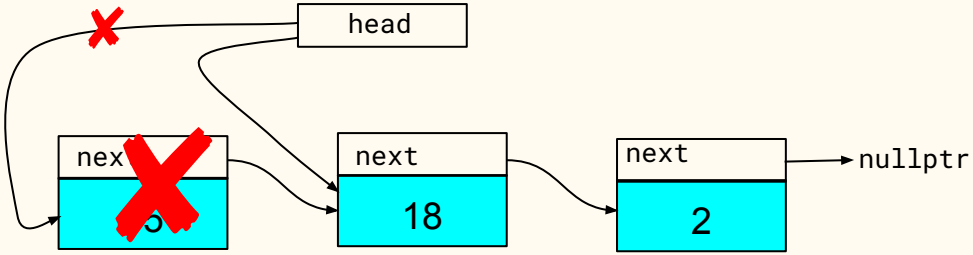
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

___ remove_head_from_list() {}
```



removing the head Node

Removing a `Node` from list

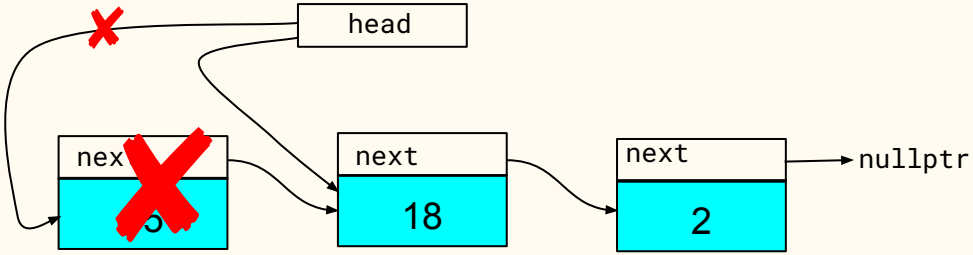
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

___ remove_head_from_list(___ ___) {}
```



removing the head Node

Removing a `Node` from list

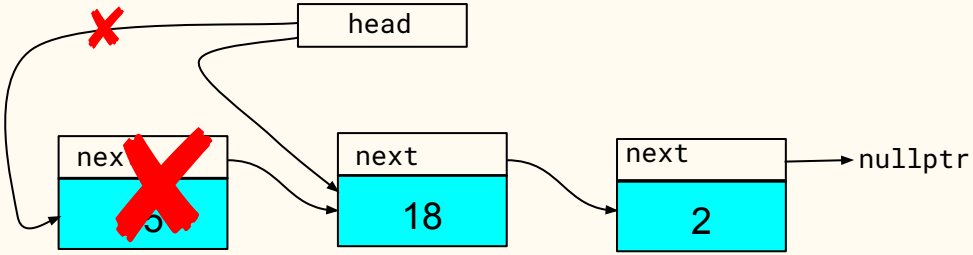
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

___ remove_head_from_list(Node*& ___) {}
```



removing the head Node

Removing a `Node` from list

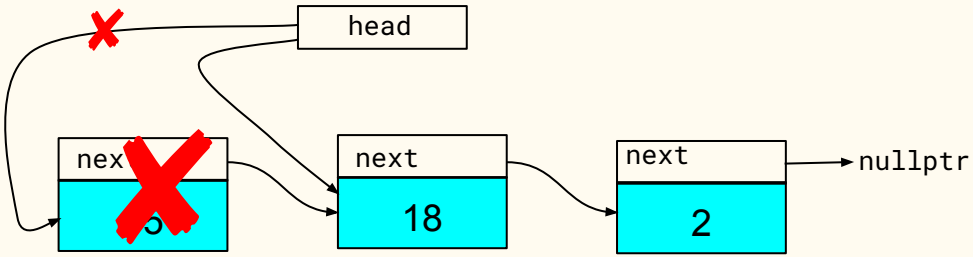
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

___ remove_head_from_list(Node*& head_ptr) {}
```



removing the head Node

Removing a `Node` from list

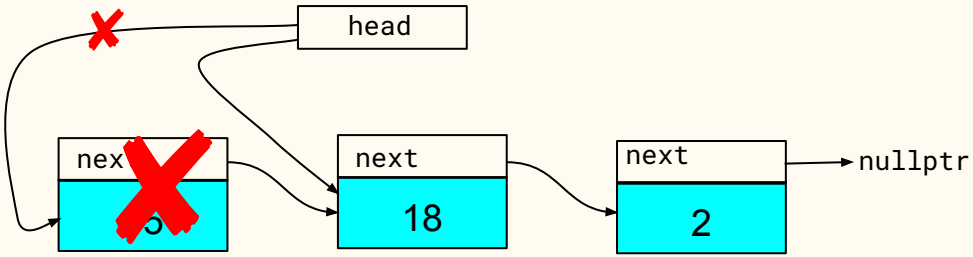
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

_21_ remove_head_from_list(Node*& head_ptr) {}
```



removing the head Node

Which return type replaces blank #21 to indicate the success or failure of the removal of the head Node?

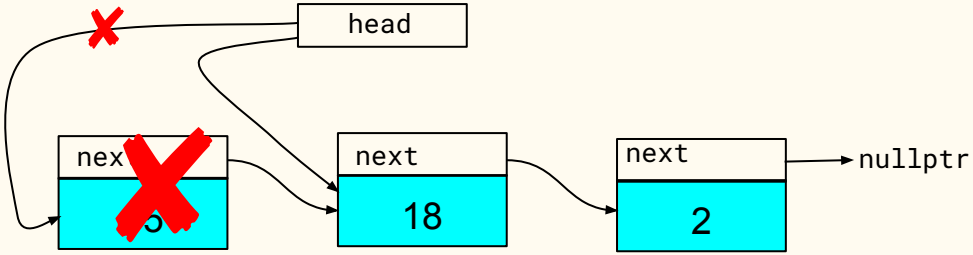
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

_21_ remove_head_from_list(Node*& head_ptr) {}
```



removing the head Node

Removing a `Node` from list

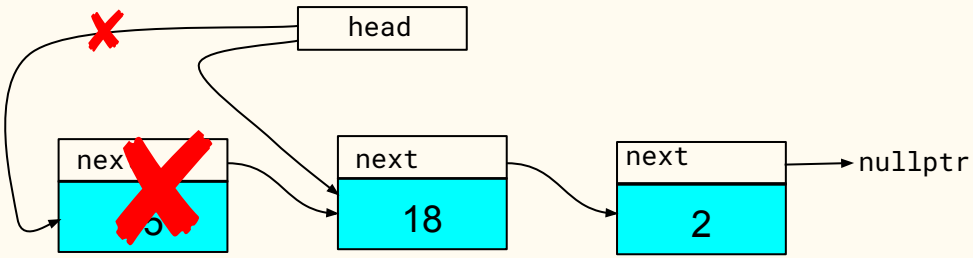
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    // store original head address
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

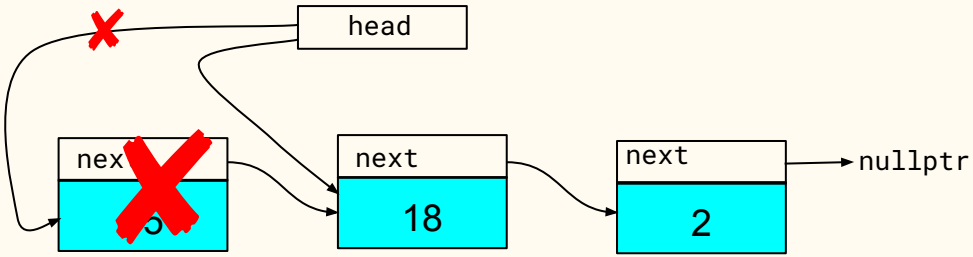
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    // store original head address
    ___
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

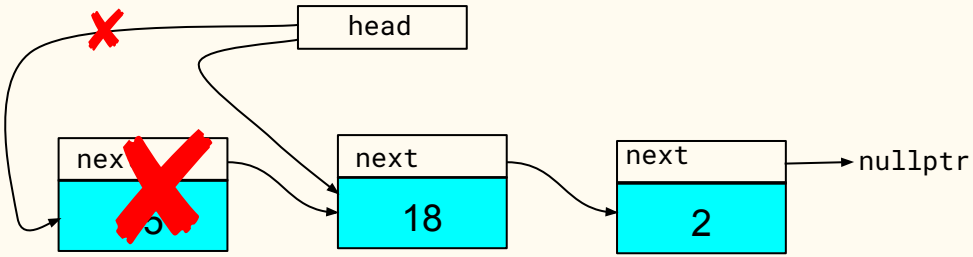
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    // store original head address
    Node* old_head = ___;
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

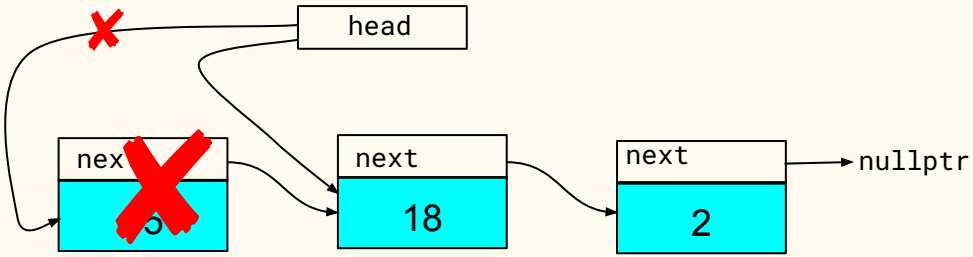
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    // store original head address
    Node* old_head = _22_;
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

What replaces blank #22 to assign the address of the original head `Node` to the `Node* old_head`?

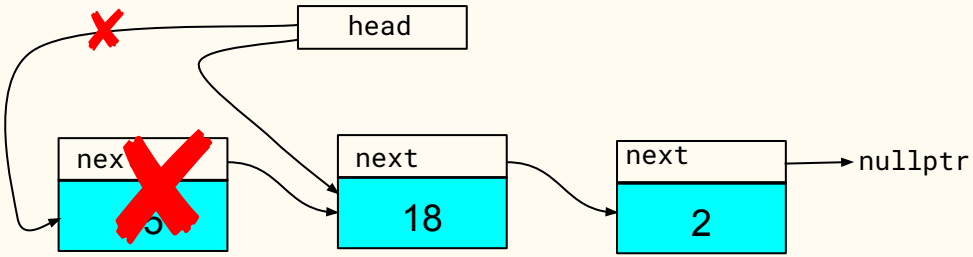
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    // store original head address
    Node* old_head = _22_;
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

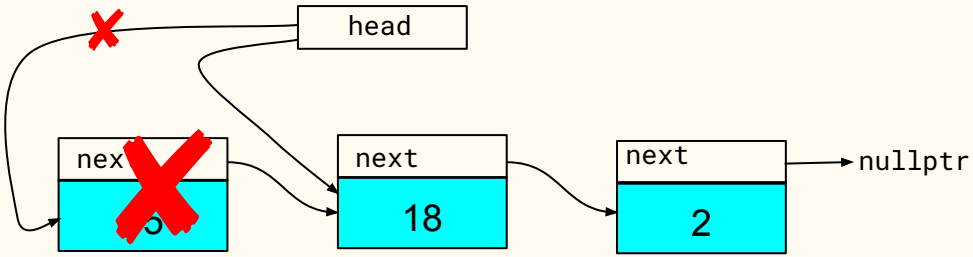
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    // store original head address
    Node* old_head = head_ptr;
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

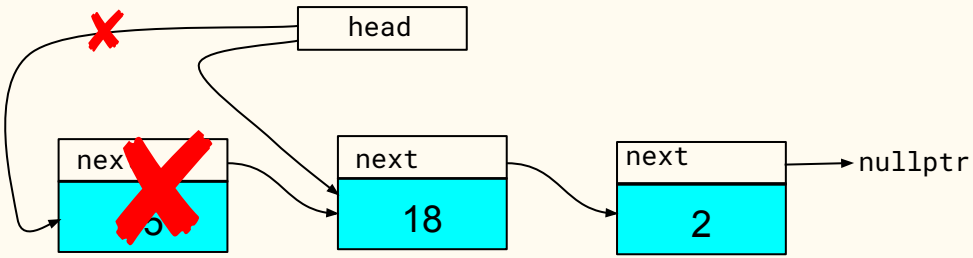
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    // set head_ptr to point to Node after head
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

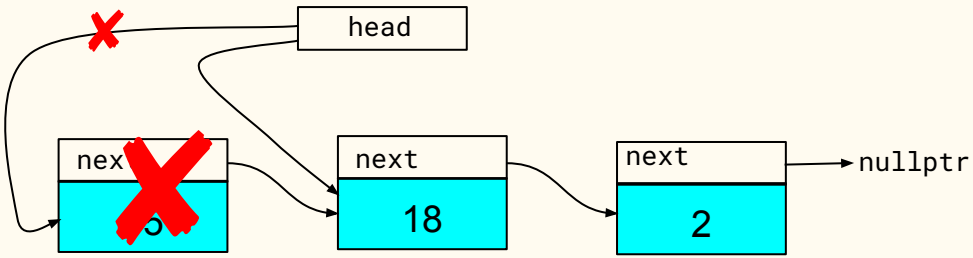
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    // set head_ptr to point to Node after head
    ___
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

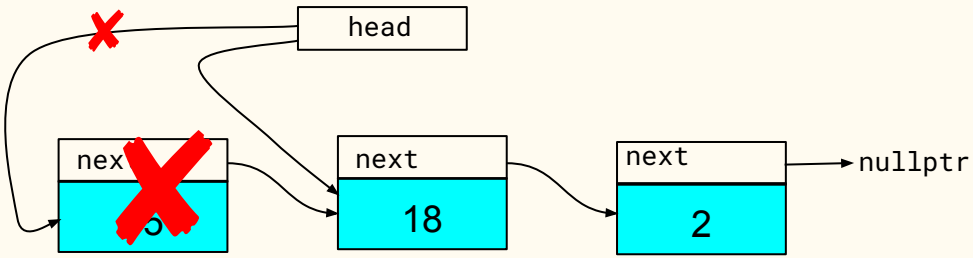
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    // set head_ptr to point to Node after head
    _23_
    // free old head Node memory
}
```



removing the head Node

Which statement replaces blank #23 to modify `head_ptr` to point to the `Node` after the old head `Node`?

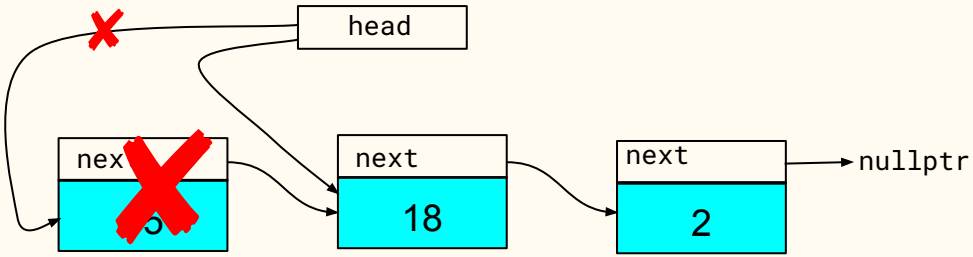
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    // set head_ptr to point to Node after head
    _23_
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

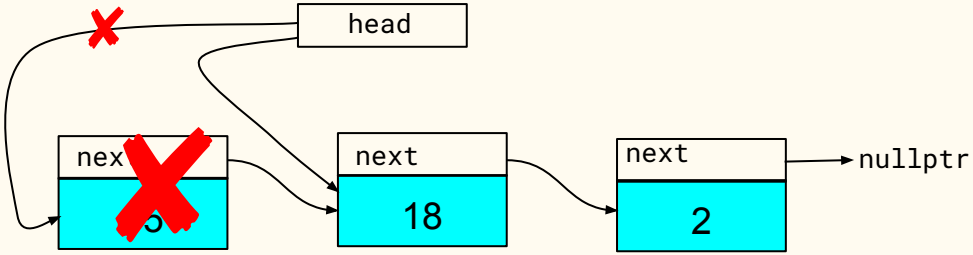
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    // set head_ptr to point to Node after head
    head_ptr = head_ptr->next;
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

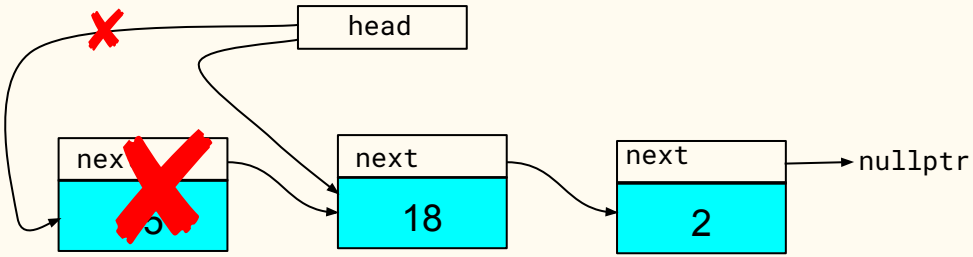
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    head_ptr = head_ptr->next;
    // free old head Node memory
}
```



removing the head Node

Removing a `Node` from list

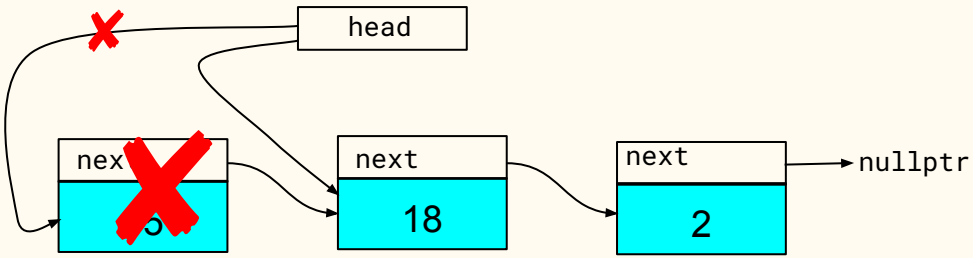
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    head_ptr = head_ptr->next;
    // free old head Node memory
    ---
}
```



removing the head Node

Removing a `Node` from list

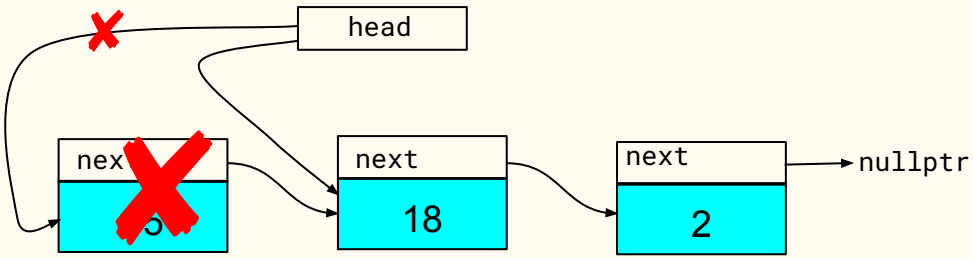
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    head_ptr = head_ptr->next;
    // free old head Node memory
    _24_
}
```



removing the head Node

Which statement replaces blank #24 to free the memory allocated for the old head Node of the list?

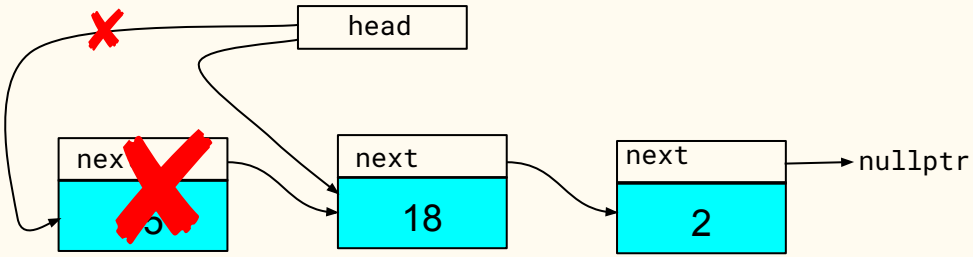
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    head_ptr = head_ptr->next;
    // free old head Node memory
    _24_
}
```



removing the head Node

Removing a `Node` from list

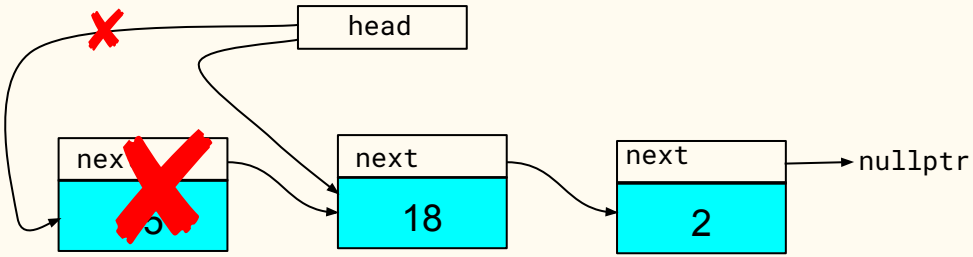
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;
    head_ptr = head_ptr->next;
    // free old head Node memory
    delete old_head;
}
```



removing the head Node

Removing a `Node` from list

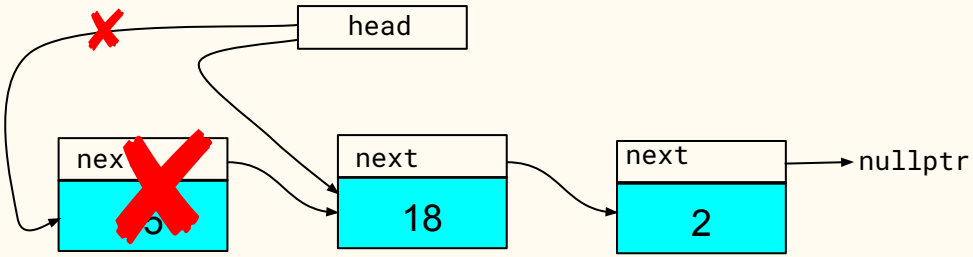
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;

    head_ptr = head_ptr->next;
    delete old_head;
}
```



removing the head Node

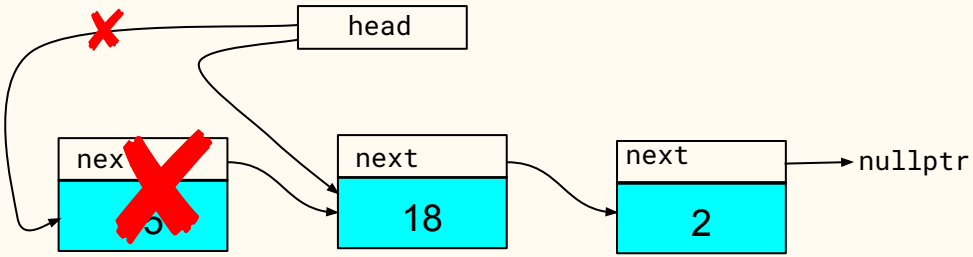
Removing a `Node` from list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    Node* old_head = head_ptr;

    head_ptr = head_ptr->next;
    delete old_head;
}
```



removing the head Node

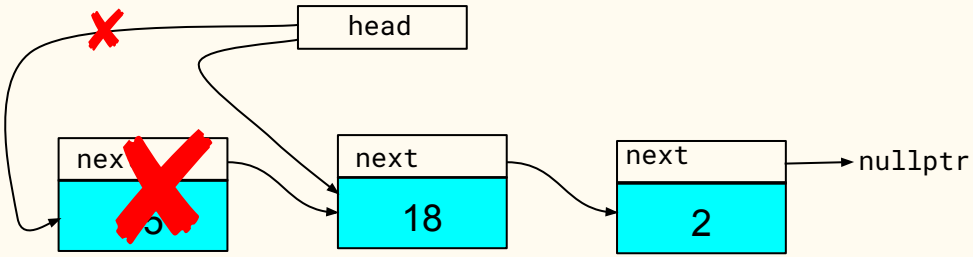
Removing a `Node` from list



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (___) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
    }
}
```

Removing a `Node` from list

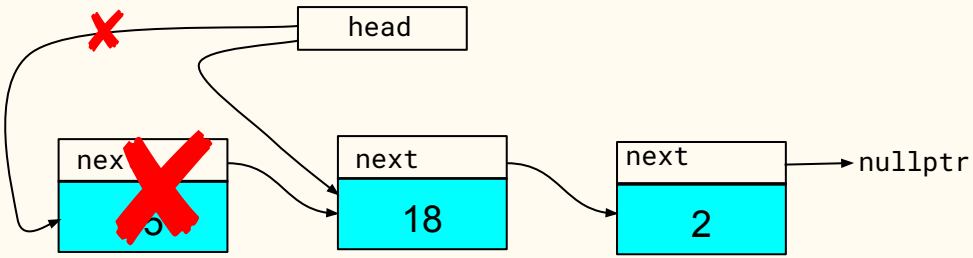


removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (_25_) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
    }
}
```

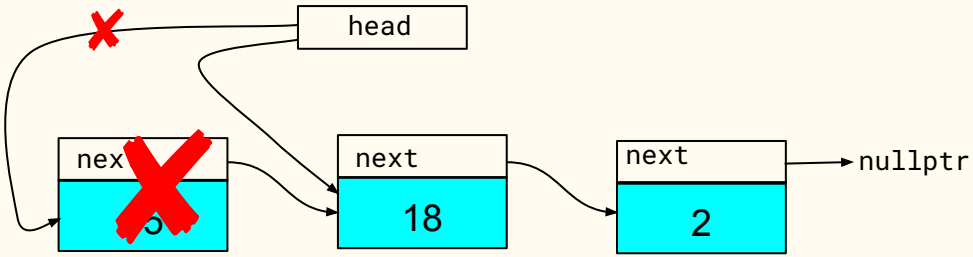
Which boolean expression replaces blank #25 so that the remove head operation only proceeds when the `head_ptr` is not the `nullptr`?

