CS 220: Introduction to Parallel Computing

Linked Lists

Lecture 12

Today's Agenda

- More Pointers
- Linked Lists

Today's Agenda

- **More Pointers**
- Linked Lists

Why Pointers?

- From a fundamental standpoint, there are two reasons why we need pointers in C
- What are they?

1. C only supports passing by value
   - We cannot modify variables that are passed into a function **unless** they are pointers
2. Dynamic memory
   - We need to have a way to refer to data on the heap

Understanding Pointers

- To get a sense of how pointers actually work, it is useful to think about how memory is organized
  - After all, pointers refer to **memory addresses**
- Let's look at:
  - The memory address
  - The variable's **name**
  - The variable's **value**

In Memory

- `int a = 12;`
- `int b = 15;`
- `int *c = &b;`
- `int **d = &c;`

- `*c = ?`
- `*d = ?`
- `**d = ?`

Address	Variable Name	Value
1000	a	12
1001	b	15
1002	c	1001
1003	d	1002

Double Pointers

- Why do we need double pointers?
- Arrays of arrays:
  - char **argv;
- We can change the **value** of a variable from inside another function with a single pointer
- We can change the what a pointer points at from inside another function with a **double pointer**

Today's Agenda

- More Pointers
- **Linked Lists**

Linked Lists

- We all know and love linked lists, or at least there's a good chance you've implemented one in the past!
- Linked lists work well in C because we can incrementally allocate memory for the list items
- Deleting, inserting, etc are all fairly manageable operations that only impact one list node (and its neighbors)

Implementing a Linked List

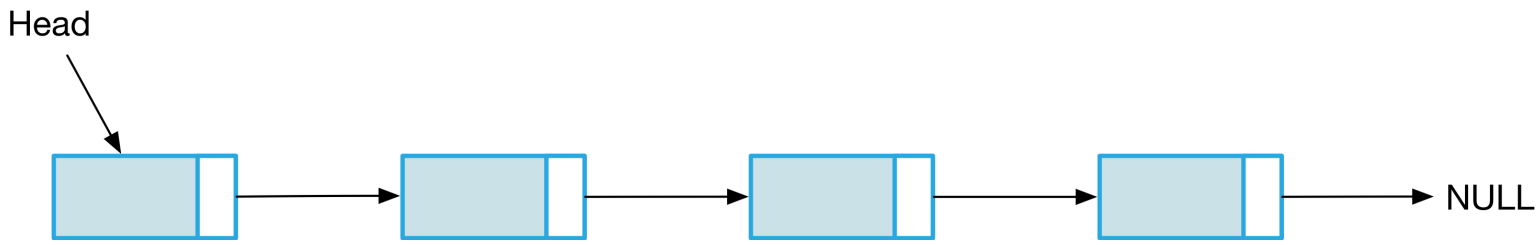
- We'll start with a pointer to the head of the list
- Then we have our list elements…
  - How should we represent a list element?
- Using a struct, we can hold data and a pointer to the next struct in the chain
  - (singly-linked list)

Node Struct

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

Linked Lists

- Here's a linked list with four elements:



- We maintain a pointer to the first element (head)
- Each element maintains a pointer to the next element
- The last element points to NULL

Insert

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

Append

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

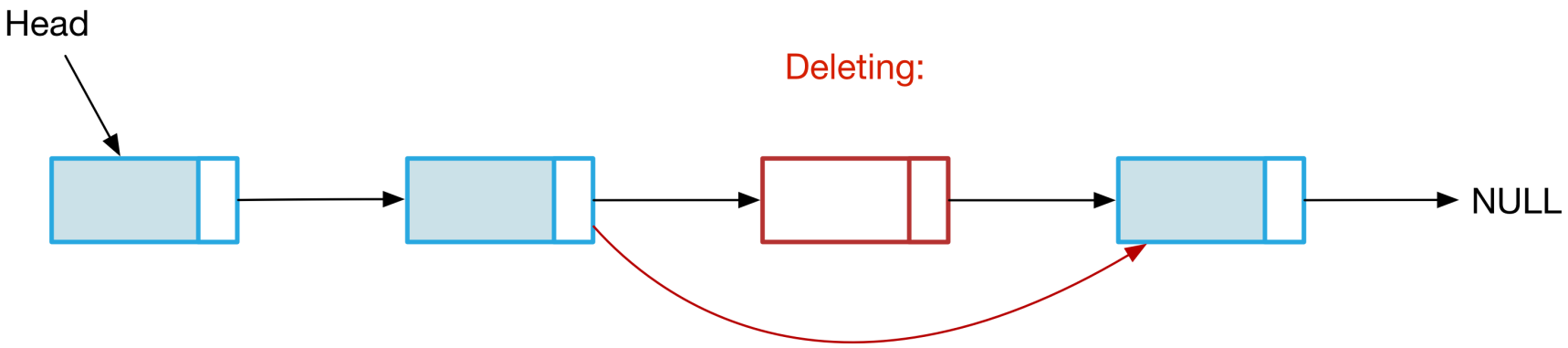
Print (1/2)

1. Use a temporary variable to store the current node
2. Start with current = head
3. 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



Delete (2/2)

1. Find the node in question
2. Update the previous node's **next** pointer
3. Print out the values to help orient yourself
4. **Remember**: in C, we have to take care of freeing memory ourselves!