Linked List Implementation

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 30
1.0.0

Let’s Insert Four Nodes Onto Our Linked List

Thankfully we wrote `createLinkedList` and `insertFront`

It's Hard to Free Every Single Node and the Linked List

```c
int main(void) {
    linked_list_t *linked_list = createLinkedList();
    node_t *n4 = insertFront(linked_list, 4);
    node_t *n3 = insertFront(linked_list, 3);
    node_t *n2 = insertFront(linked_list, 2);
    node_t *n1 = insertFront(linked_list, 1);
    printList(linked_list);
    free(linked_list);
    free(n1);
    free(n2);
    free(n3);
    free(n4);
    return EXIT_SUCCESS;
}
```

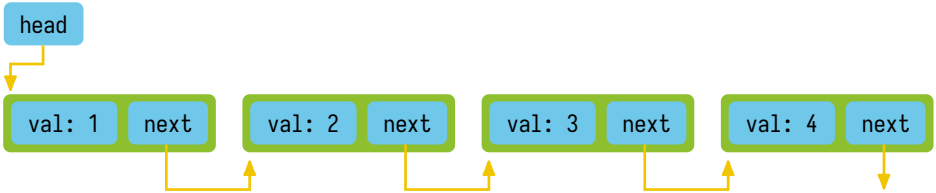
We Can Write a Function to Free Everything For Us

This function should just take a pointer to a linked list,
free it, and free all nodes (be careful of use after free!)

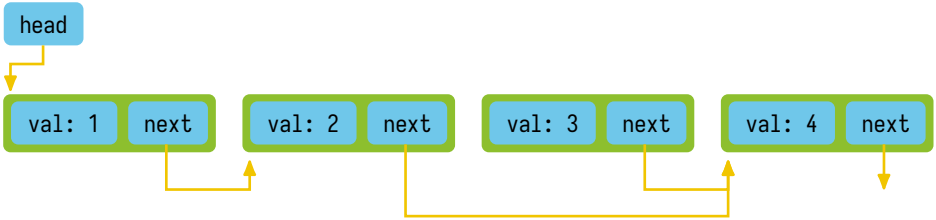
We Can Free After Getting the Pointer We Need

```
void freeLinkedList(linked_list_t *linked_list) {
    node_t *current = linked_list->head;
    free(linked_list);
    while (current != NULL) {
        node_t *next = current->next;
        free(current);
        current = next;
    }
}
```

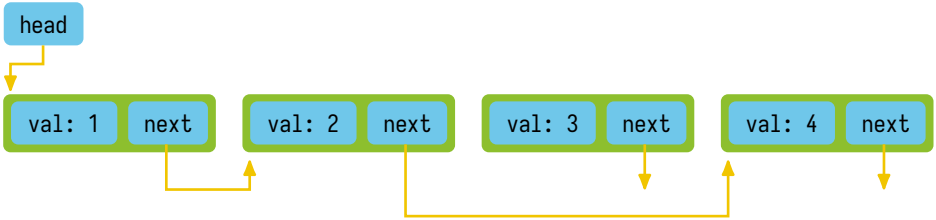
How Do We Remove a Node (3) From Our List?



How Do We Remove a Node (3) From Our List?



How Do We Remove a Node (3) From Our List?



To Remove a Node We Need to Know the Previous One

We could create a variable called `previous` initialized to `NULL`

Iterate through the list until we find our node, while updating the previous

Once we find the node to remove, we know the previous node as well

To Remove a Node We Need to Know the Previous One

We could create a variable called `previous` initialized to `NULL`

Iterate through the list until we find our node, while updating the previous

Once we find the node to remove, we know the previous node as well

Edge cases:

- Previous is `NULL` after the search, meaning this node is the head
- The node isn't in the list (crashing or doing nothing is okay)

This `removeNode` Does Nothing for a Node Not in the List

```
void removeNode(linked_list_t *linked_list, node_t *node) {
    node_t *current = linked_list->head;
    node_t *previous = NULL;
    while (current != node && current != NULL) {
        previous = current;
        current = current->next;
    }
    if (current == NULL) {
        return;
    }
    if (previous == NULL) {
        linked_list->head = current->next;
    }
    else {
        previous->next = current->next;
    }
    current->next = NULL;
}
```