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (_25_) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
    }
}
```



removing the head Node

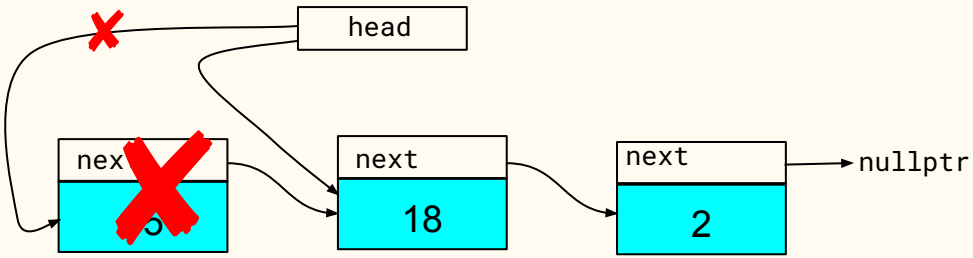
Removing a `Node` from list



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
    }
}
```

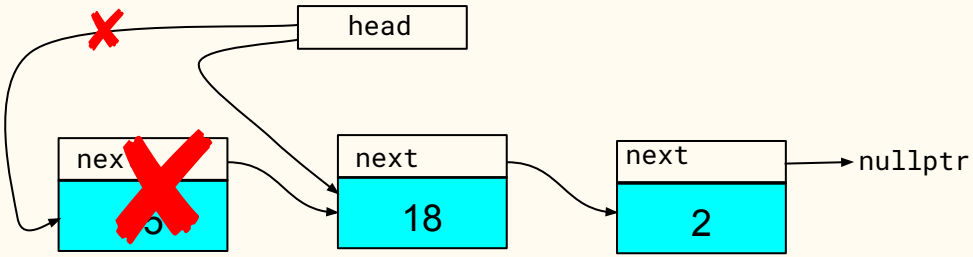
Removing a `Node` from list



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;

        ---
    }
}
```

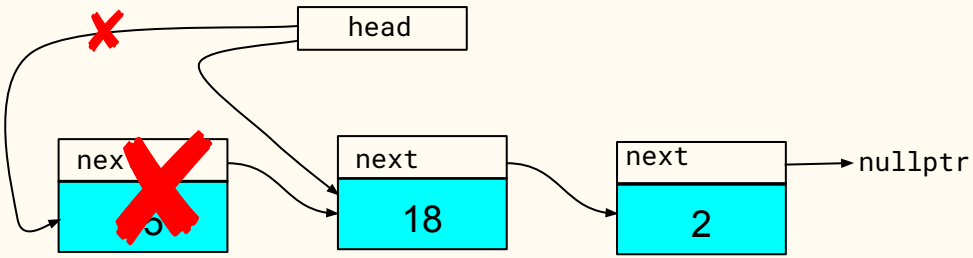
Removing a `Node` from list



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
        _26_
    }
}
```

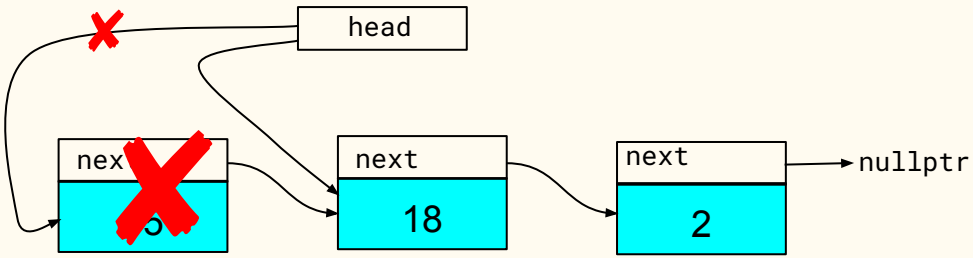
Which statement replaces blank #26 to indicate that removal of the head `Node` was successful?



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
        _26_
    }
}
```

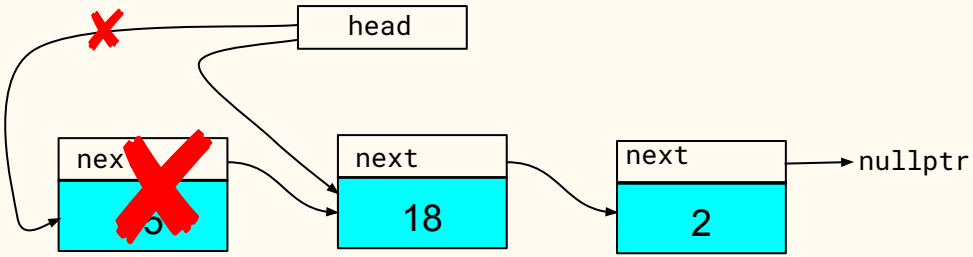
Removing a `Node` from list



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
        return true;
    }
}
```

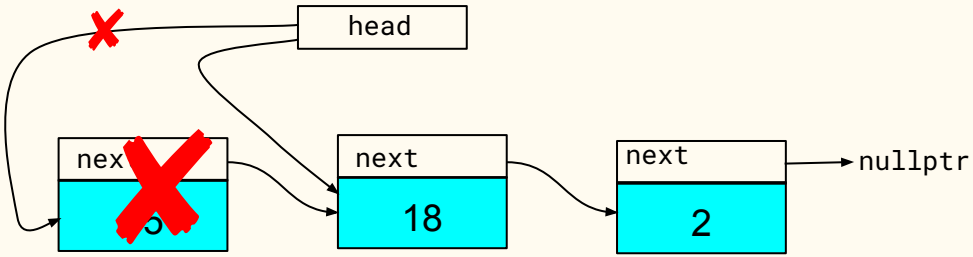
Removing a `Node` from list



removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
        return true;
    }
    ---
}
```

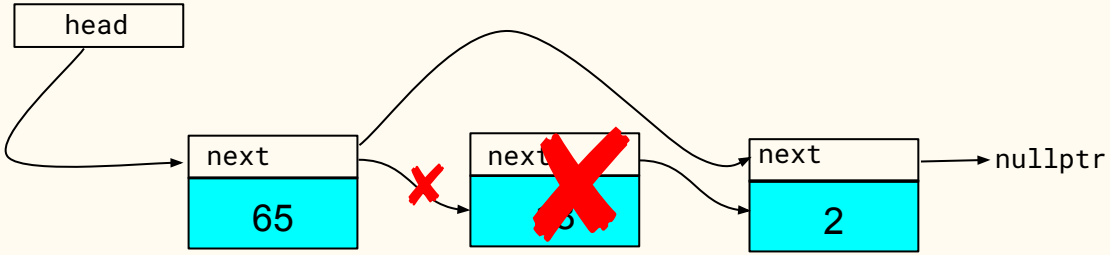
Removing a `Node` from list



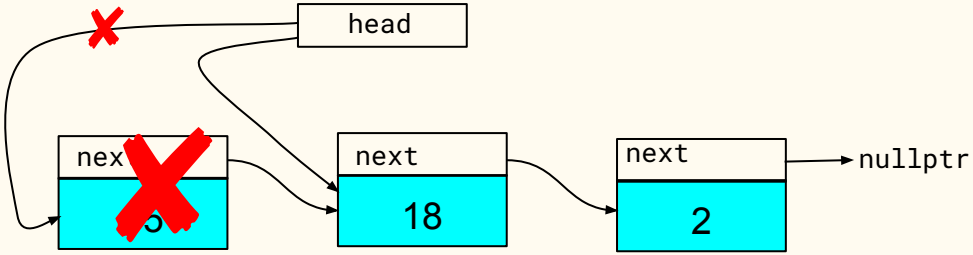
removing the head Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

bool remove_head_from_list(Node*& head_ptr) {
    if (head_ptr != nullptr) {
        Node* old_head = head_ptr;

        head_ptr = head_ptr->next;
        delete old_head;
        return true;
    }
    return false;
}
```

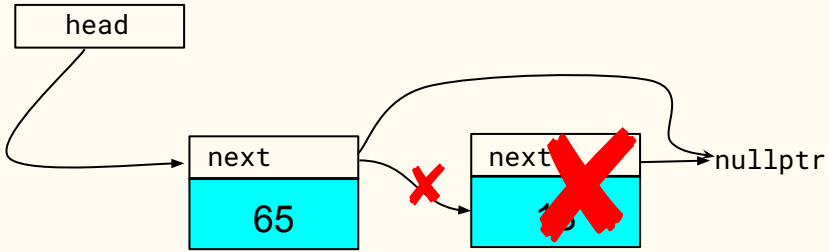
Removing a `Node` from list



removing interior Node



removing the head Node

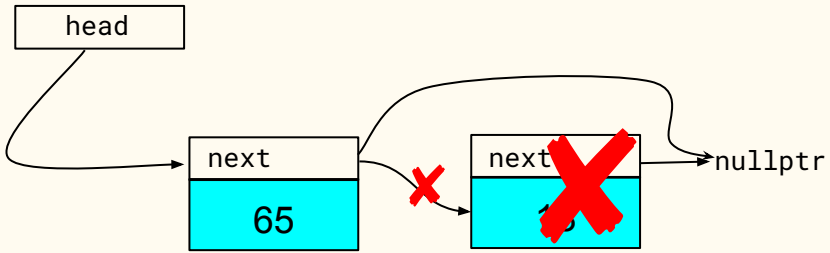


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list() { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

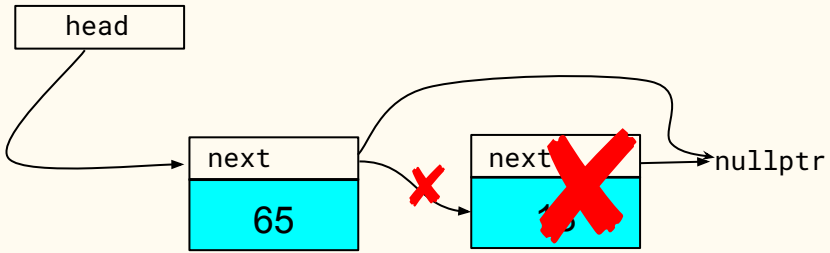


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(___ ___) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

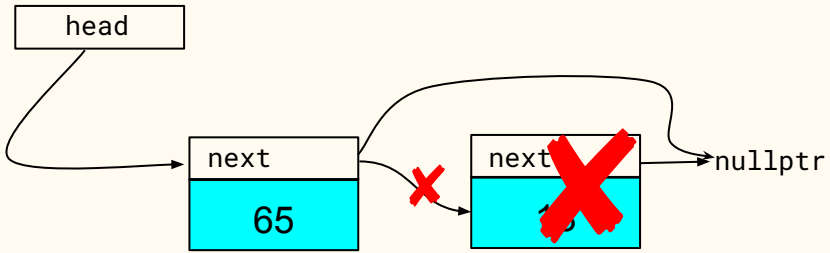


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& ___) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

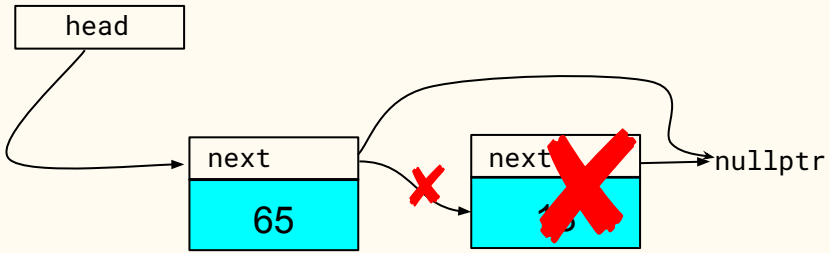


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) { }
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

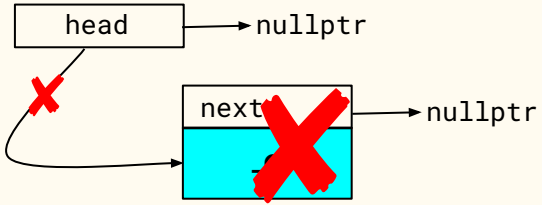


removing the tail Node

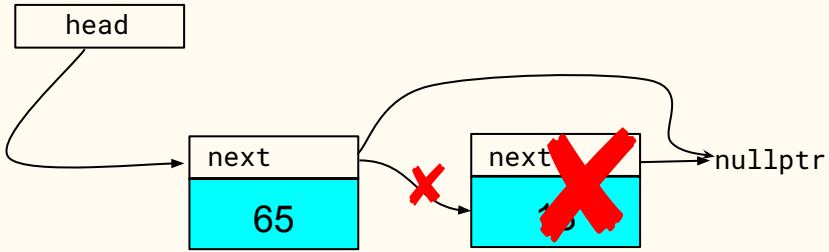
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

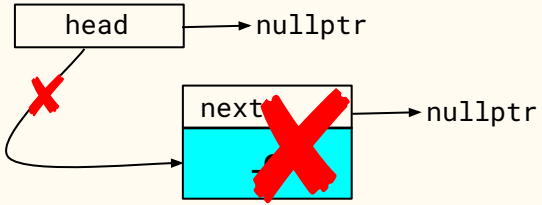


removing the tail Node

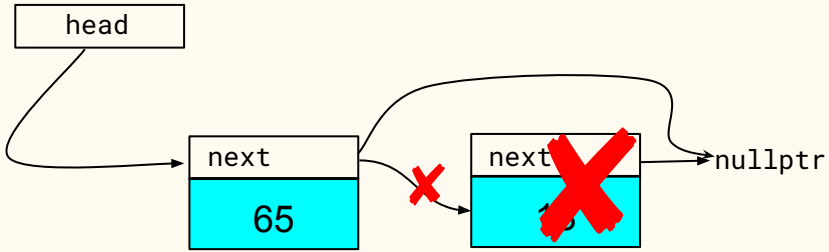
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    ___
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

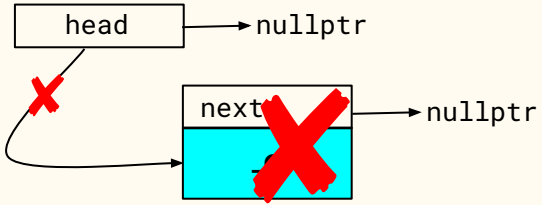


removing the tail Node

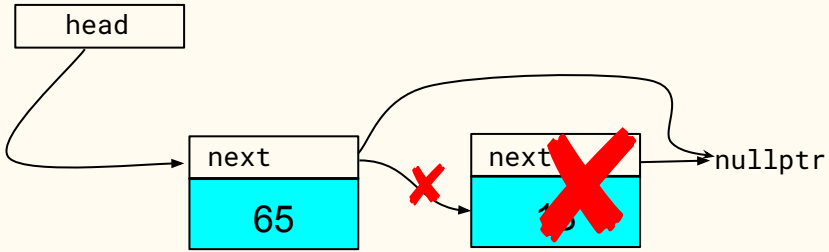
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (_28_) {

    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

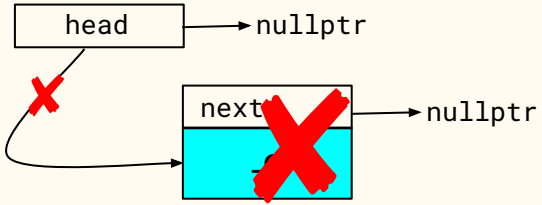


removing the tail Node

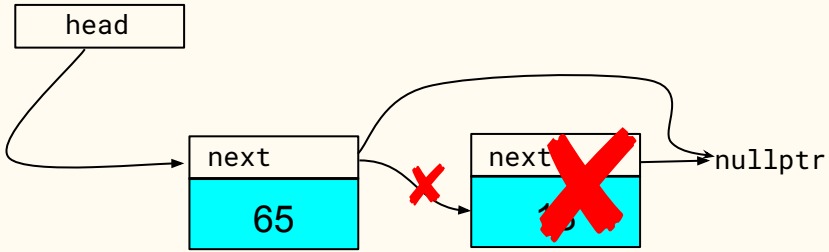
Which boolean expression replaces blank #28 to check for the parameter `head_ptr` being passed `nullptr`?

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (_28_) {

    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

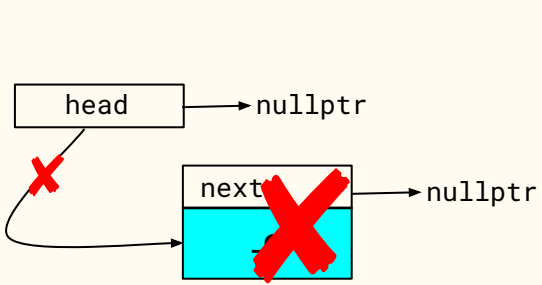


removing the tail Node

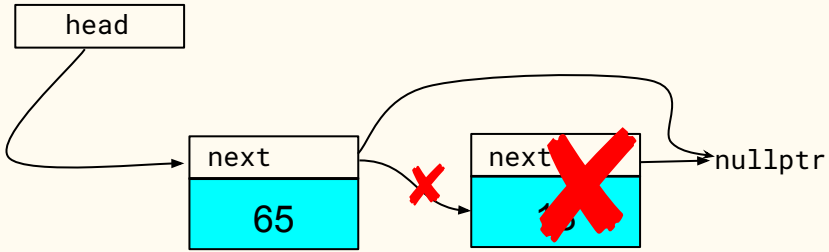
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (head_ptr == nullptr) {

    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

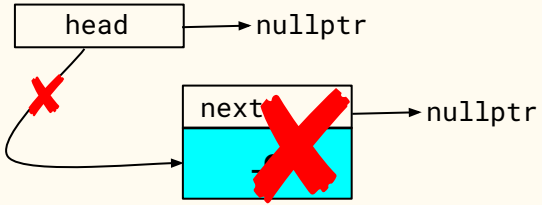


removing the tail Node

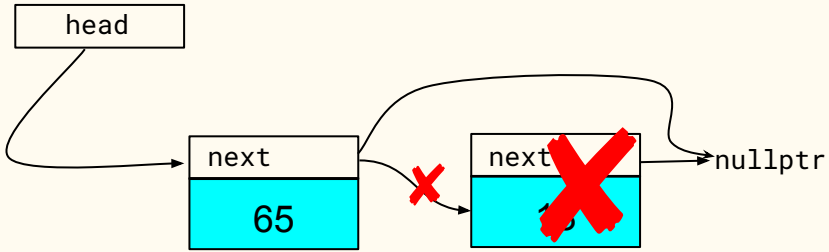
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (head_ptr == nullptr) {
        ---
    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

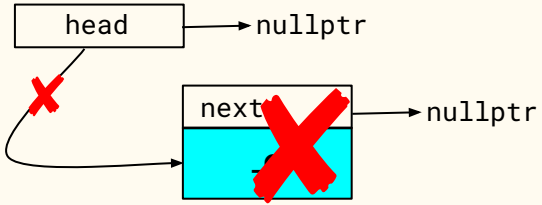


removing the tail Node

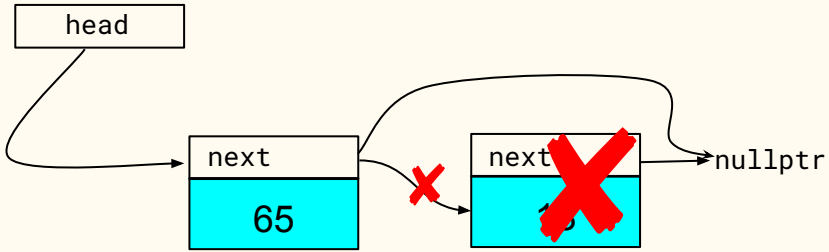
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (head_ptr == nullptr) {
        _29_
    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

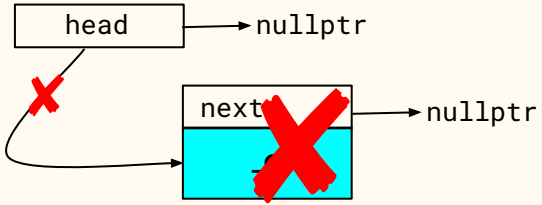


removing the tail Node

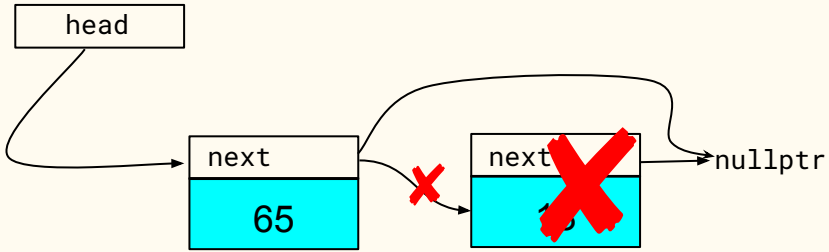
Which statement replaces blank #29 to indicate that the `Node` is not removed when `head_ptr` is passed `nullptr`?

