We can define a struct containing a pointer to the same type of struct:

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

- we can use the `next` field to link these structs in a list
- each `struct` contains data and a pointer to the next `struct` in the list
- need to separately store a pointer to the first `struct` (the `head`)
- the `next` field last item in the list contains NULL

![A program's memory (not to scale)](image)
For simplicity, we often assume each list element only stores a single int.

It real code it might store a large amount of data or many fields, e.g.:

```c
struct address_node {
    struct address_node *next;
    char *name;
    char *address;
    char *telephone;
    char *email;
};
```
A function to add a node

We’ve seen a similar function for creating a struct

```
// Create a node using the data and next pointer provided
// Return a pointer to this node
struct node *create_node(int data, struct node *next) {
    struct node *n;
    n = malloc(sizeof(struct node));
    n->data = data;
    n->next = next;
    return n;
}
```

source code for linked_list.c
Building a list using only create_node()

```c
int main(void) {
    // head will always point to the first element of our list
    struct node *head = create_node(5, NULL);
    head = create_node(4, head);
    head = create_node(3, head);
    head = create_node(2, head);
    head = create_node(1, head);
    print_data(head);
    return 0;
}
```

source code for linked_list.c
create_node makes a node with a NULL next and we point head at it
How it works 2

The 2nd node points its “next” at the old head, then it replaces head with its own address.
The process continues . . .

[Diagram of a linked list with nodes labeled 3, 4, and 5, and a NULL pointer at the end. The diagram is labeled "A program's memory (not to scale)."
Linked lists don’t have indexes . . .

We can’t loop through them in the same way as arrays

We have to follow the links from node to node

If we reach a NULL node pointer, it means we’re at the end of the list

```c
// Loop through a list of nodes, printing out their data
void print_data(struct node *n) {
    while (n != NULL) {
        // have printed from list's head up to just
        printf("%d\n", n->data);
        n = n->next;
    }
}
```

source code for linked_list.c
Looping through a Linked List

Start with a pointer that’s a copy of Head
Looping through a Linked List

After you’re finished with a node, copy its Next pointer to reach the next node.
Looping through a Linked List

Eventually, copying the next pointer results in NULL.

That’s when the loop stops
Let’s use a Linked List to track the players in a game

We’re going to start by adding players to the game

We want to be able to print all the players that are currently in the game (the list of players can change as the game goes on)

We might want to control the order of the list, so we need to be able to insert at a particular position

We also want to be able to find and remove players from the list if they’re knocked out of the round
What will our nodes look like?

We’re definitely going to want a basic node struct

Let’s start with a name

And a pointer to the next node

```c
struct player {
    struct player *next;
    char name[MAX_NAME_LENGTH];
};
```

source code for battle_royale.c
We’ll want a function that creates a node

```c
// Allocate memory for a player node
// Populate that node with new_name and new_next
// Return a pointer to the allocated memory

struct player *create_player(char new_name[MAX_NAME_LENGTH], struct player *new_next) {
    struct player *new_player = malloc(sizeof(struct player));
    new_player->next = new_next;
    strcpy(new_player->name, new_name);
    return new_player;
}
```

source code for battle_royale.c
Creating the list itself

Note that we don’t need to specify the length of the list!

```c
int main(void) {
    struct player *head = create_player("Marc", NULL);
    head = create_player("Chicken", head);
    head = create_player("Shrey", head);
    head = create_player("Anusha", head);
}
```

source code for battle_royale.c
Using `create_player #1`

Head points at the First Player, its next is NULL
Using `create_player #2`

The New Player is created and copies the head pointer for its next

Next copies head
Using `create_player #3`

`create_node` returns a pointer to `New Player`, which is assigned to `head`

Next copies `head`
Printing out the list of players

How do we traverse a list to see all the elements in it?

- Loop through, starting with the pointer to the head of the list
- Use whatever data is inside the player node
- Then move onto the next pointer from that player node
- If the pointer is NULL, then we’ve reached the end of the list

// Takes the head of the list
// Prints out everyone who is in the list

```c
void print_players(struct player *head) {
    struct player *current = head;
    while (current != NULL) {
        // I'm looking at a valid player
        printf("%s\n", current->name);
        current = current->next;
    }
}
```

source code for battle_royale.c
What else does the program need?
- Add players to the game
  - Inserting into a list
- Maintain a list of players that’s in order
  - Inserting into a specific position in a list
Inserting Nodes into a Linked List

Linked Lists allow you to insert nodes in between other nodes
We can do this by simply aiming next pointers to the right places
We find two linked nodes that we want to put a node between
We take the next of the first node and point it at our new node
We take the next of the new node and point it at the second node
This is much less complicated with diagrams . . .
Before we’ve tried to insert anything
Create a node

A new node is made, it’s not connected to anything yet
Connect the new node to the second node

Alter the **next** pointer on the New Node
Connect the first node to the new node

Alter the **next** pointer on the First Node
We can use insertion to have greater control of where players end up

In this example, Chicken is inserted after the head (Marc), then Aang is also inserted after Marc (and before Chicken)

```c
int main(void) {
    // create the list of players
    struct node *head = NULL;
    create_player(head);
    head->next = create_player("Chicken", head);
    head->next = insert_after("Aang", head);
    print_players(head);
    return 0;
}
```
Insertion with some conditions

- **We can now insert into any position in a Linked List**
- We can read the data in a node and decide whether we want to insert before or after it
- Let’s insert our elements into our list based on alphabetical order
- We’re going to use a `string.h` function, `strcmp()` for this
- `strcmp()` compares two strings, and returns
  - 0 if they’re equal
  - negative if the first has a lower ascii value than the second
  - positive if the first has a higher ascii value than the second
Finding where to insert