We Could Also Think About the Problem More Indirectly

Check the address values for head and next until you find the nodes address

After finding your address, update the value to point to the node's next

This gets rid of the corner case with `previous`
(this may be very confusing at first)

Adapted from: https://github.com/mkirchner/linked-list-good-taste

We Can Re-Write `removeNode` to Check All Pointers

```
void removeNode(linked_list_t *linked_list, node_t *node) {
    node_t **p = &linked_list->head;
    while (*p != node && *p != NULL) {
        p = &(*p)->next;
    }
    if (*p == NULL) {
        return;
    }
    *p = node->next;
}
```

Note: we can remove the `if` statement to segfault when the node isn't found

We Can Write Other Useful Functions

When we use lists, we may want to know if the list is empty, or the length of the list (let's try writing both)

We Could Also Write Length Recursively

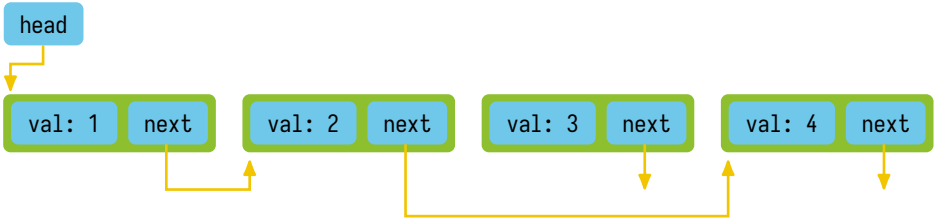
```
bool isEmpty(linked_list_t *linked_list) {
    return linked_list->head == NULL;
}

int length(linked_list_t *linked_list) {
    int len = 0;
    node_t *current = linked_list->head;
    while (current != NULL) {
        len += 1;
        current = current->next;
    }
    return len;
}
```

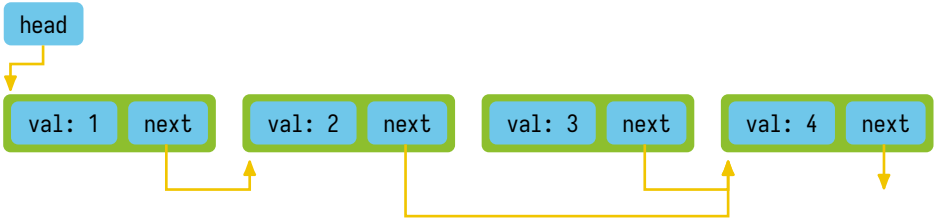
Note: we could write `isEmpty` in terms of `length`

```
  return length(linked_list) == 0;
```

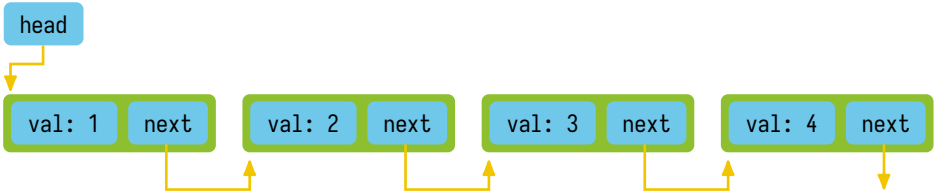
How Would We Insert a Node (3) After Another Node (2)?



How Would We Insert a Node (3) After Another Node (2)?



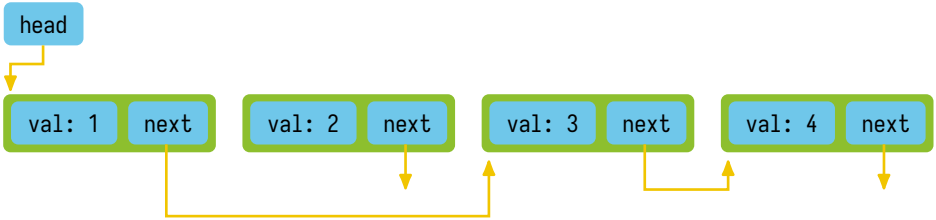
How Would We Insert a Node (3) After Another Node (2)?