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (head_ptr == nullptr) {
        _29_
    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

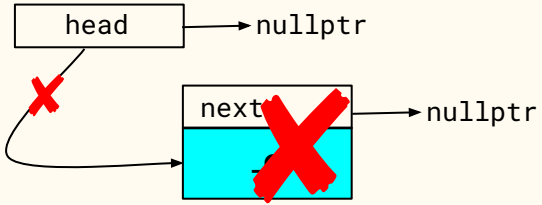


removing the tail Node

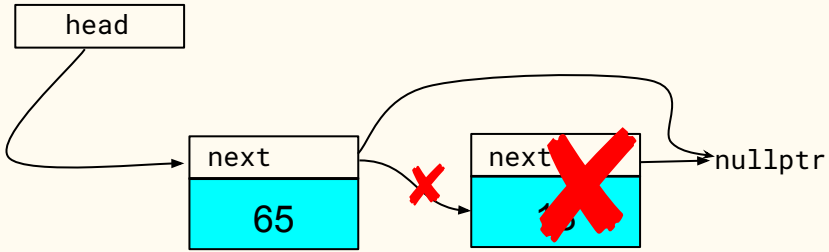
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    // check for nullptr passed as head_ptr
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

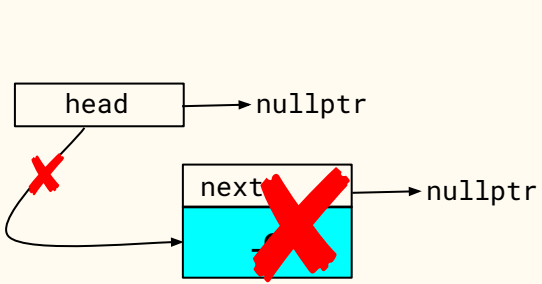


removing the tail Node

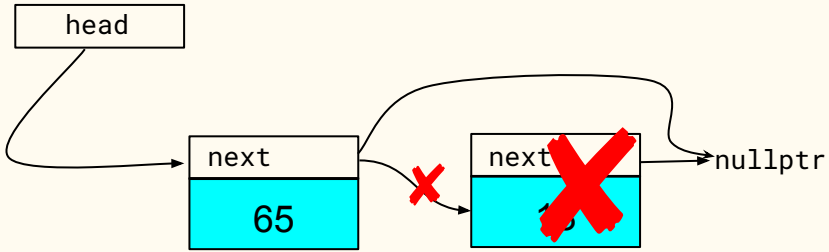
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

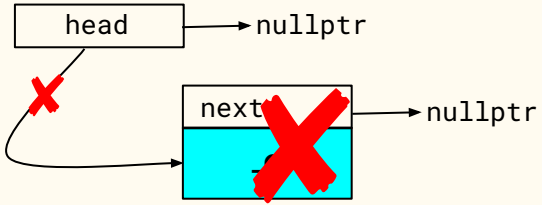


removing the tail Node

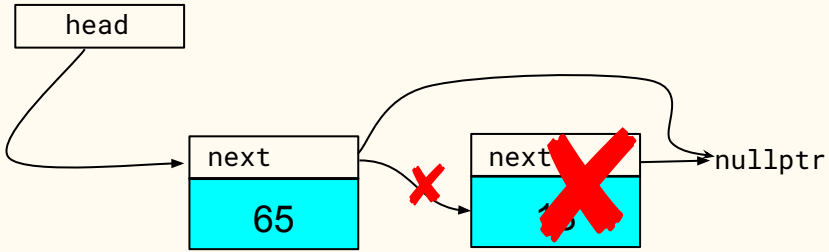
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    ___
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

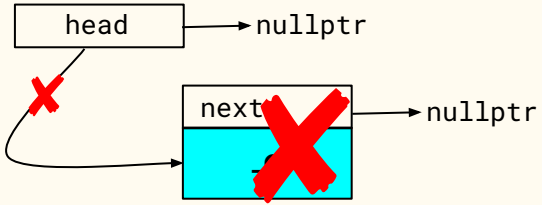


removing the tail Node

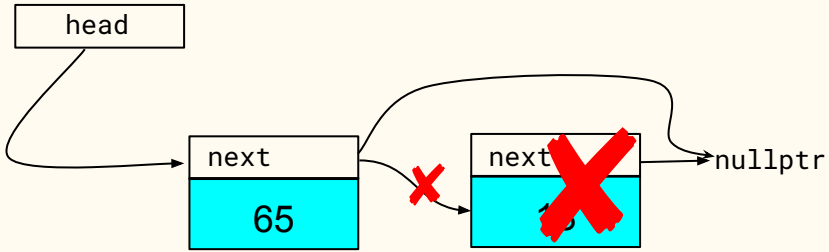
How do we determine when the list's head `Node` is also the list's tail `Node`?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    ___
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

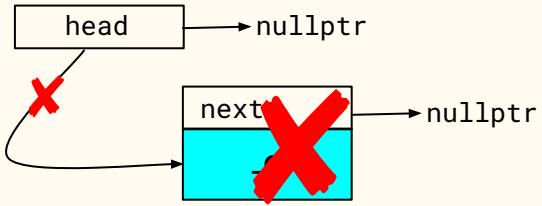


removing the tail Node

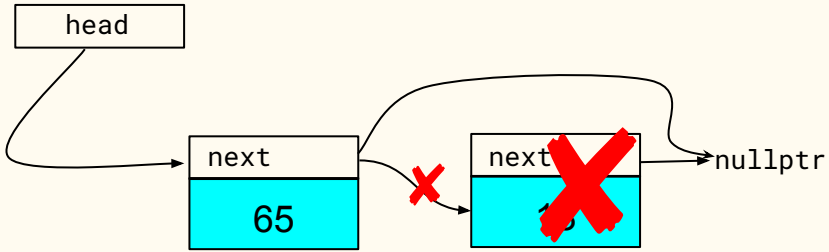
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (___) {

    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

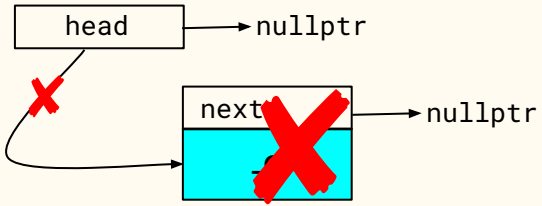


removing the tail Node

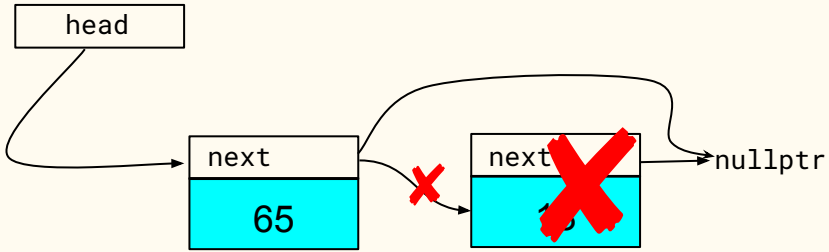
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (_30_) {

    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

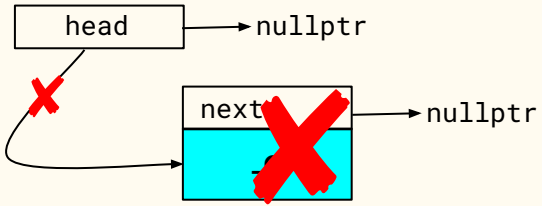


removing the tail Node

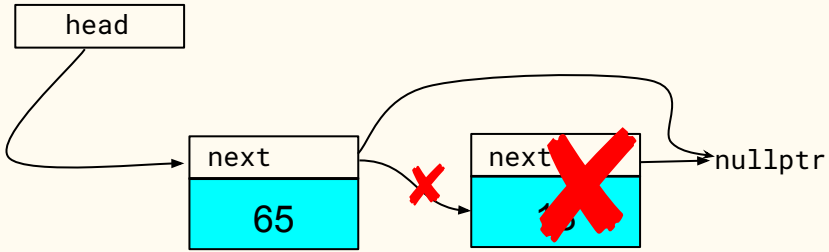
Which boolean expression replaces blank #30 to determine when the list consists of a single Node?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (_30_) {

    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

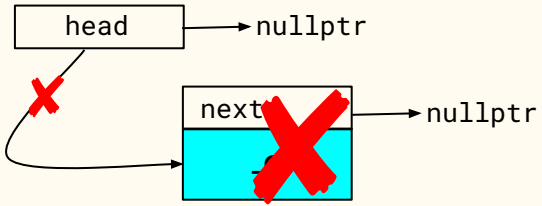


removing the tail Node

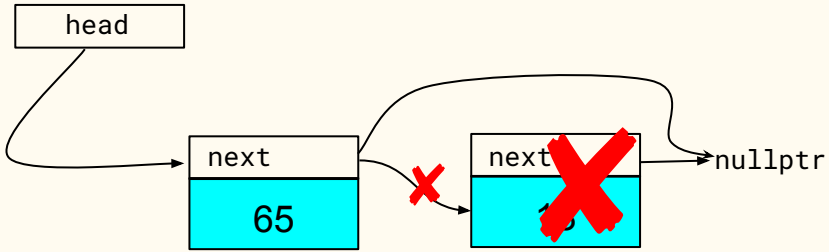
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {

    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

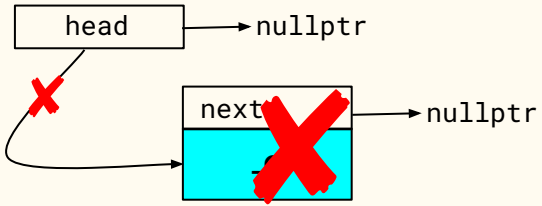


removing the tail Node

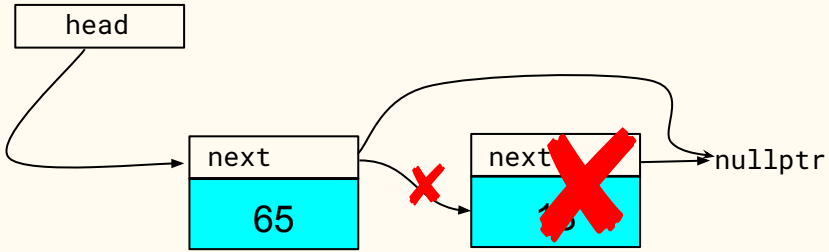
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        ---
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

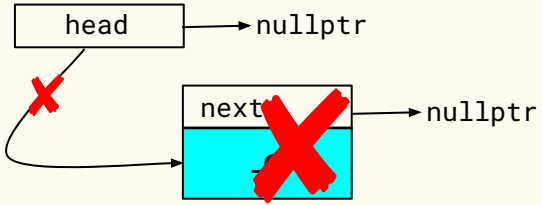


removing the tail Node

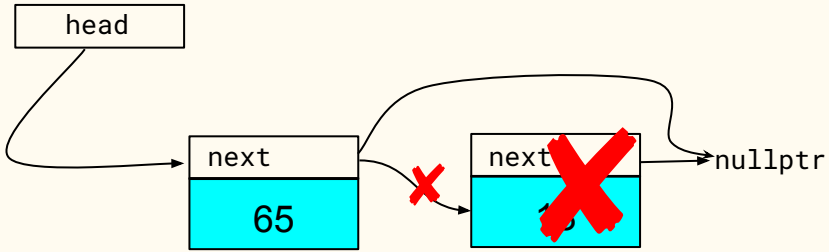
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        _31_
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

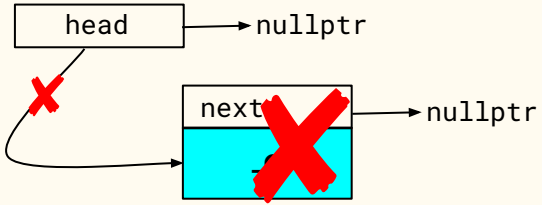


removing the tail Node

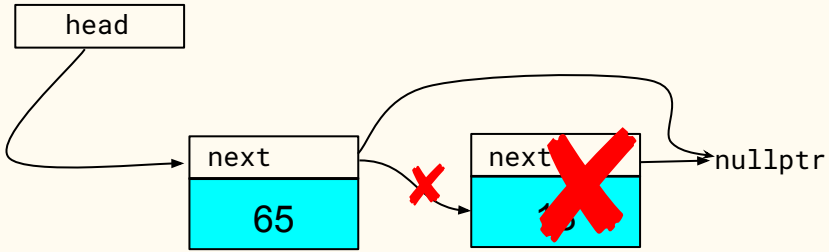
Which statement replaces blank #31 to free the memory allocated for the tail Node?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        _31_
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

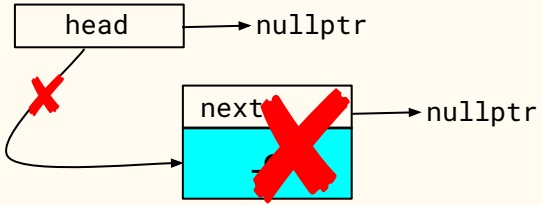


removing the tail Node

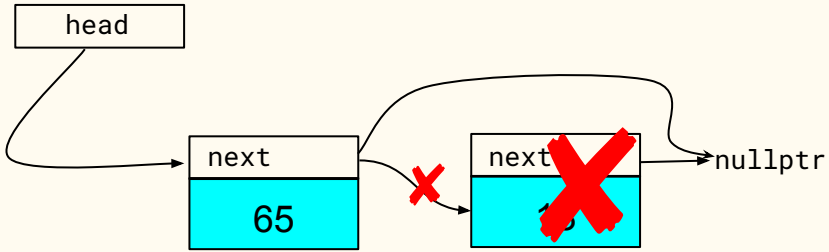
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

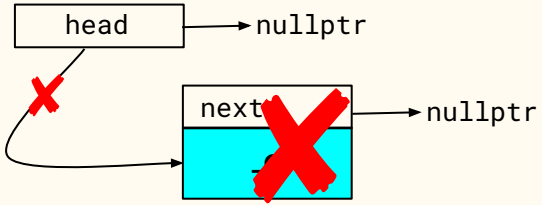


removing the tail Node

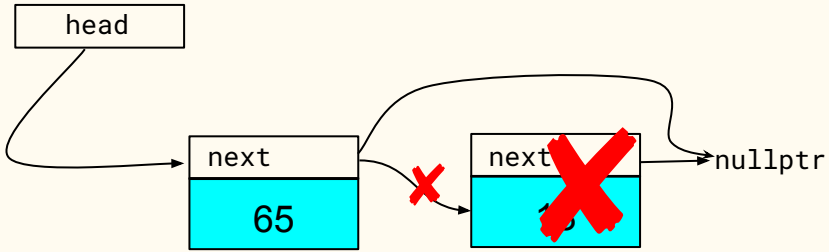
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        ---
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

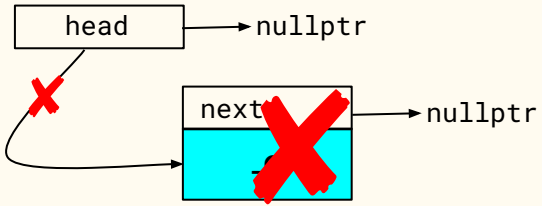


removing the tail Node

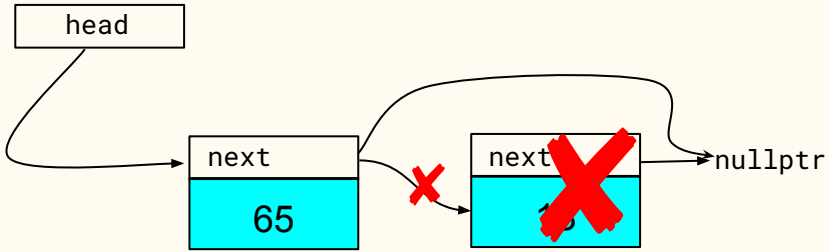
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        _32_
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

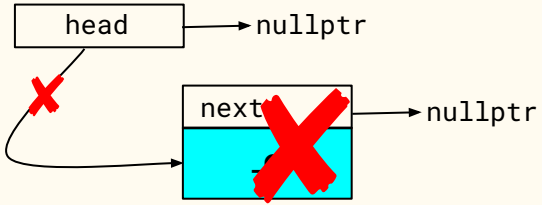


removing the tail Node

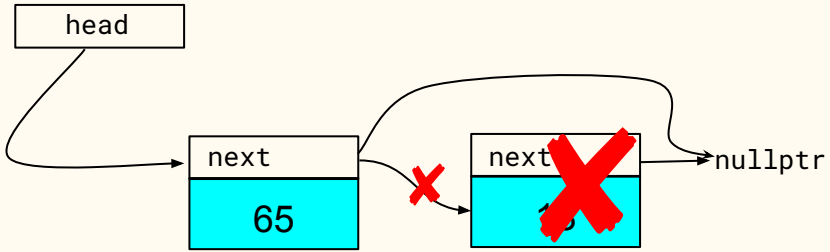
Which statement (replacing blank #32) assigns the appropriate address to `head_ptr` to avoid the creation of a dangling pointer?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        _32_
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

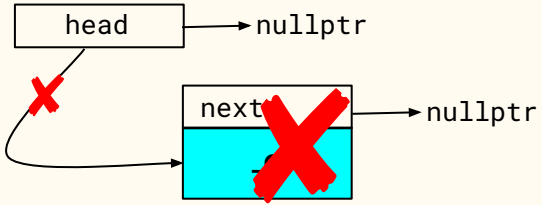


removing the tail Node

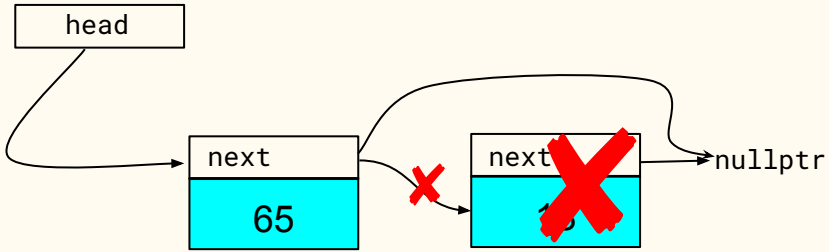
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

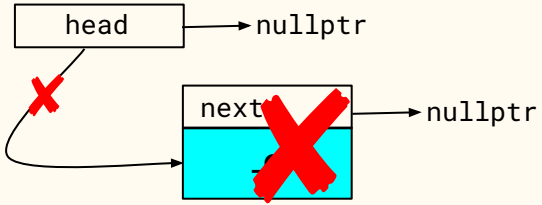


removing the tail Node

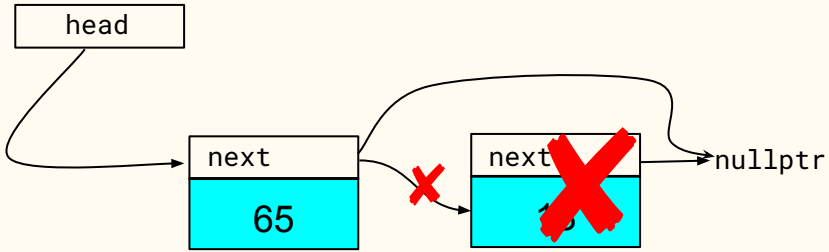
Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        ---
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

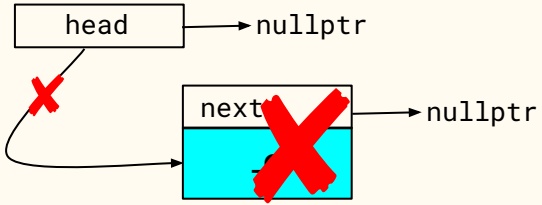


removing the tail Node

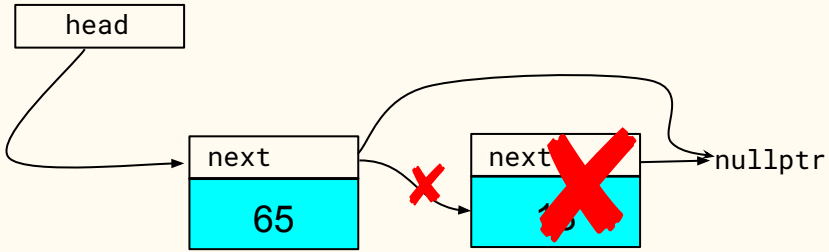
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        _33_
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

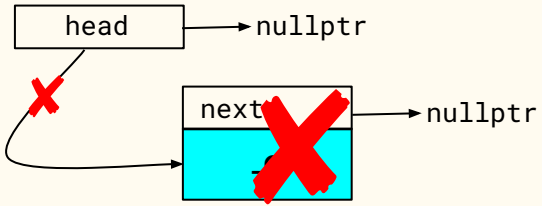


removing the tail Node

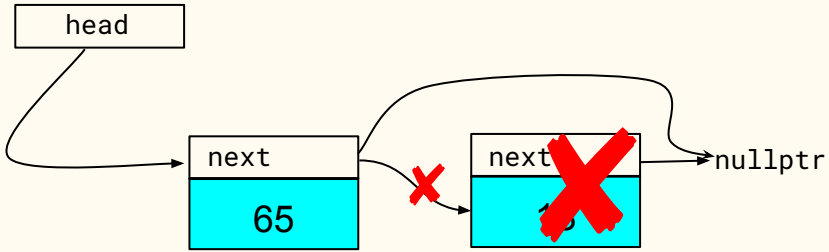
Which statement replaces blank #33 to indicate that the tail pointer has been successfully removed?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        _33_
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