We’re going to loop through the list

This loop assumes the list is already in alphabetical order
Each time we loop, we’re going to keep track of the previous player
We’ll test the name of each player using `strcmp()`
We stop looping once we find the first name that’s “higher” than ours
Then we insert before that player
Finding the insertion point

Attempting to insert a player with name: "C" into a sorted list while maintaining the alphabetical order

Insert between these two players
// Inserts into a list (head) alphabetically.
// Assumes the list is already in alphabetical order
// Returns the head of the list (it may have changed)
struct player *insert_alpha(struct player *head, char new_name[MAX_NAME_LENGTH]) {
    // loop through and find the insertion point
    struct player *current = head;
    struct player *previous = NULL;
    while (current != NULL && strcmp(current->name, new_name) < 0) {
        // go past any names that are earlier than us alphabetically
        previous = current;
        current = current->next;
    }
    // current is now the first player in the list with a name "higher or equal"
    // than new_name
    // previous is now the last player in the list with a name "lower"
    // than new_name
    if (previous == NULL) {
        // Inserting at the head of a list
        // also works for an empty list (head == NULL)
        head = create_player(new_name, head);
    } else {
        // Inserting further down the list
        previous->next = create_player(new_name, previous->next);
    }
    return head;
}
If we want to remove a specific player, given their name

We need to look through the list and see if a player name matches the one we want to remove.

To remove, we’ll use next pointers to connect the list around the player node.

Then, we’ll free the node itself that we don’t need anymore.
Removing a player node

If we want to remove the Second Player

A program's memory (not to scale)
Skipping the player node

Alter the First Player’s **next** to bypass the player node we’re removing.
Freeing the removed node

Free the memory from the now bypassed player node
Finding the right player

Loop until you find the right match

This is very similar to finding the insertion point earlier

```c
// Loop through list looking for a match of
// player's name to rem_name
// Remove and free that node (if it exists)
// Return the possibly-changed head of the list
// Note: head only changes if we remove the first player
struct player *remove_player(struct player *head,
                              char rem_name[MAX_NAME_LENGTH]) {

    // Loop through and stop with curr->name == rem_name
    struct player *curr = head;
    struct player *prev = NULL;
    while (curr != NULL && strcmp(rem_name, curr->name) != 0) {
        prev = curr;
        curr = curr->next;
    }
}
```

source code for battle_royale.c
// Loop through list looking for a match of
// player's name to rem_name
// Remove and free that node (if it exists)
// Return the possibly-changed head of the list
// Note: head only changes if we remove the first player

struct player *remove_player(struct player *head,
                              char rem_name[MAX_NAME_LENGTH]) {

    // Loop through and stop with curr->name == rem_name
    struct player *curr = head;
    struct player *prev = NULL;
    while (curr != NULL && strcmp(rem_name, curr->name) != 0) {
        prev = curr;
        curr = curr->next;
    }

    // Remove and free that node (if it exists)
    if (prev) {
        prev->next = curr->next;
        free(curr);
    } else {
        head = curr->next;
    }

    return head;
}

source code for battle_royale.c
In a Battle Royale, people are removed from the game one at a time until only one person is left. They are the winner!

We can create a list of players

We can make sure it’s in a nice alphabetical order

We can remove a single player from the list

Now we need to remove players one at a time

When there’s only one left, they are the winner!
Once our list is created, we can loop through the game

We print out the player list

Our user will tell us who was knocked out

If there’s only one player left, we stop looping
The Game Loop

This will keep running until we find a winner

```c
while (print_players(head) > 1) {
    head = remove_by_name(head);
    printf("----------\n");
}
printf("is the winner!\n");
free_list(head);
```

source code for main.c
Remember, All memory allocated (by malloc) needs to be freed

- We can run `dcc --leakcheck` to see whether there’s leaking memory
- What do we find?
- There are pieces of memory we’ve allocated that we’re not freeing!

Let’s write a function that frees a whole linked list

- Loop through the list, freeing the nodes
- Just be careful not to free one that we still need the pointer from!
Looping to free nodes

A program's memory (not to scale)

First Node
Next 3

Second Node
Next 2

Third Node
Next 1

NULL

head

Loop pointer
Looping to free nodes

A program's memory (not to scale)

First Node

Next
3

Second Node

Next
2

Third Node

Next
1

NULL

head

Save a pointer to the node we're freeing

Pointer to node we're freeing

Loop pointer

Loop pointer is copied from First Node's next before we free First Node
Loopying to free nodes

A program's memory (not to scale)

First Node

Next

3

Second Node

Next

2

Third Node

Next

1

NULL

We can free the node now that we've copied its Next

Head

Pointer to node we're freeing

Loop pointer
Looping to free nodes

A program's memory (not to scale)

First Node
Next 3

Second Node
Next 2

Third Node
Next 1
NULL

Pointer to node we're freeing

Move our pointers along to the next nodes

Loop pointer
// Free all elements of the list starting at head
void free_list(struct player *head) {
    struct player *curr = head;
    while (curr != NULL) {
        struct player *free_node = curr;
        curr = curr->next;
        free(free_node);
    }
}

source code for battle_royle.c
What have we written in this program?

- Creation of nodes
- Looping through a list
- Insertion of nodes into specific locations
- Finding locations using loops
- Removal of nodes

Managing memory (allocation and freeing)
A Challenge - randomisation

Can we remove a random player from the list?

- Look up the functions `rand()` and `srand()` in the C Standard Library
- We can generate a random number and loop that many times into the list
- Then remove that player
- We will probably want to track how many items are in the list also . . .