

CS 521: Systems Programming

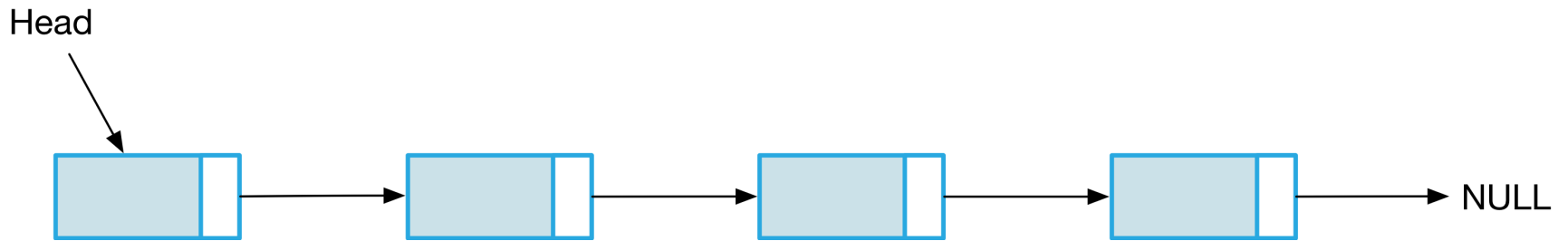
Data Structures: Linked Lists

Lecture 8

Linked Lists

- We probably all know linked lists already, and they tend to be one of those things programmers implement early on in their careers!
- Linked lists work particularly well in C
 - We can incrementally allocate memory for the list items
 - Inserting new items is trivial
- They are often a good fit in concurrent programs because changes can be *localized*

A Basic Linked List



Implementation [1/2]

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

- We'll start with a pointer to the head of the list
- Then we have our list elements...
 - How should we represent a list element?
- Using a `struct`, we can hold data and a pointer to the next `struct` in the chain (singly-linked list)

Implementation [2/2]

- Each struct maintains a `next` pointer to another struct
- Once we hit `NULL` we know we've reached the end of the list
- We can also add a `prev` pointer to create a doubly-linked list
 - Additionally a *tail* pointer can be useful to allow nodes to be added at the beginning or end

Motivation: Why Linked Lists?

- Linked lists tend to be compared to arrays (or things like ArrayLists)
 - They really **aren't** meant for the same purpose, though
- A linked list is great when you will be adding or removing many items
 - Don't need to shift things around in memory or resize allocations
- Linked lists are **BAD** if you will search them frequently or want to access them via indexes

More Motivation

- Linked lists are also a great opportunity to get to play with double pointers!
- We can have `insert()`, `delete()`, etc. update the pointer to the list head in place
- Plus, there will be regular pointers and `structs` too...

Insert

- Allocate memory for the new node
- Update the new node's data/value
- Set its **next** pointer to the current head
- Update the **head** pointer
 - Should now point to the newly-inserted node
 - How do we do this? Can it be done with a single pointer?

Append (add to the end)

1. Loop through the array until we find a node whose **next** pointer points at `NULL`
 - (End of the list)
2. Allocate memory for the new node
3. Update the new node's data/value
4. Set the **next** pointer to `NULL` (new end of list)
5. Set the old last node's **next** pointer to the new node

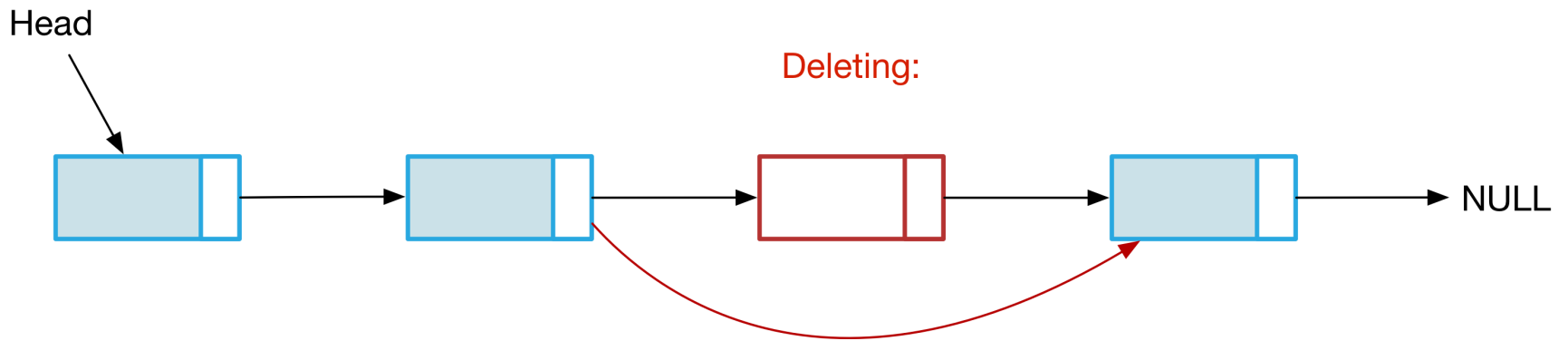
Print [1/2]

- Use a temporary variable to store the current node
- Start with `current = head`
- While the current node isn't `NULL` :
 - Print its value
 - Move to the next node

Print [2/2]

```
void print(struct list_node* head_p) {
    struct list_node *curr = head_p;
    while (curr != NULL) {
        printf("%d -> ", curr->data);
        curr = curr->next;
    }
    printf("\n");
}
```

Delete [1/2]



Delete [2/2]

- Find the node in question
- Update the previous node's **next** pointer
- **Remember**: in C, we have to take care of freeing memory ourselves!
- What happens if we delete the head or tail?

Search

- This is the one where linked lists struggle
- Loop through, checking every element
- $O(n)$

Lab 5

- Let's implement a linked list!