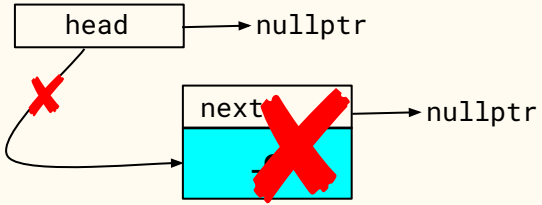


removing the tail Node

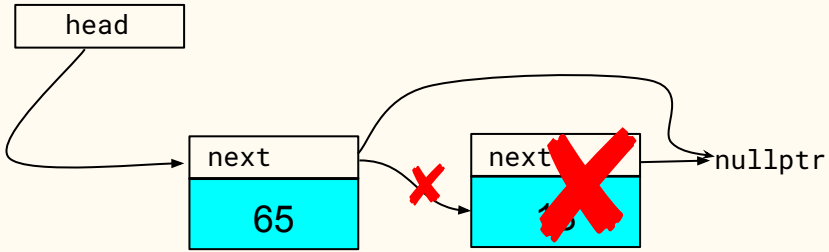
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    // handle case when head Node is tail Node
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

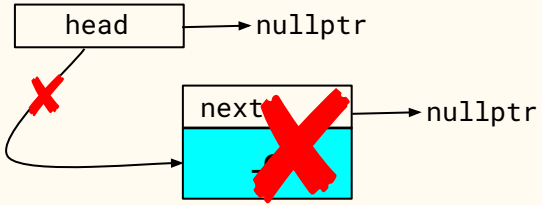


removing the tail Node

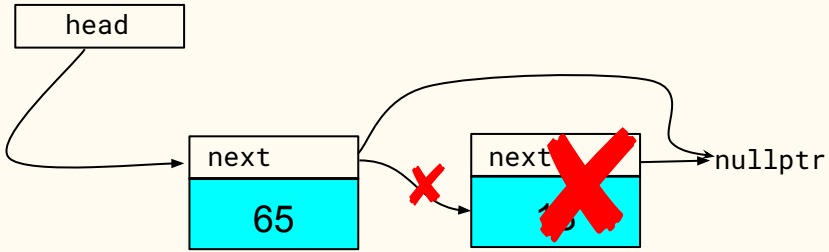
Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



removing the tail Node (1 Node list)

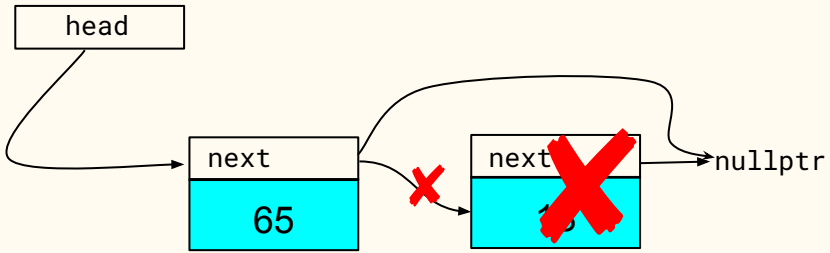


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

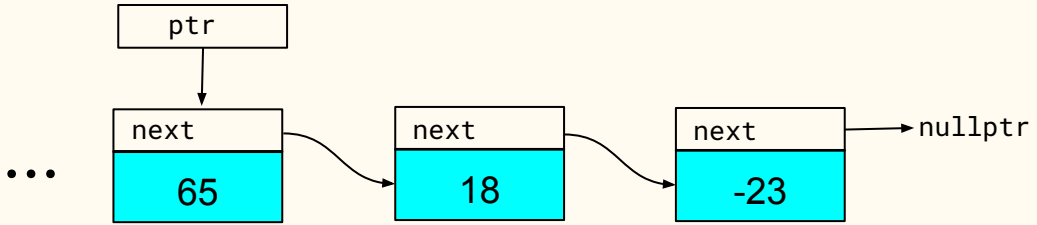


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

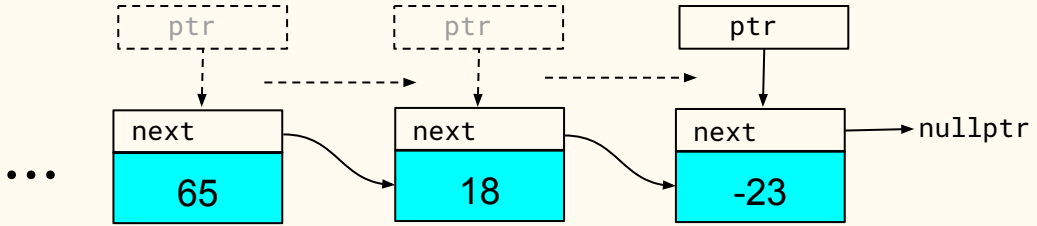


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

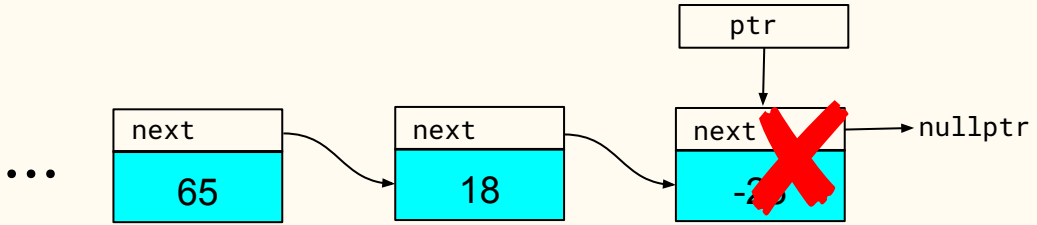


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

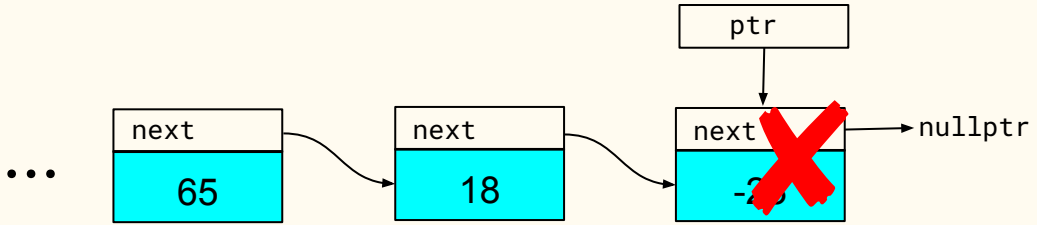


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

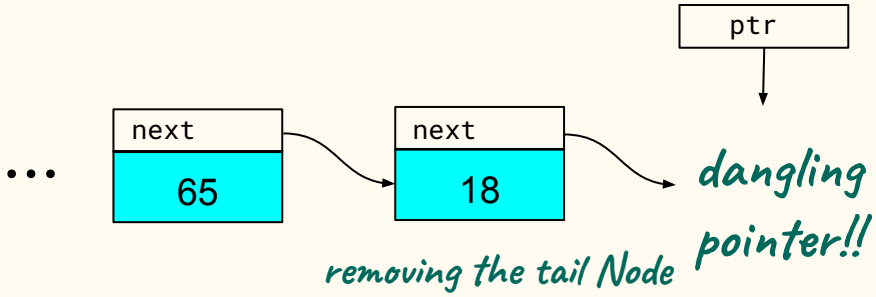


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

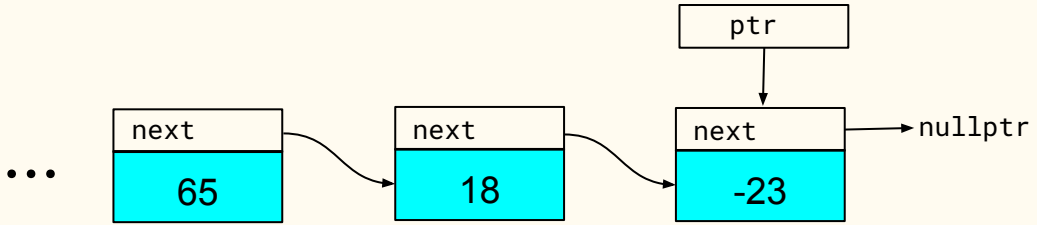
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

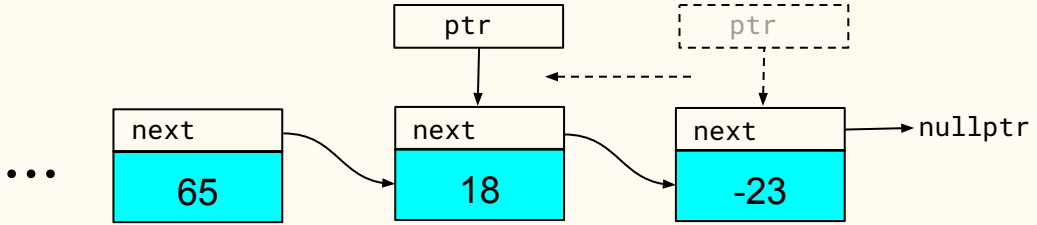


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

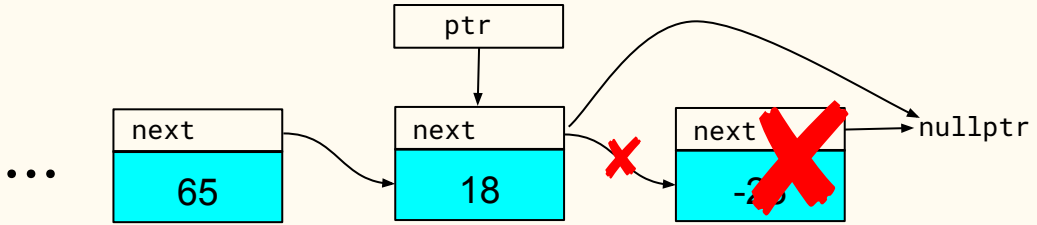


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }

    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

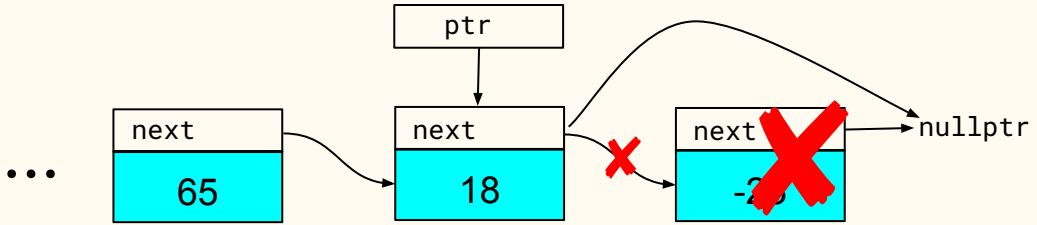


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

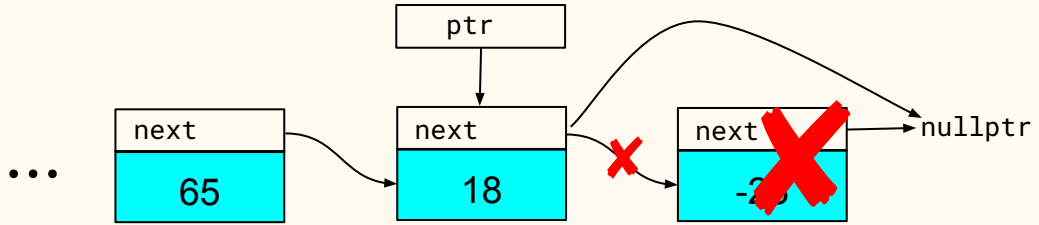


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (___) {
         // traverse list
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

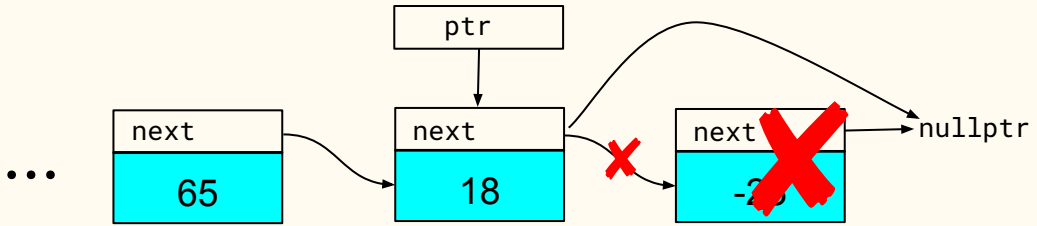


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (___ != nullptr) {
         // traverse list
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

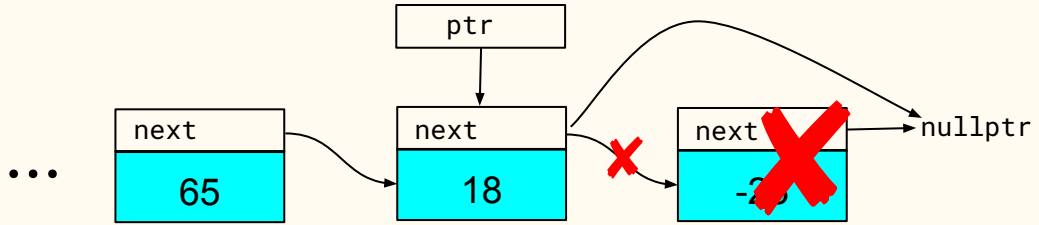


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (_34_ != nullptr) {
         // traverse list
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

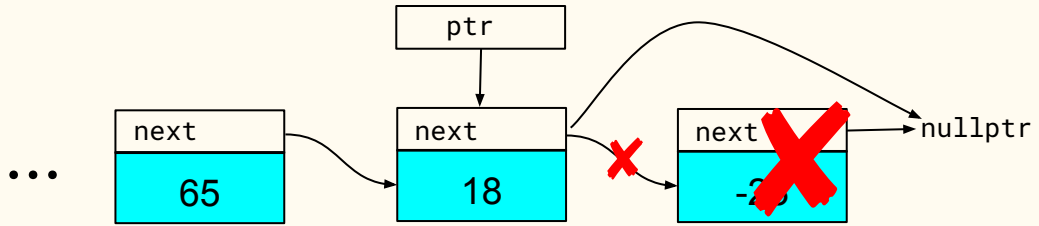


removing the tail Node

Which expression replaces blank #34 to determine when `sec_to_last` reaches the second to last `Node`?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (_34_ != nullptr) {
         // traverse list
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

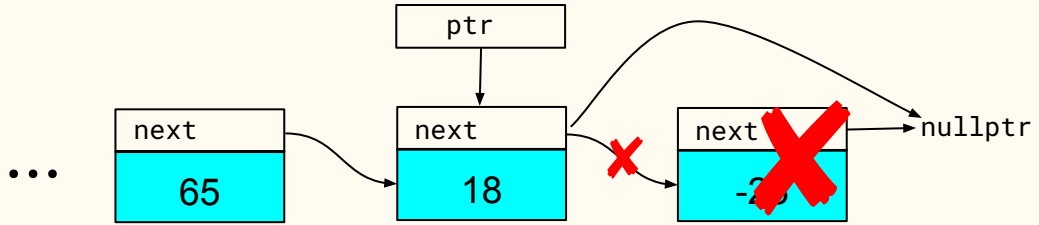


removing the tail Node

Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        // traverse list
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



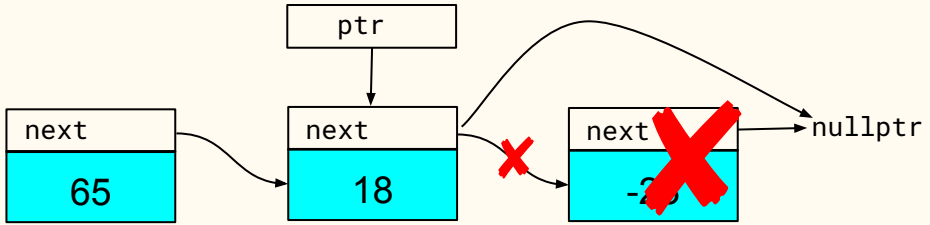
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        // traverse list
        ---
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



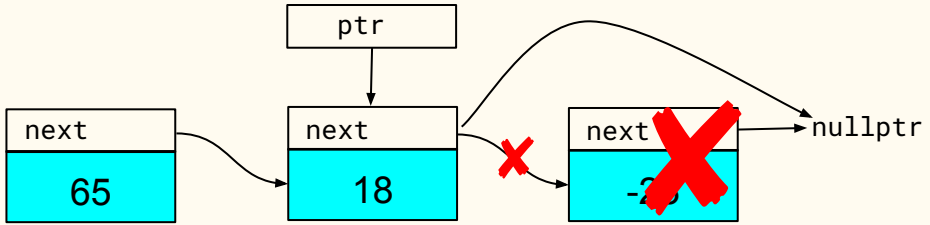
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        // traverse list
        _35_
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



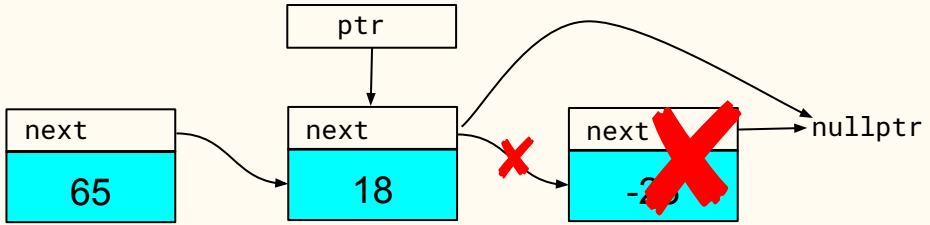
removing the tail Node

Which statement replaces blank #35 to point `sec_to_last` at the next `Node` in the list?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        // traverse list
        _35_
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



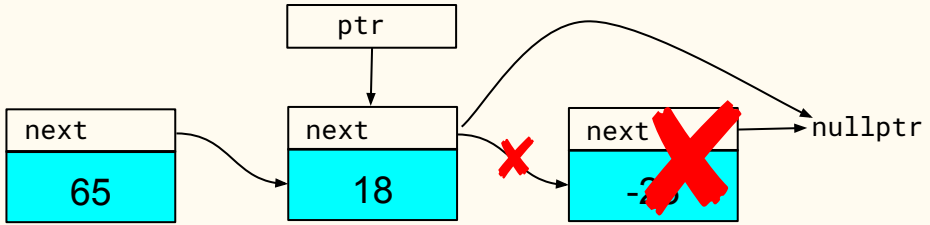
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        // traverse list
        sec_to_last = sec_to_last->next;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...

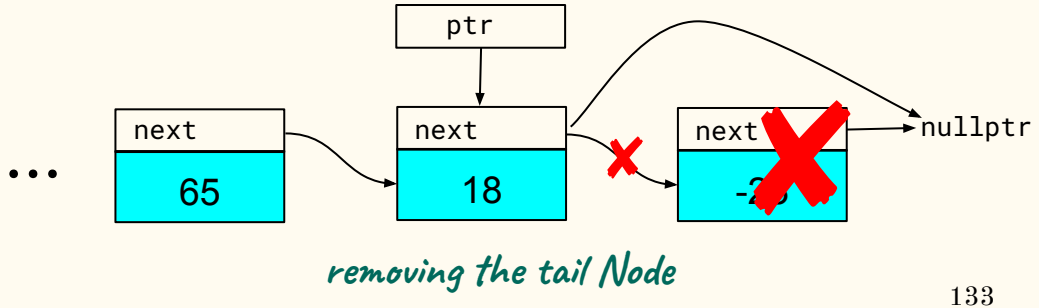


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



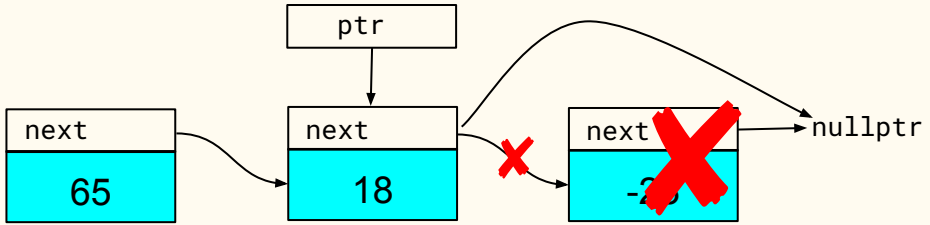
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    ---
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



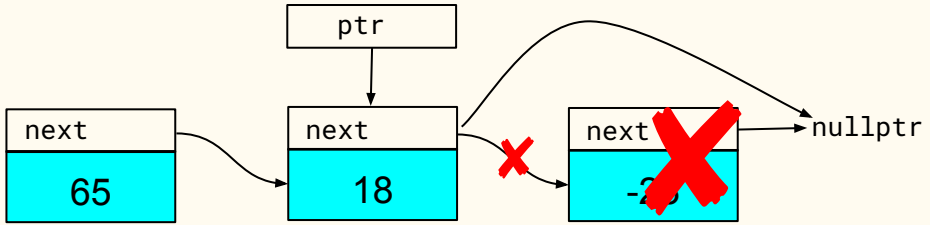
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    _36_
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



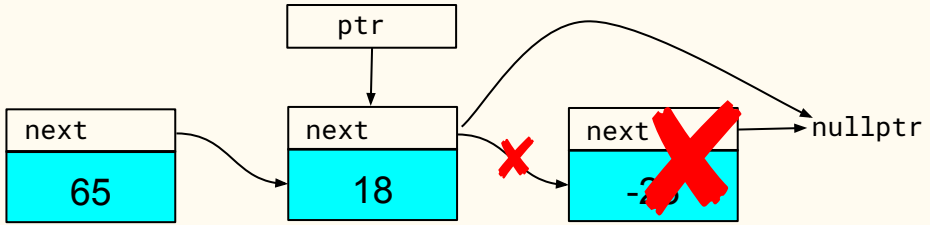
removing the tail Node

Which statement replaces blank #36 to free the memory allocated for the tail `Node` of the list?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    _36_
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



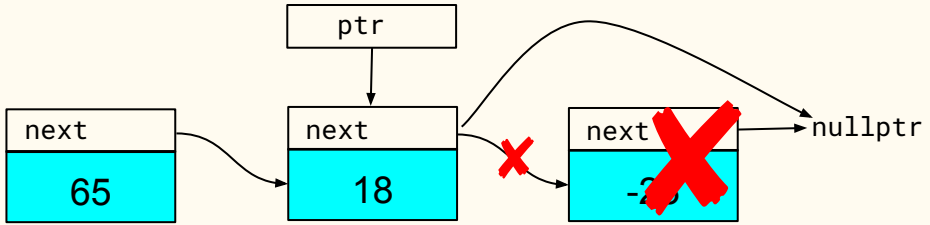
removing the tail Node

Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



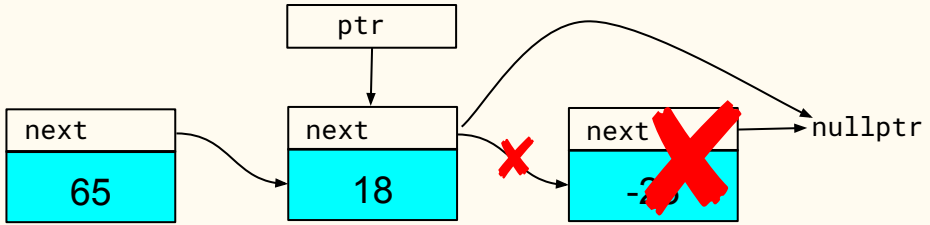
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    ---
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



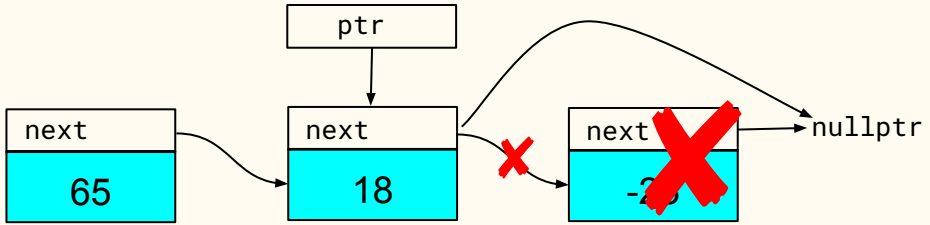
removing the tail Node

Removing a Node from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    _37_
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



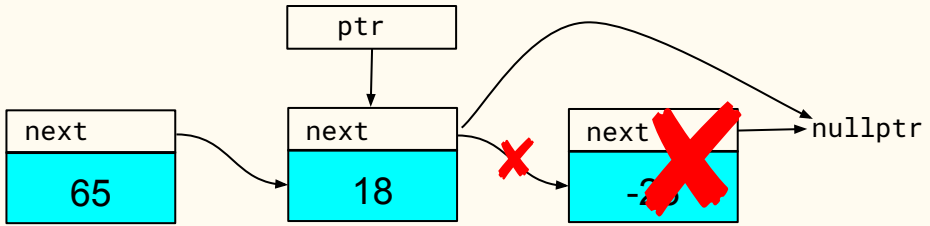
removing the tail Node

Which statement (replacing blank #37) makes the old second to last `Node` the new tail `Node`?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    _37_
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...