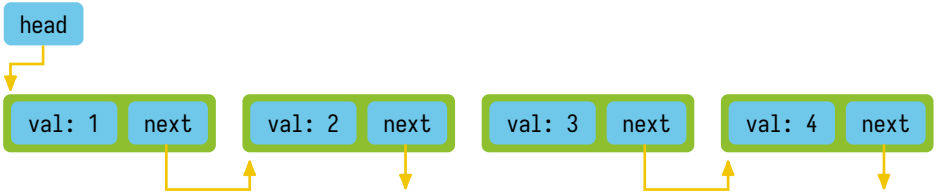
We Assume That `after` is in the List

```
void insertAfter(node_t *after, node_t *node) {
    node->next = after->next;
    after->next = node;
}
```

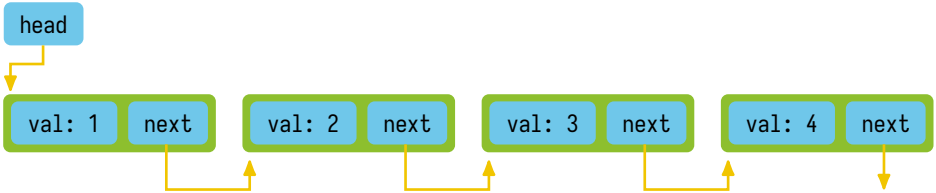
What About Inserting Node (2) Before Node (3)?



What About Inserting Node (2) Before Node (3)?



What About Inserting Node (2) Before Node (3)?



Our Implementation Needs `previous`, Like `removeNode`

```
void insertBefore(linked_list_t *linked_list, node_t *before, node_t *node) {
    node_t *current = linked_list->head;
    node_t *previous = NULL;
    while (current != before) {
        previous = current;
        current = current->next;
    }
    if (previous == NULL) {
        linked_list->head = node;
    }
    else {
        previous->next = node;
    }
    node->next = before;
}
```

We Could Create a Function for Our Common Operation

```
node_t **findIndirect(linked_list_t *linked_list, node_t *target) {
    node_t **p = &linked_list->head;
    while (*p != target) {
        p = &(*p)->next;
    }
    return p;
}
void removeNode(linked_list_t *linked_list, node_t *node) {
    node_t **p = findIndirect(linked_list, node);
    *p = node->next;
    node->next = NULL;
}
void insertBefore(linked_list_t *linked_list, node_t *before, node_t *node) {
    node_t **p = findIndirect(linked_list, before);
    *p = node;
    node->next = before;
}
```

What About a Function to Insert a New Node at the Back?

```
node_t *insertBack(linked_list_t *linked_list, int val) {
    node_t *node = createNode(val);
    node_t **p = findIndirect(linked_list, NULL);
    *p = node;
    return node;
}
```

But this code looks familiar...

What About a Function to Insert a New Node at the Back?

```
node_t *insertBack(linked_list_t *linked_list, int val) {
    node_t *node = createNode(val);
    node_t **p = findIndirect(linked_list, NULL);
    *p = node;
    return node;
}
```

But this code looks familiar...

```
void insertEnd(linked_list_t *linked_list, node_t *node) {
    insertBefore(linked_list, NULL, node);
}
node_t *insertBack(linked_list_t *linked_list, int val) {
    node_t *node = createNode(val);
    insertEnd(linked_list, node);
    return node;
}
```

Another Function Can Remove and Return the First Node

```
node_t *removeFront(linked_list_t *linked_list) {
    node_t *node = linked_list->head;
    if (node != NULL) {
        linked_list->head = node->next;
    }
    return node;
}
```

We Can Now Move a Node from the Front to the Back

```
int main(void) {
    linked_list_t *linked_list = createLinkedList();
    insertBack(linked_list, 1);
    insertBack(linked_list, 2);
    insertBack(linked_list, 3);
    insertBack(linked_list, 4);
    printLinkedList(linked_list);
    node_t *n = removeFront(linked_list);
    insertEnd(linked_list, n);
    printLinkedList(linked_list);
    freeLinkedList(linked_list);
    return EXIT_SUCCESS;
}
```

All of the Linked List Functions We Wrote Today

```
void freeLinkedList(linked_list_t *linked_list);
bool isEmpty(linked_list_t *linked_list);
int length(linked_list_t *linked_list);
void insertAfter(node_t *after, node_t *node);
void insertBefore(linked_list_t *linked_list, node_t *before, node_t *node);
void insertEnd(linked_list_t *linked_list, node_t *node);
node_t *insertBack(linked_list_t *linked_list, int val);
node_t *removeFront(linked_list_t *linked_list);
```