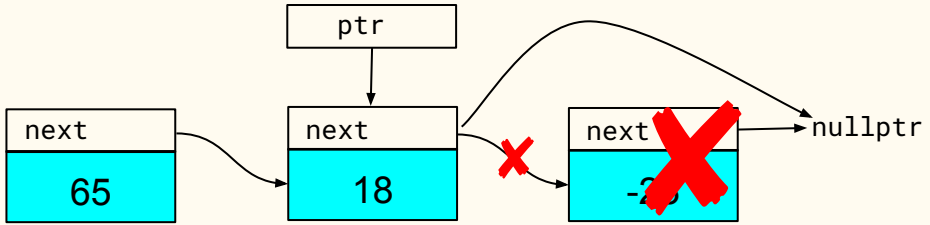
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    sec_to_last->next = nullptr;
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...

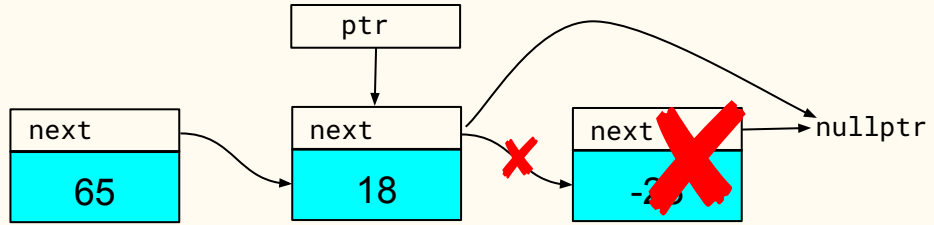


removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    sec_to_last->next = nullptr;
    ---
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



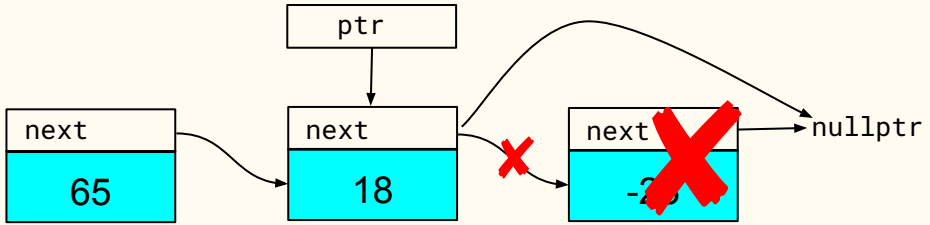
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    sec_to_last->next = nullptr;
    _38_
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...

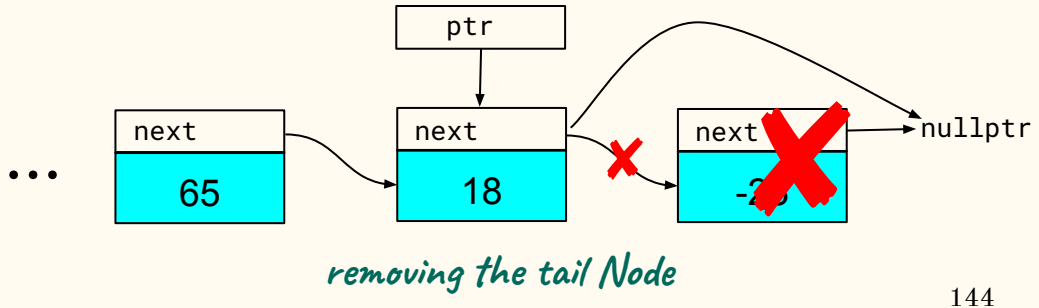


removing the tail Node

Which statement (replacing blank #38) indicates that the tail `Node` was successfully removed?

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    sec_to_last->next = nullptr;
    _38_
}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```



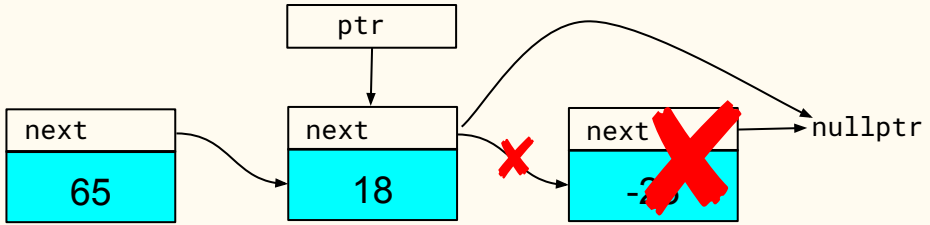
removing the tail Node

Removing a `Node` from list

```
bool remove_tail_from_list(Node*& head_ptr) {
    if (head_ptr == nullptr) {
        return false;
    }
    if (head_ptr->next == nullptr) {
        delete head_ptr;
        head_ptr = nullptr;
        return true;
    }
    // remove tail from list with 2+ Nodes
    Node* sec_to_last = head_ptr;
    while (sec_to_last->next->next != nullptr) {
        sec_to_last = sec_to_last->next;
    }
    delete sec_to_last->next;
    sec_to_last->next = nullptr;
    return true;
}
```

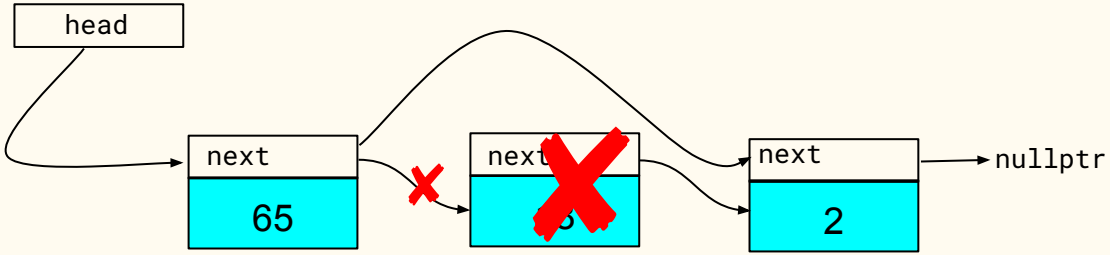
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

...

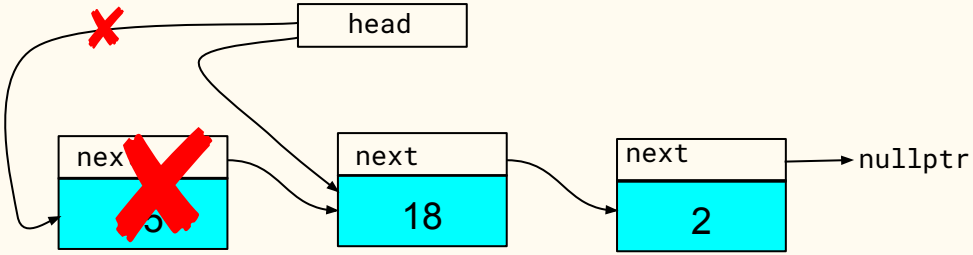


removing the tail Node

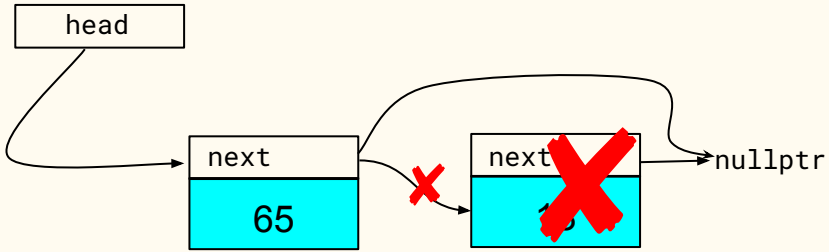
Removing a `Node` from list



removing interior Node

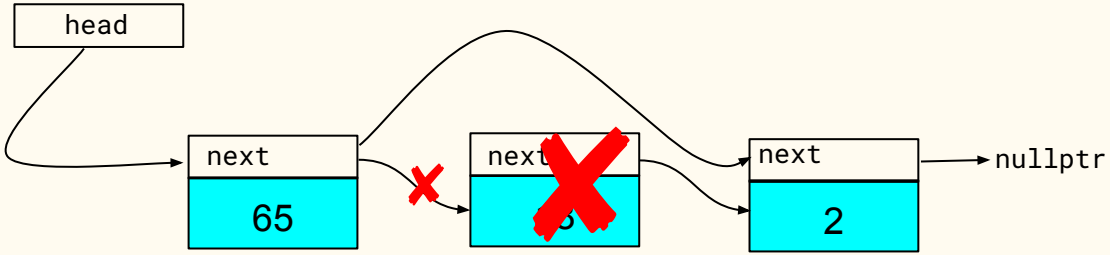


removing the head Node



removing the tail Node

Removing a `Node` from list

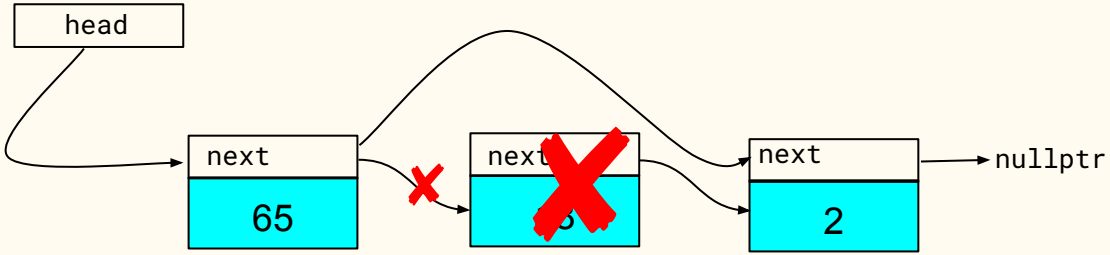


removing interior Node

```
bool remove_node_from_list() {}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

Removing a `Node` from list

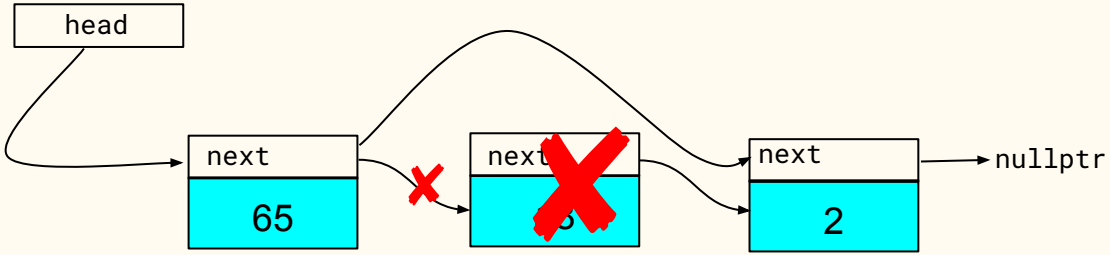


removing interior Node

```
bool remove_node_from_list(___ ___) {}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

Removing a `Node` from list

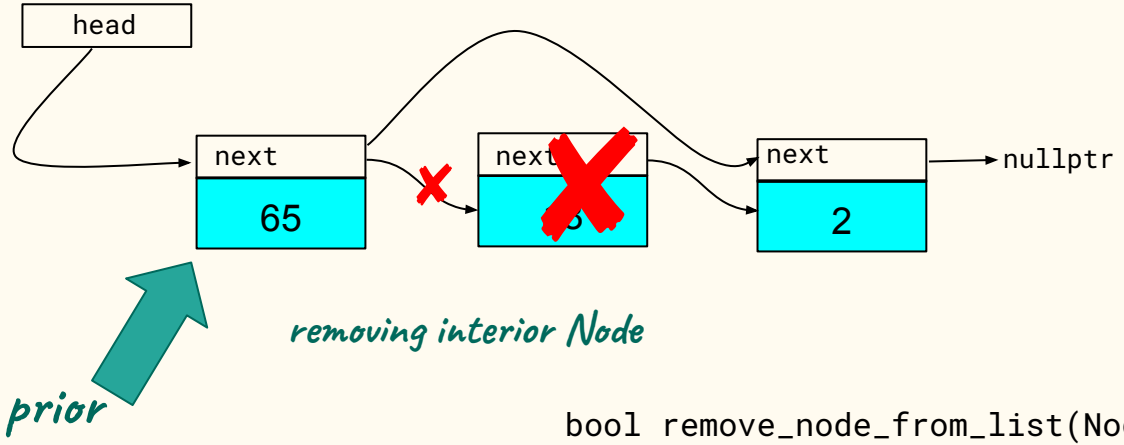


removing interior Node

```
bool remove_node_from_list(Node* ___) {}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

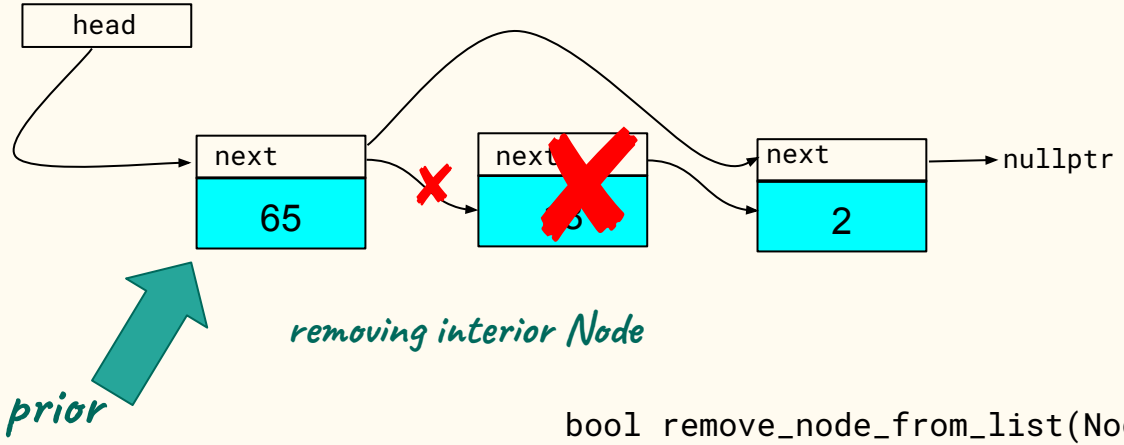
Removing a `Node` from list




```
bool remove_node_from_list(Node* prior) {}
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

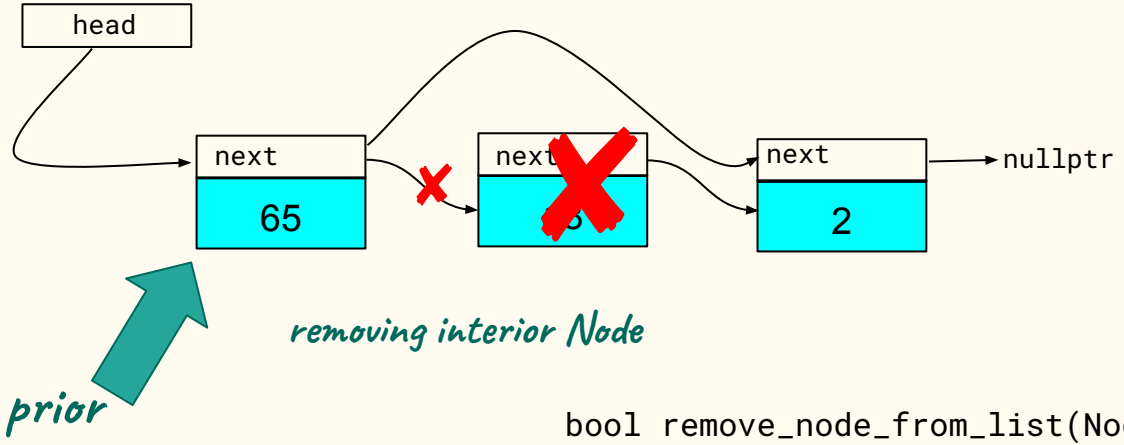
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (___) {

    }
}
```

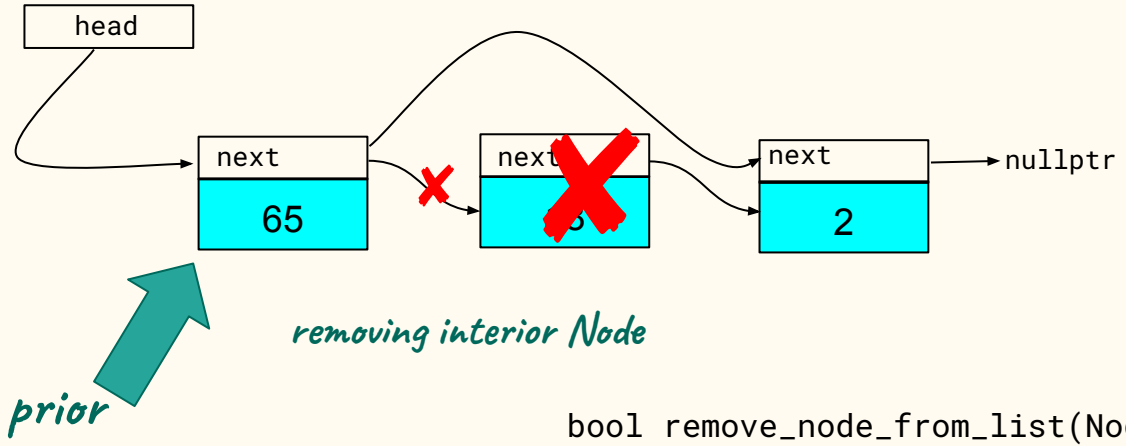
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (_39_) {

    }
}
```

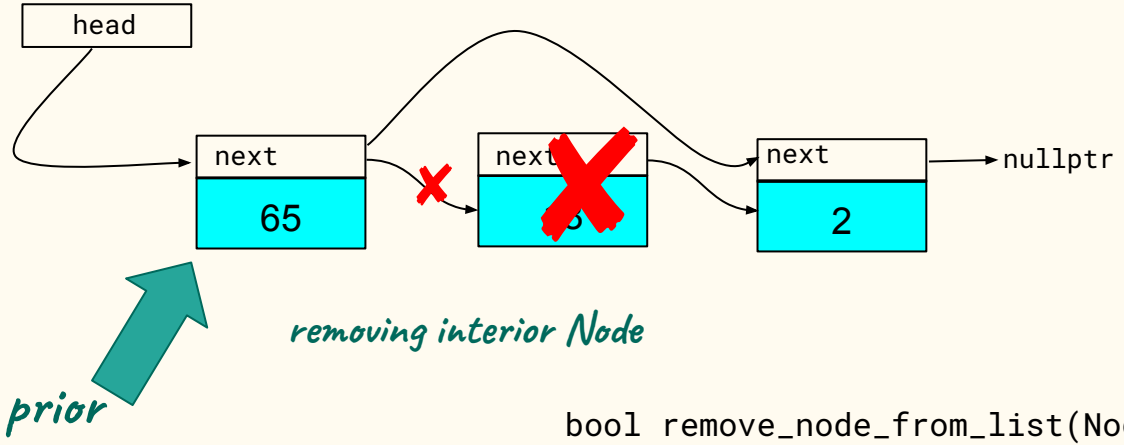
Which boolean expression replaces blank #39 so that removal only occurs when `prior` is not passed `nullptr`?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (_39_) {

    }
}
```

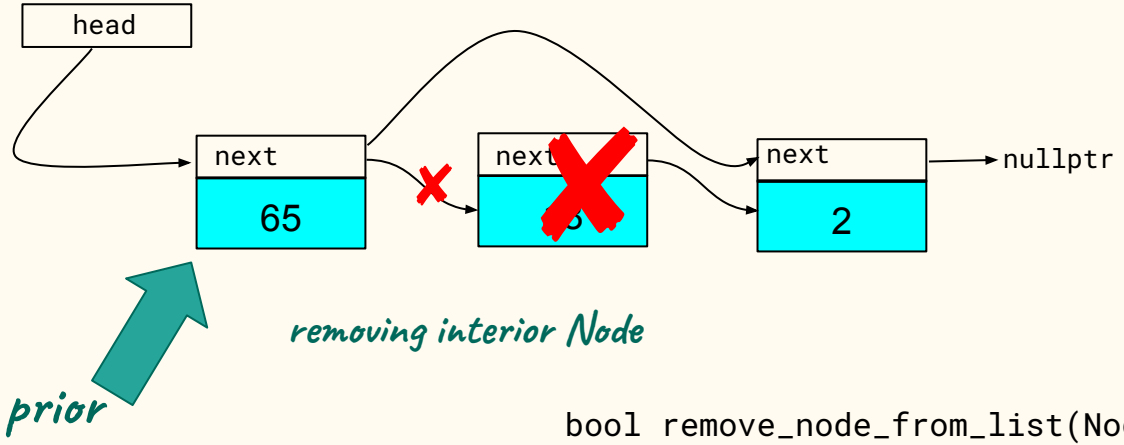
Removing a Node from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {

    }
}
```

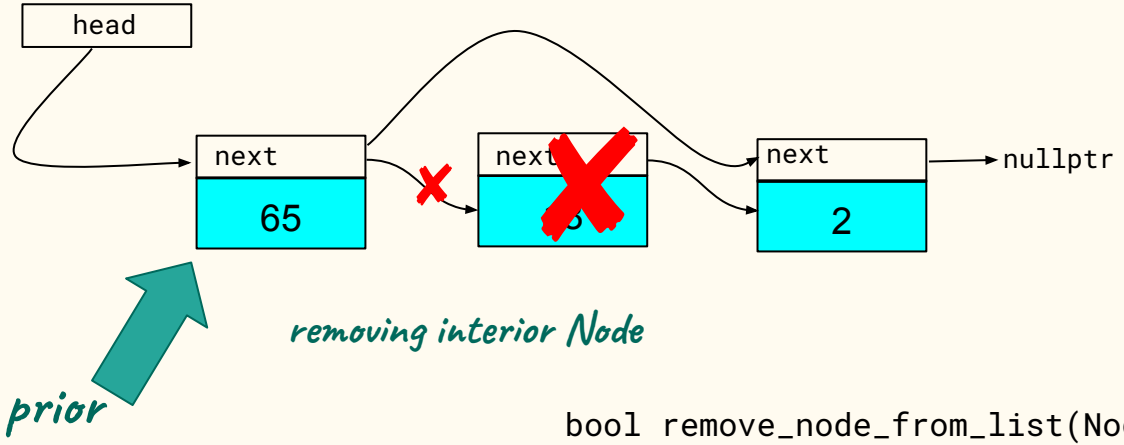
Removing a Node from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        ___

    }
}
```

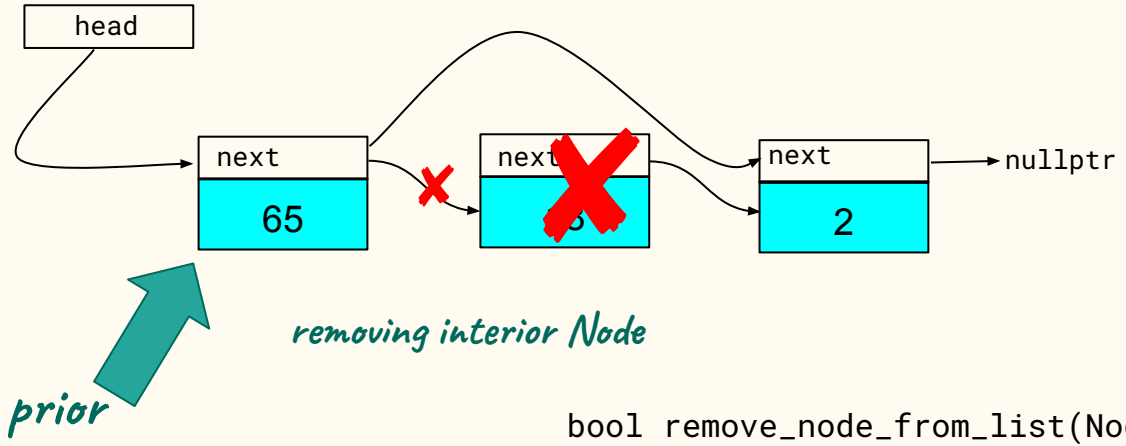
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = _40_;

    }
}
```

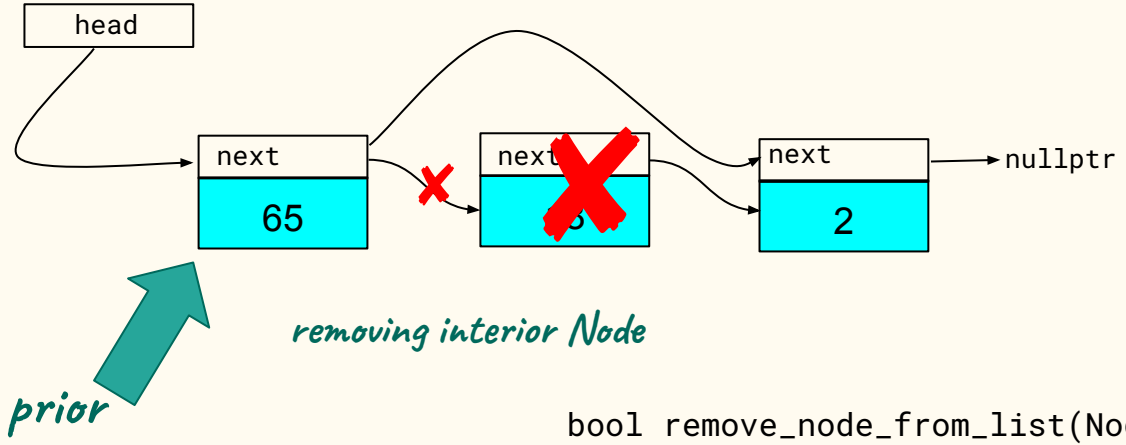
Which expression replaces blank #40 to assign the address of the `Node` to remove `victim`?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = _40_;

    }
}
```

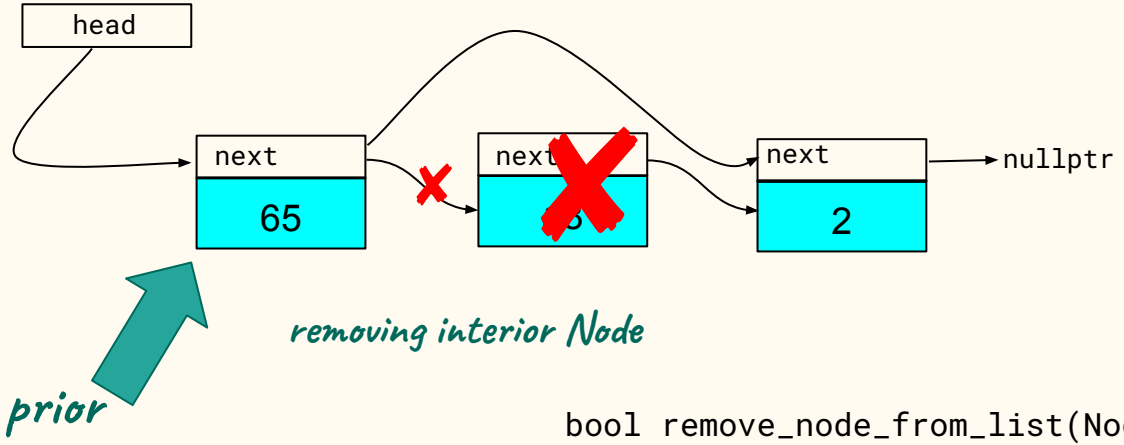
Removing a Node from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

    }
}
```

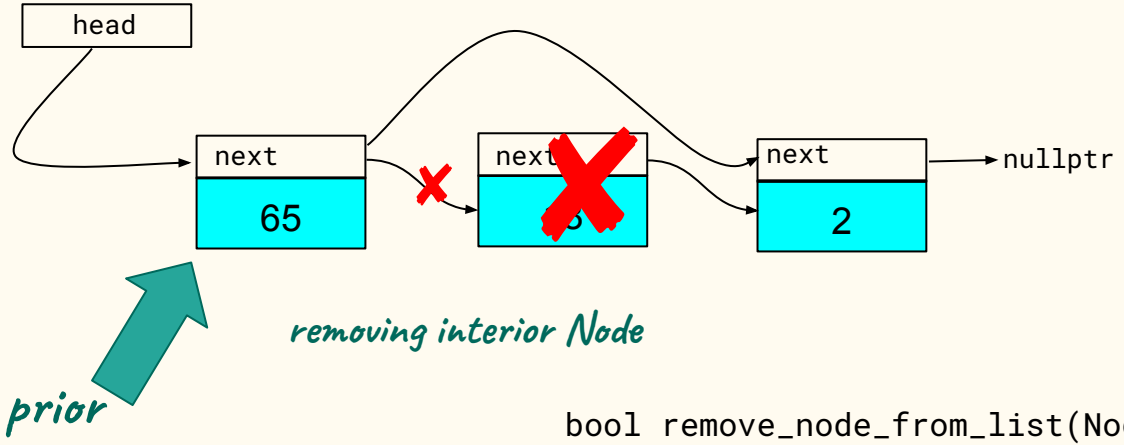
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        ___

    }
}
```

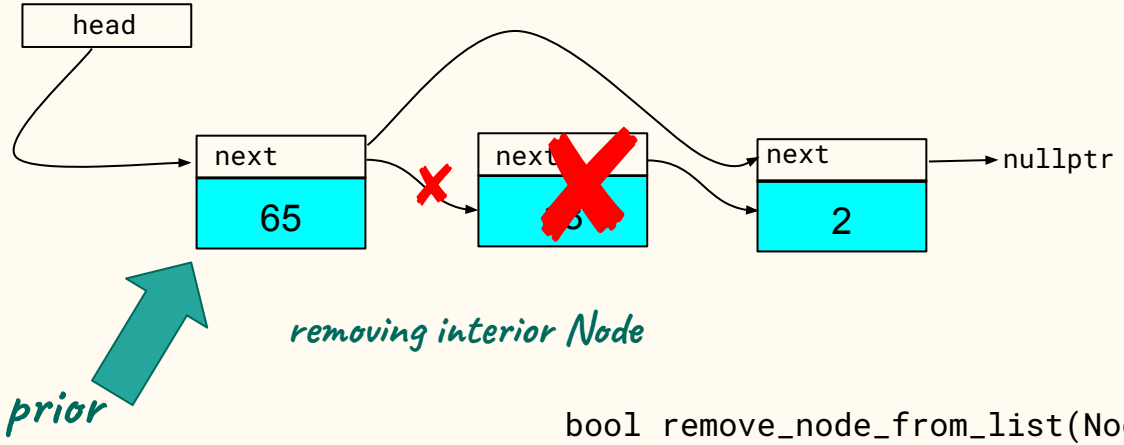
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = ___;

    }
}
```

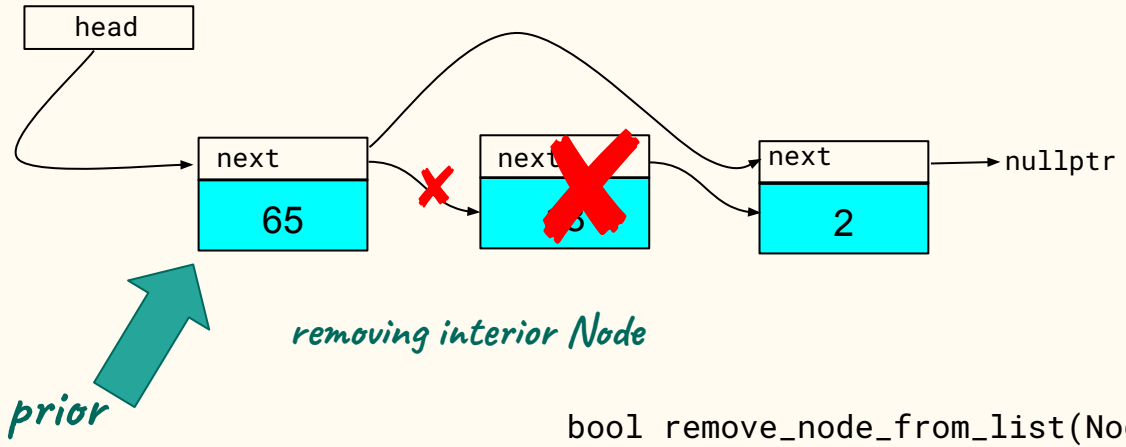
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = _41_;

    }
}
```

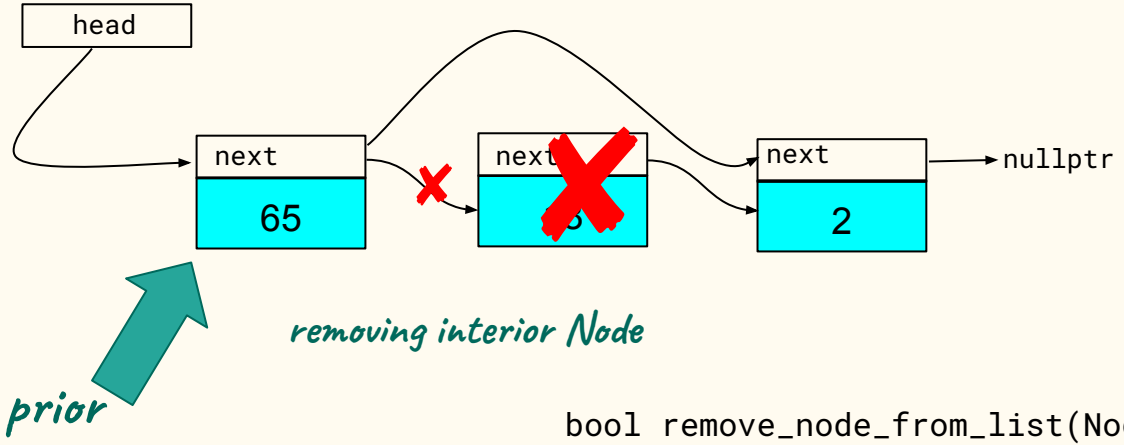
Which expression replaces blank #41 to assign the address of the `Node` following the `Node` being removed to the `next` pointer of `prior`?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = _41_;

    }
}
```

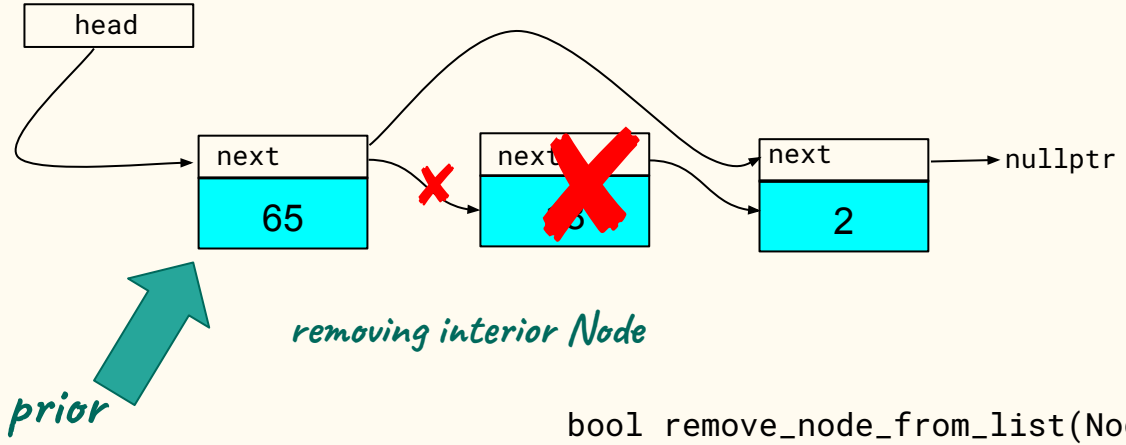
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

    }
}
```

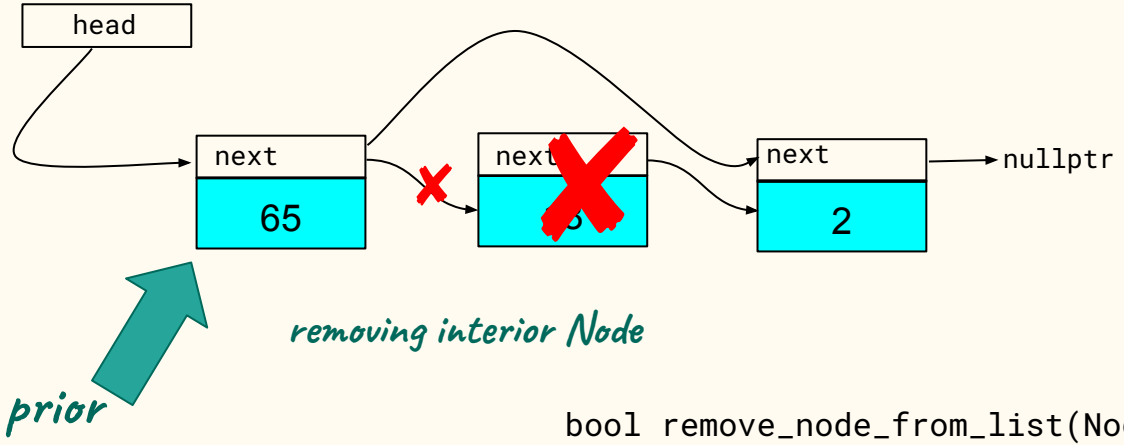
Removing a Node from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        ___
    }
}
```

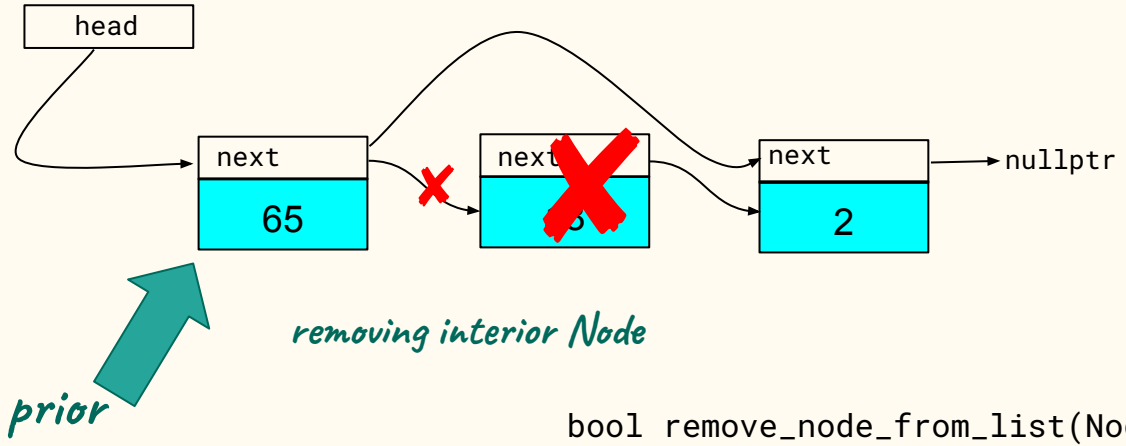
Removing a Node from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        _42_
    }
}
```

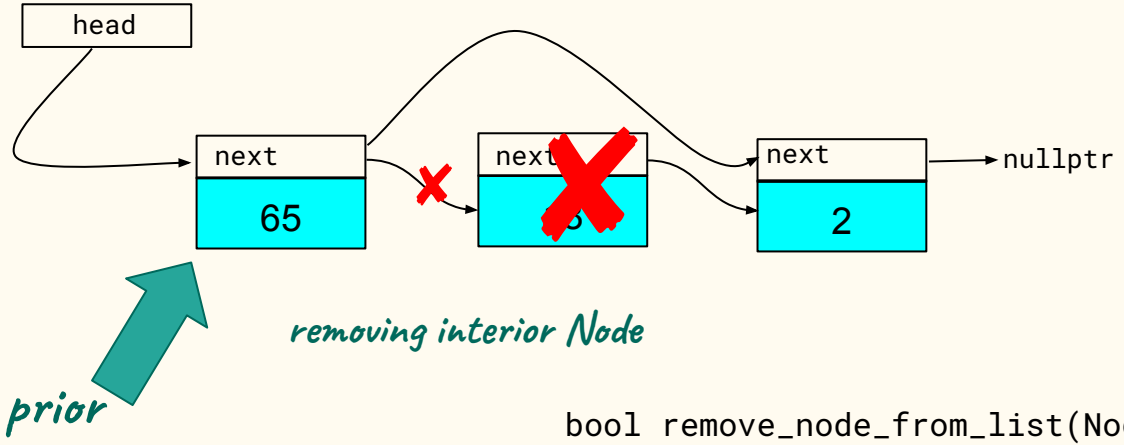
Which expression replaces blank #42 to free the memory for the `Node` being removed?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        _42_
    }
}
```

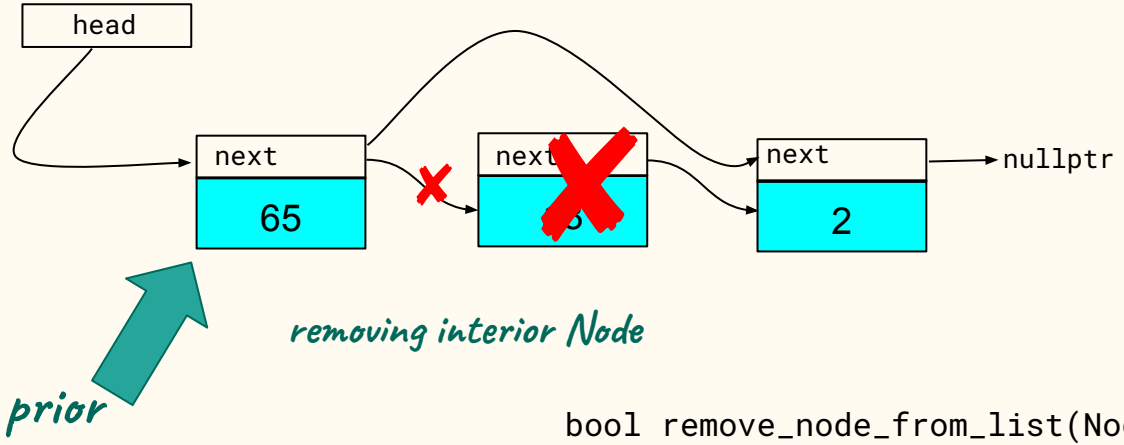
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        delete victim;
    }
}
```

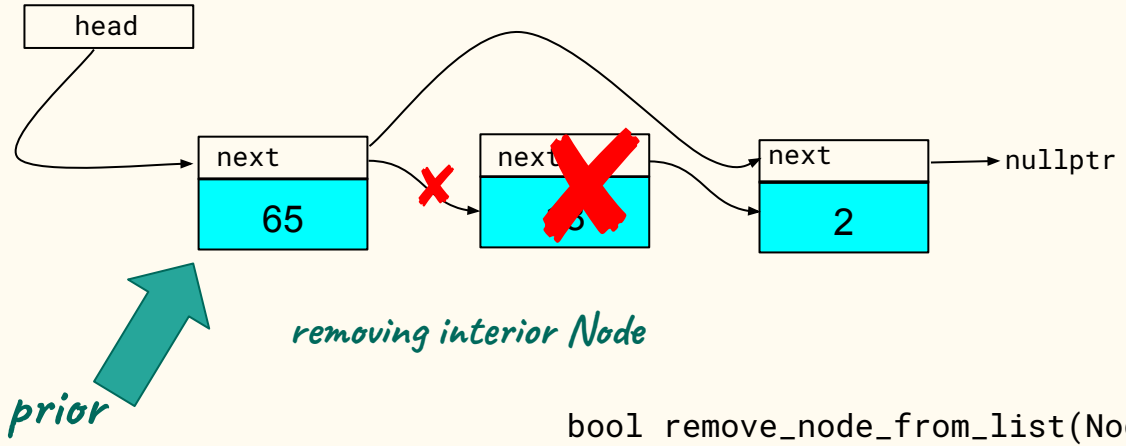
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        delete victim;
        ---
    }
}
```

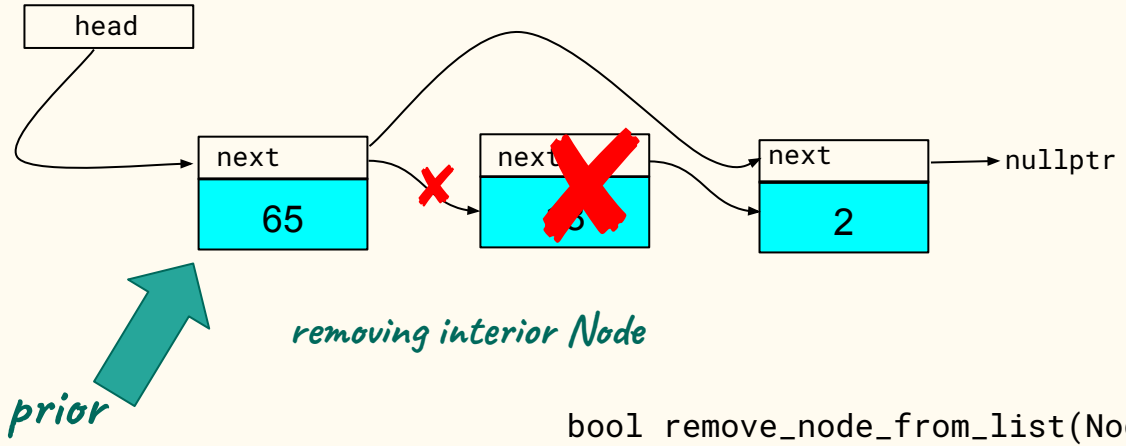
Removing a `Node` from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        delete victim;
        _43_
    }
}
```

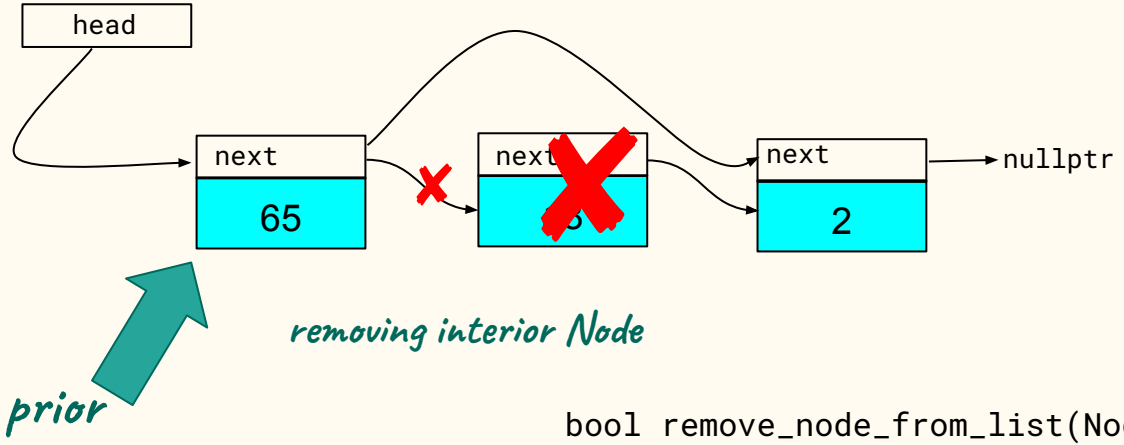
Which expression replaces blank #43 to indicate that the `Node` was successfully removed from the list?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        delete victim;
        _43_
    }
}
```

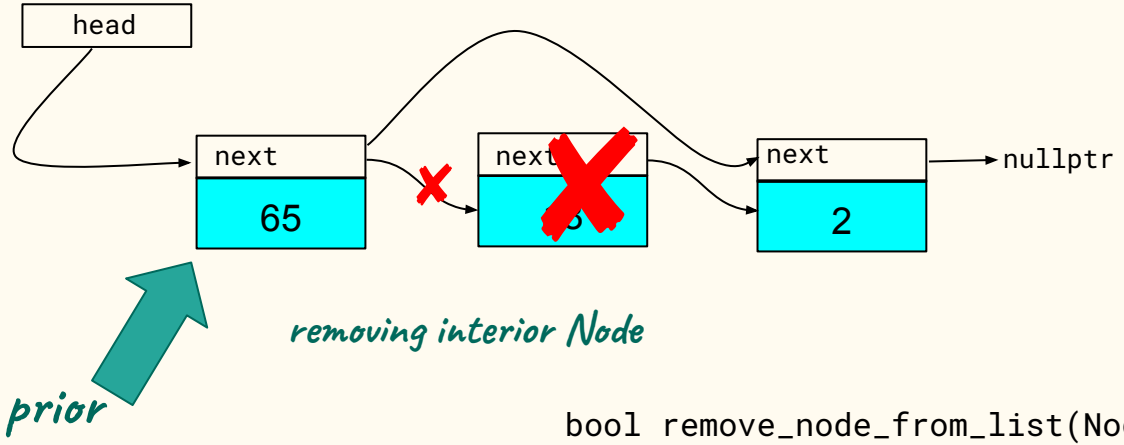
Removing a Node from list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        delete victim;
        return true;
    }
}
```

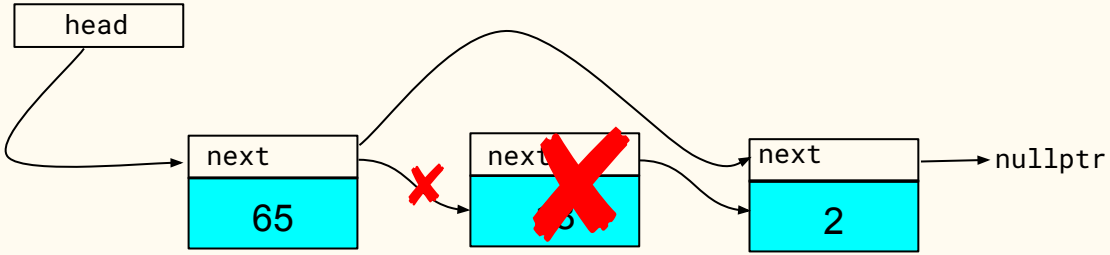
Removing a `Node` from list



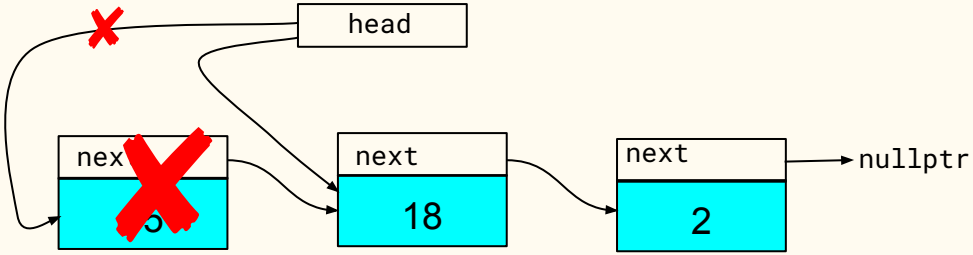

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
bool remove_node_from_list(Node* prior) {
    if (prior != nullptr) {
        Node* victim = prior->next;

        prior->next = prior->next->next;

        delete victim;
        return true;
    }
    return false;
}
```

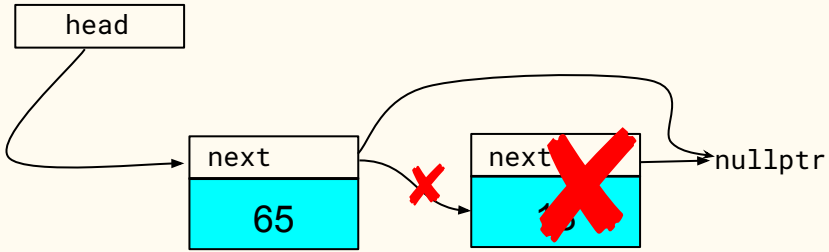
Removing a `Node` from list



removing interior Node

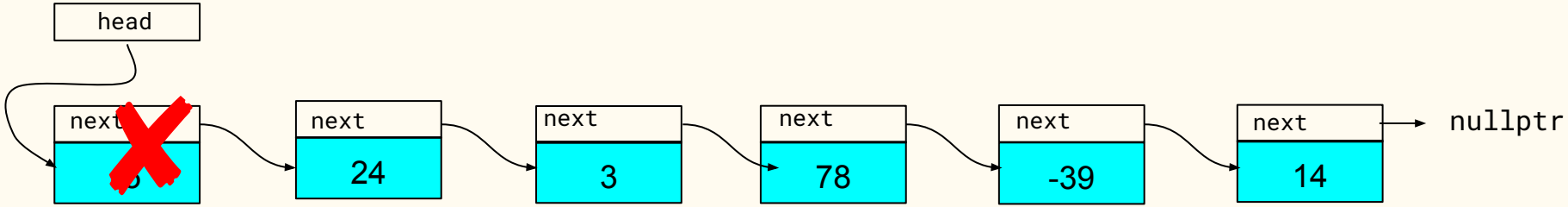


removing the head Node

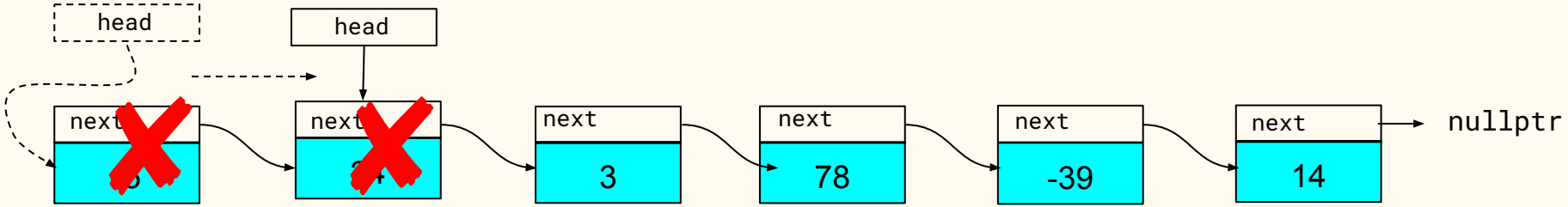


removing the tail Node

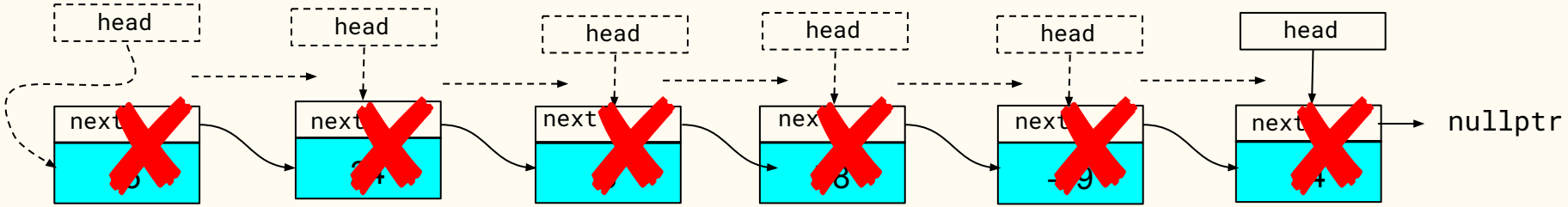
Removing all Nodes from a list



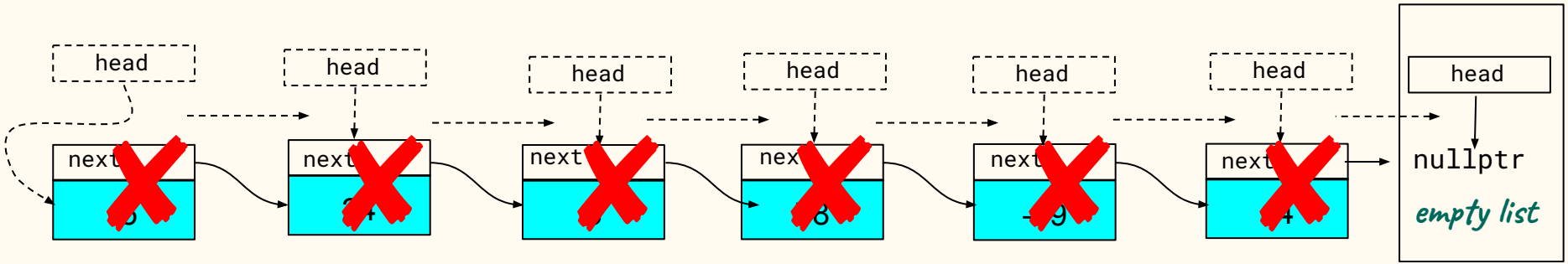
Removing all Nodes from a list



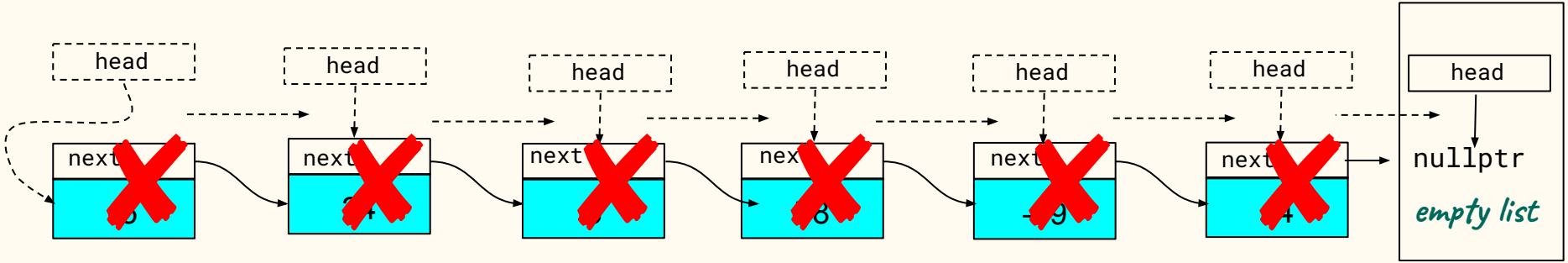
Removing all Nodes from a list



Removing all Nodes from a list



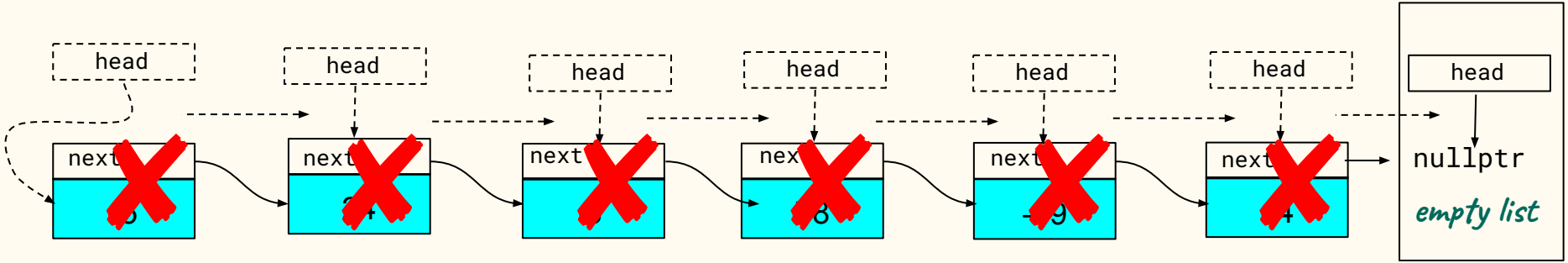
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list() { }
```

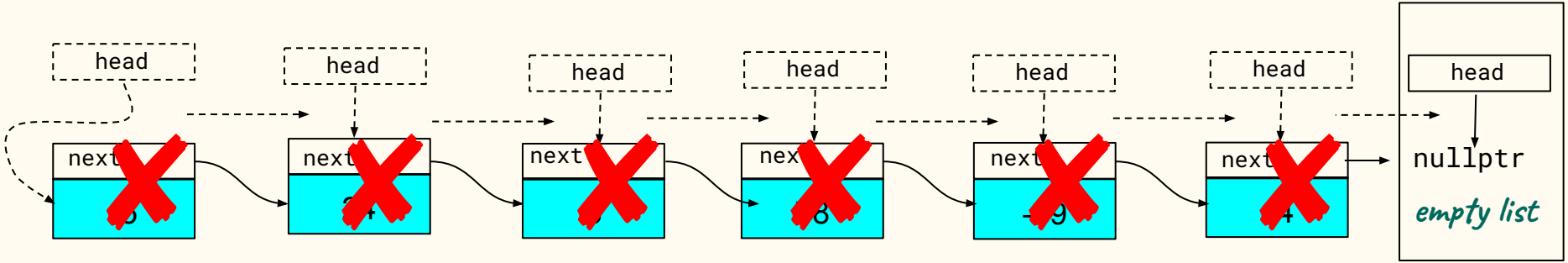
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(___ ___) { }
```

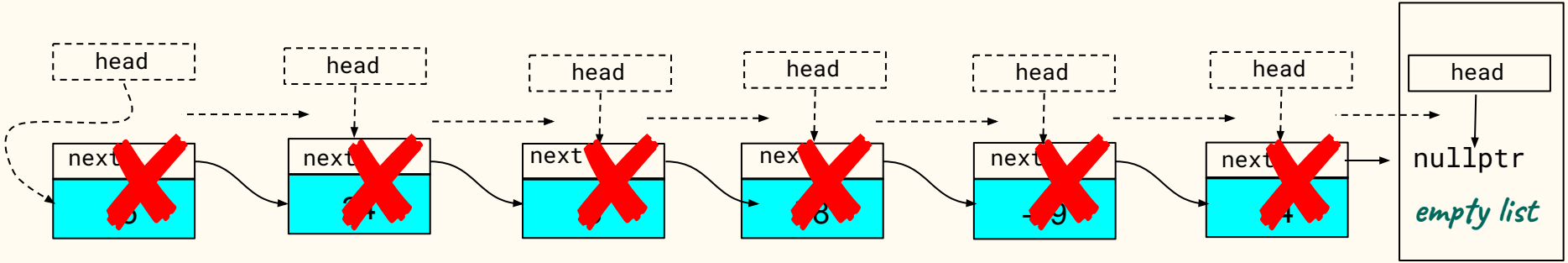
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(___ head_ptr) { }
```

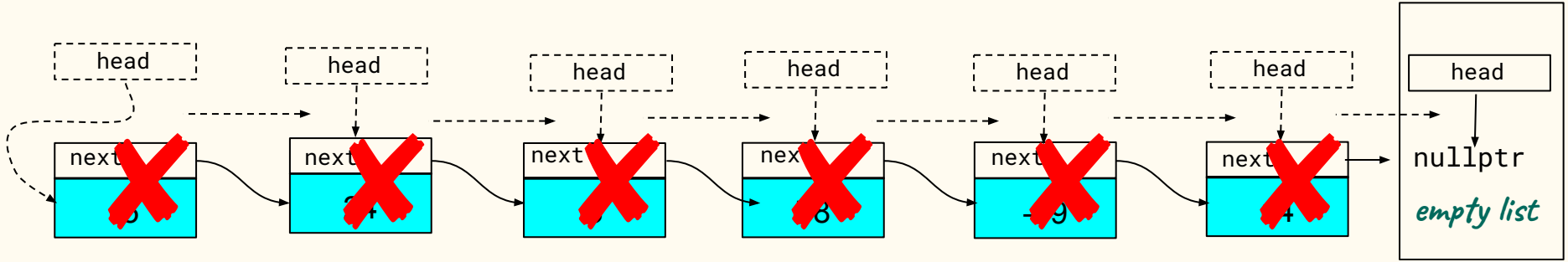
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(_45_ head_ptr) { }
```

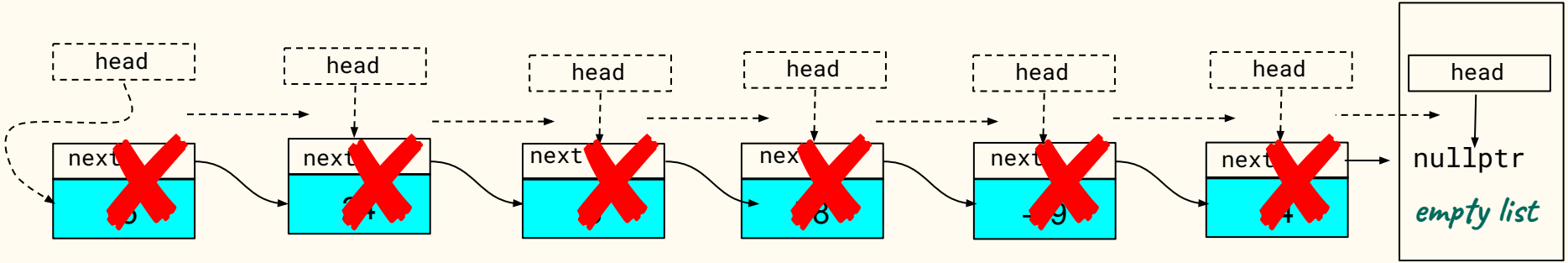
Which type replaces blank #45 for the `head_ptr` parameter declaration?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(_45_ head_ptr) { }
```

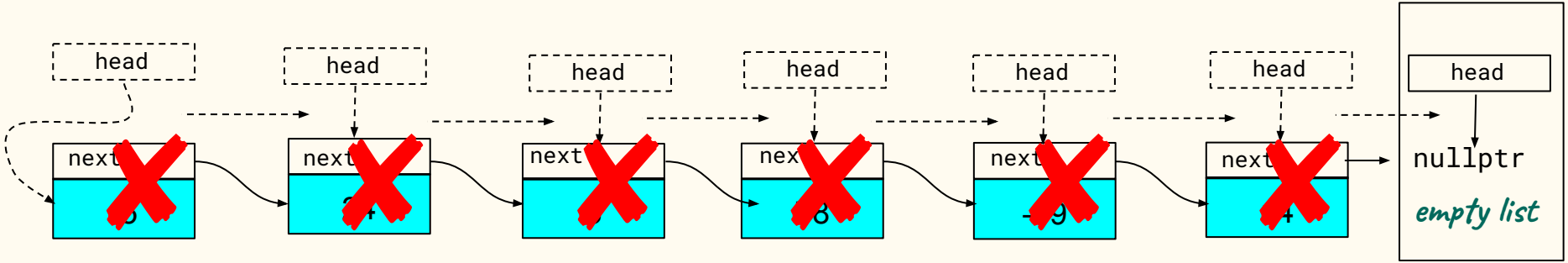
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(___) {

    }
}
```

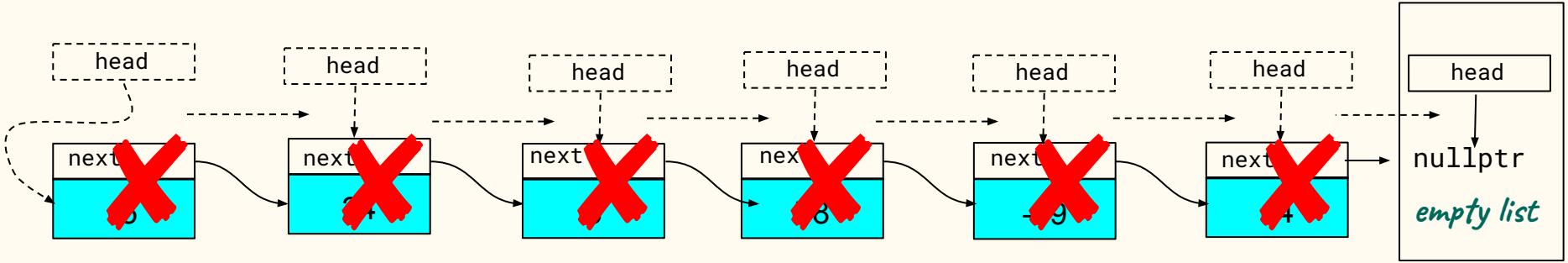
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(_46_) {

    }
}
```

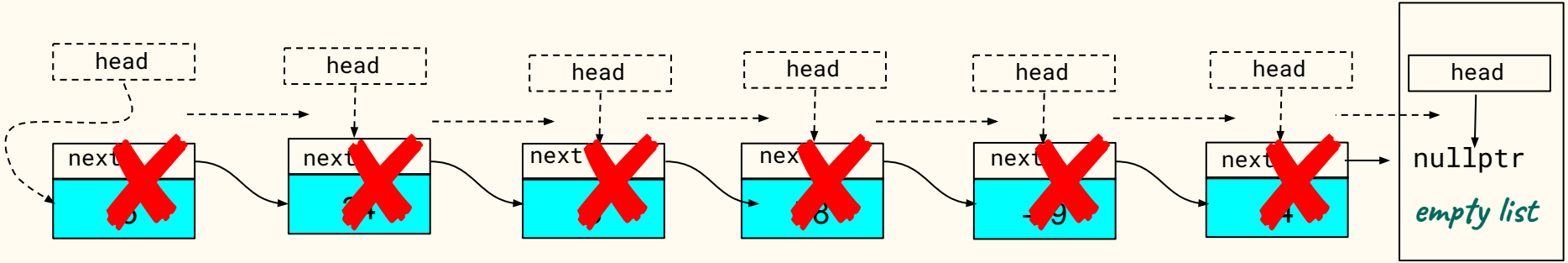
Which boolean expression replacing blank #46 will allow `head_ptr` to traverse the list as long as the list is not empty?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(_46_) {

    }
}
```

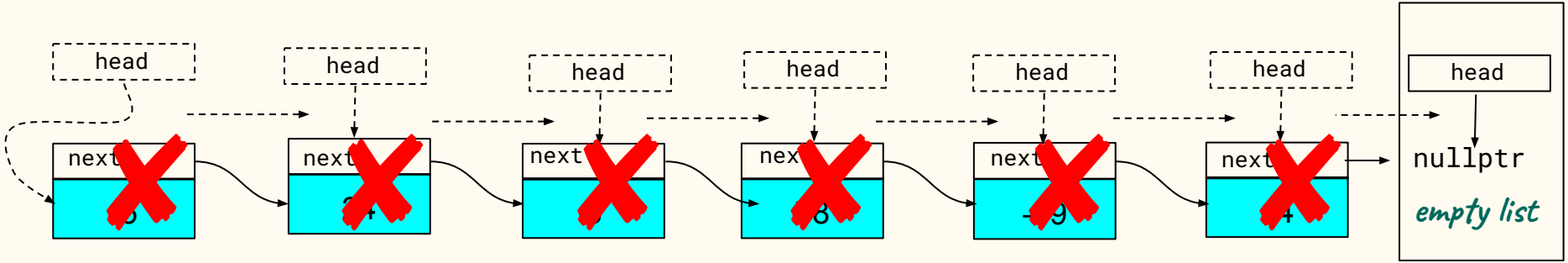
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        ---
    }
}
```

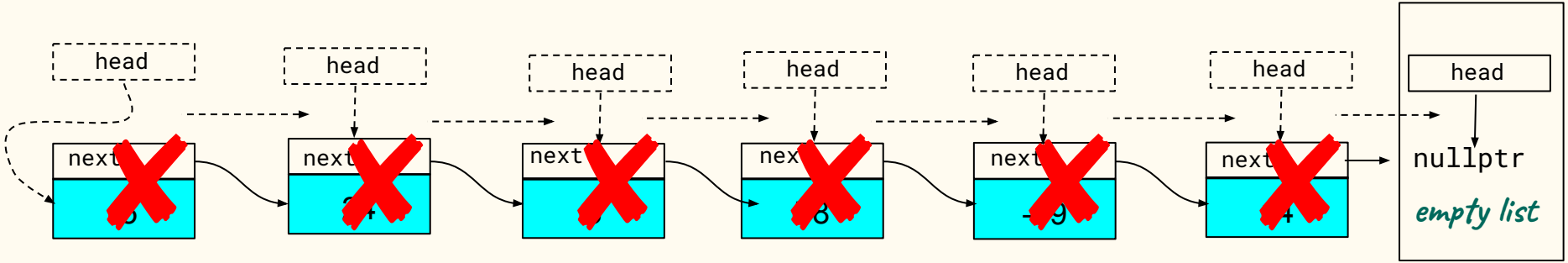
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        ---
    }
}
```

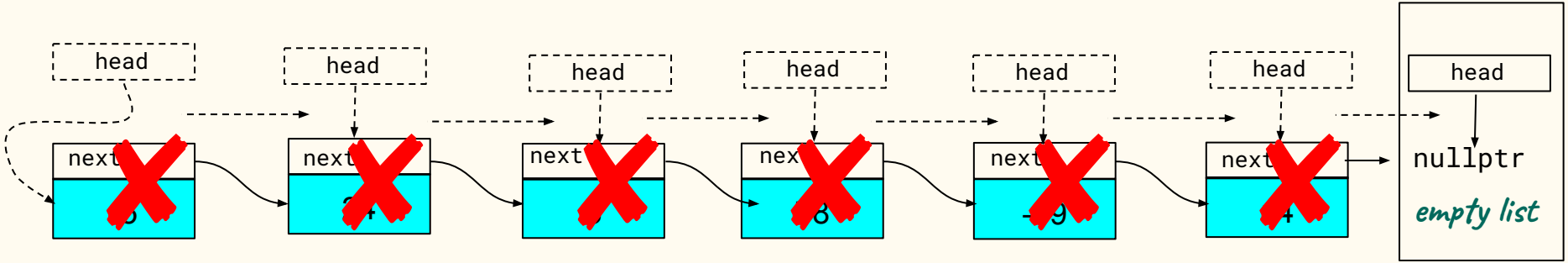
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = ___;
    }
}
```

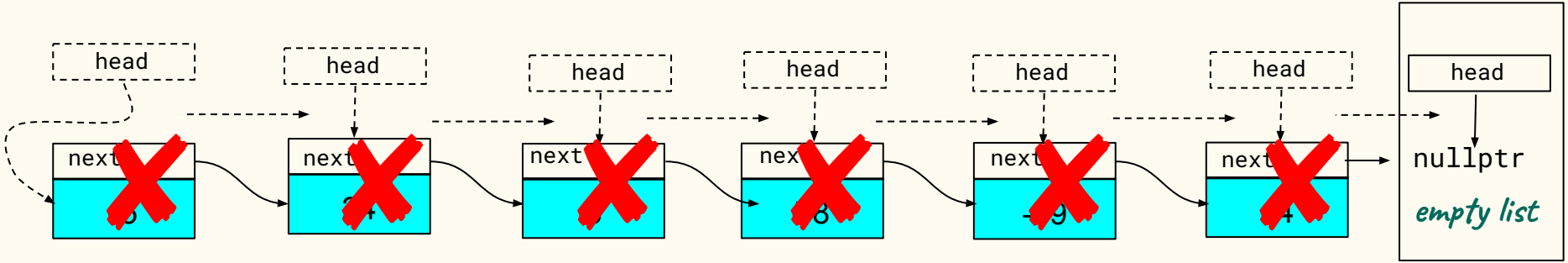
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = _47_;
    }
}
```

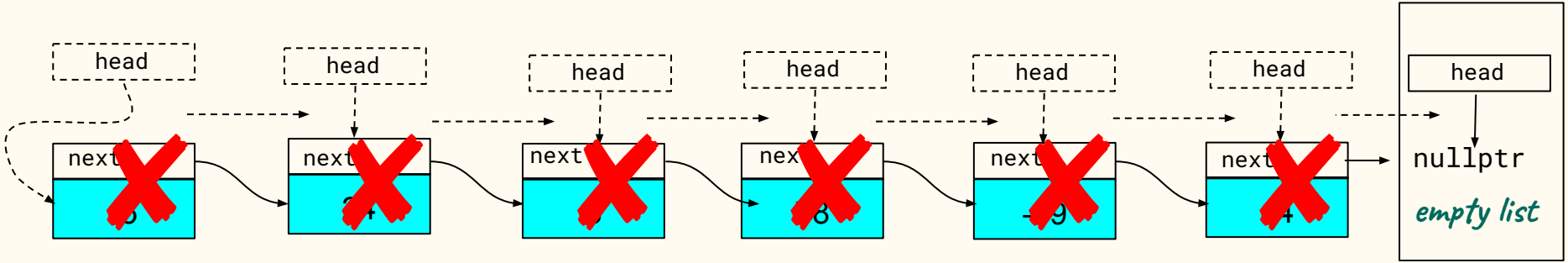
Which expression replaces blank #47 to assign the address of the next `Node` in the list to `next`?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = _47_;
    }
}
```

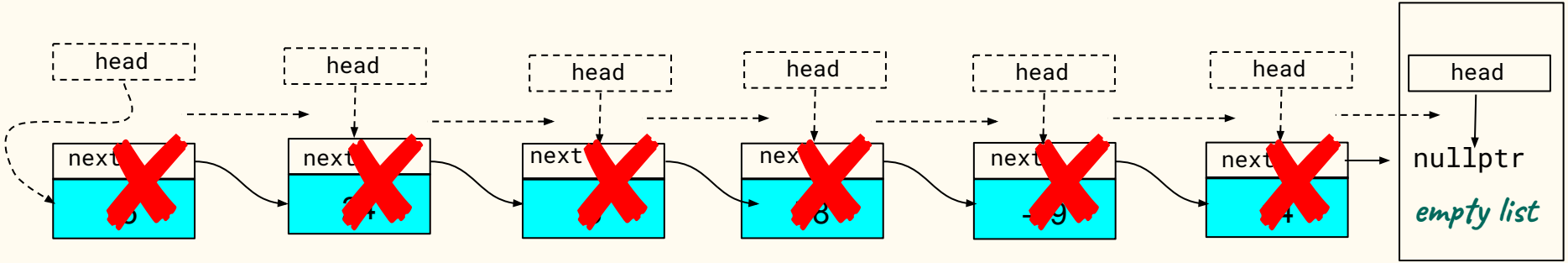
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
    }
}
```

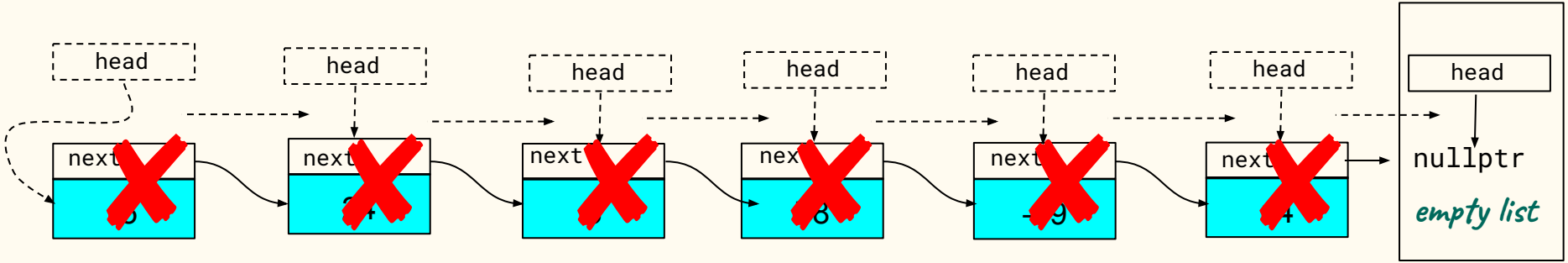
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        ---
    }
}
```

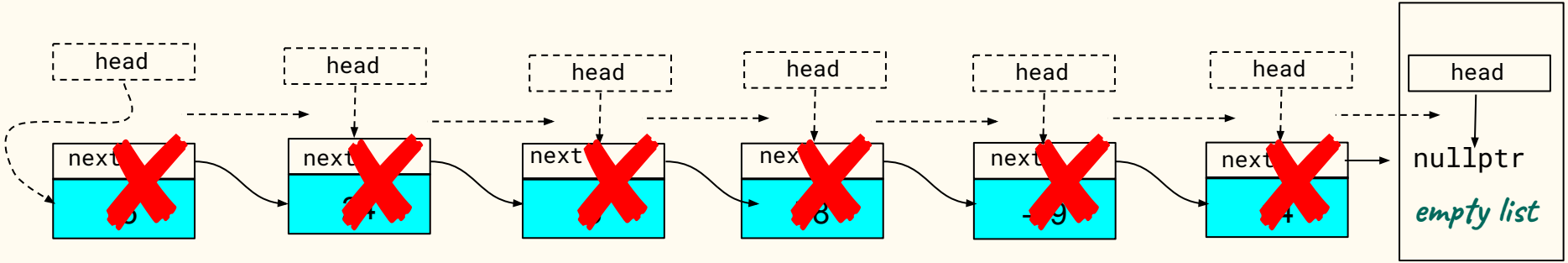
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        _48_
    }
}
```

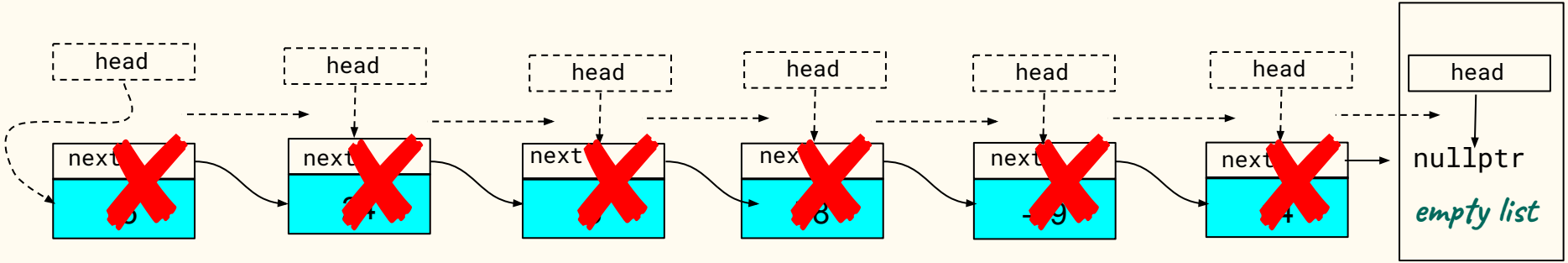
Which statement replaces blank #48 to free the memory of the current head node?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        _48_
    }
}
```

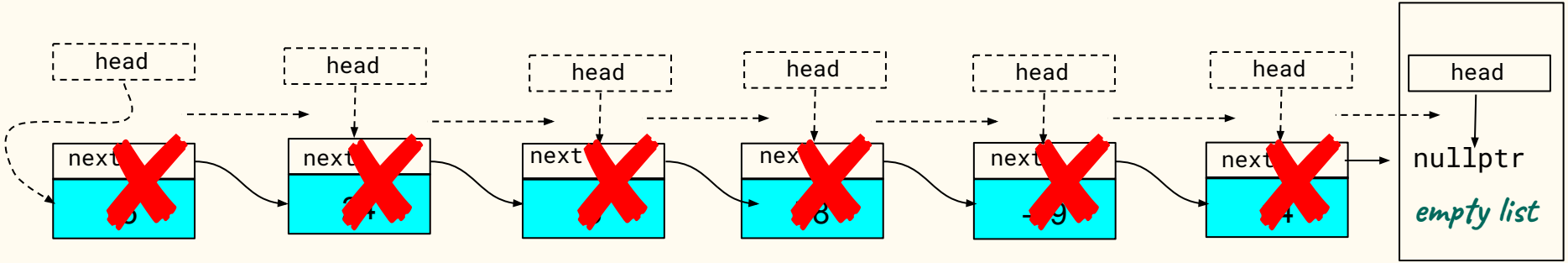
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        delete head_ptr;
    }
}
```

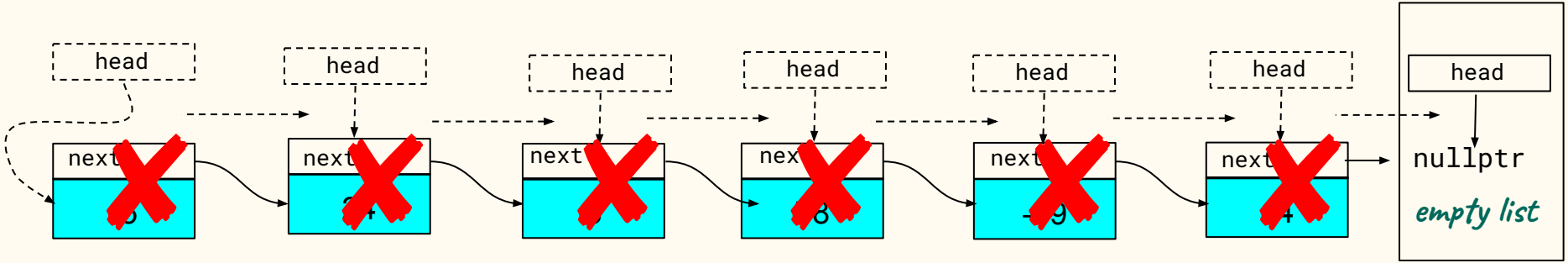
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        delete head_ptr;
        ---
    }
}
```

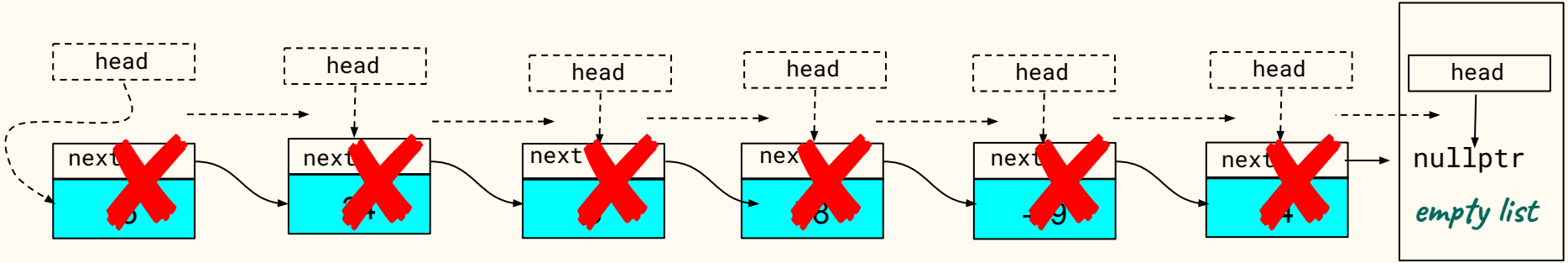
Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        delete head_ptr;
        _49_
    }
}
```

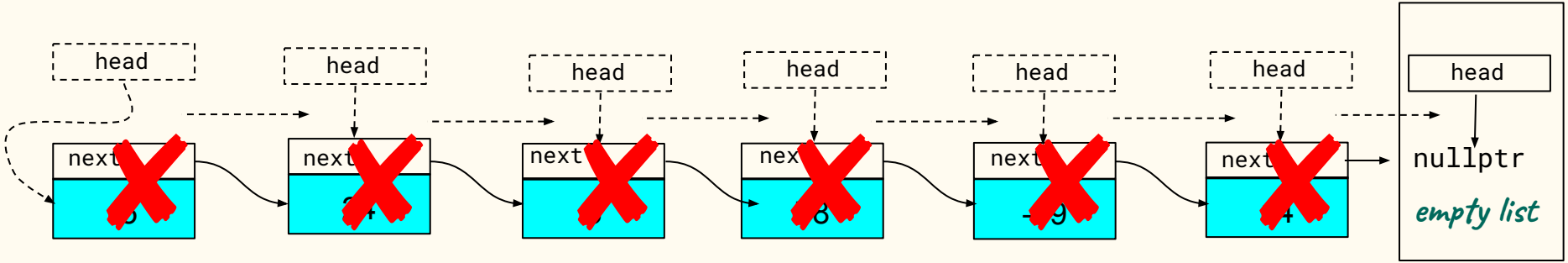
Which statement assigns `head_ptr` to the address of the next `Node` to remove from the list?




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        delete head_ptr;
        _49_
    }
}
```

Removing all Nodes from a list




```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void clear_list(Node*& head_ptr) {
    while(head_ptr != nullptr) {
        Node* next = head_ptr->next;
        delete head_ptr;
        head_ptr = next;
    }
}
```