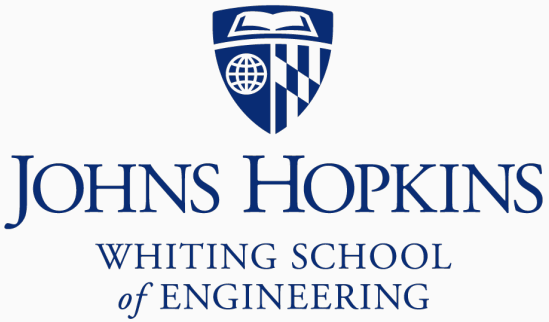
Linked lists

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

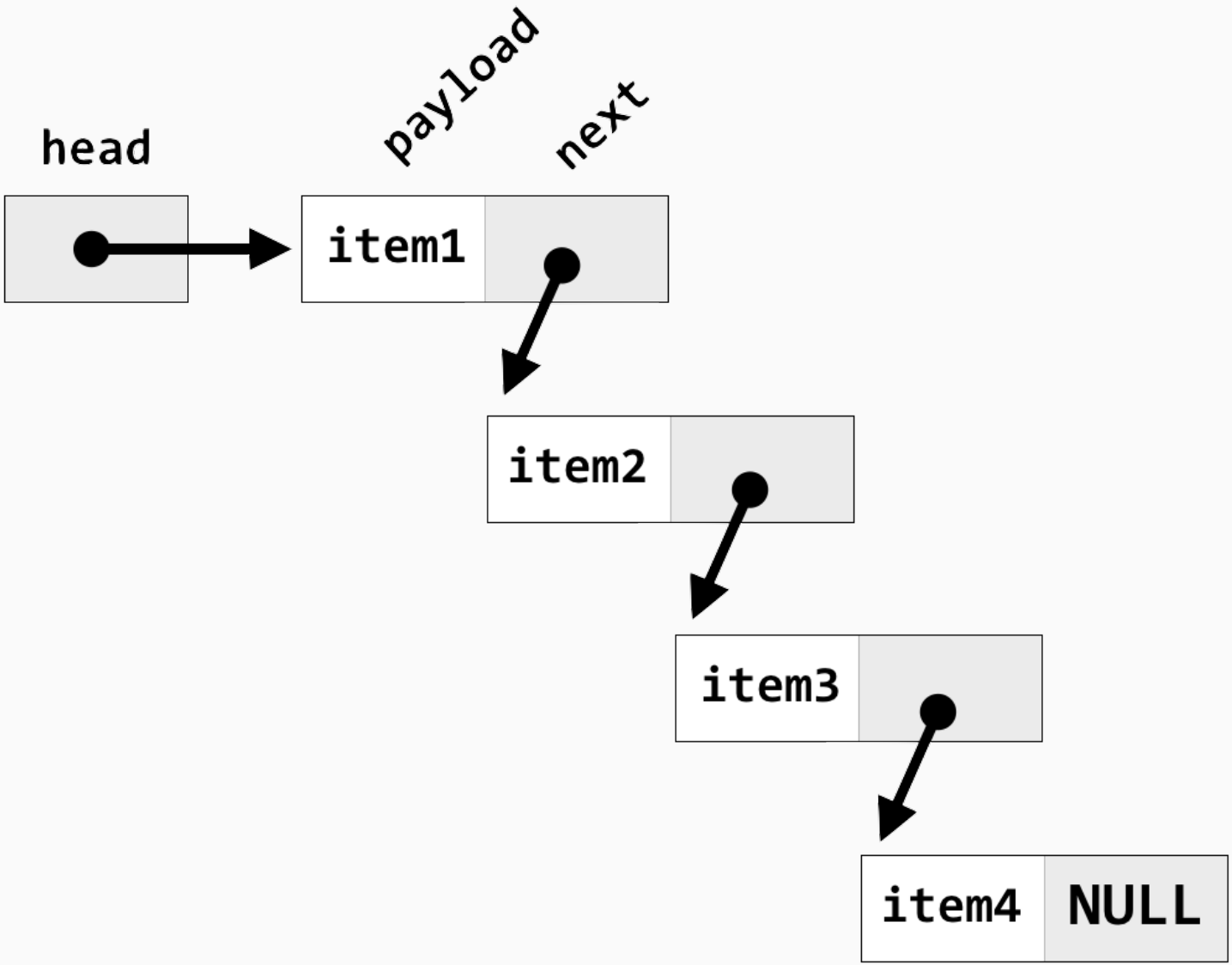
Linked lists

Sequence of structures (*nodes*) connected by pointers (*links*)

Each node has relevant data associated (*payload*)

- `music_collection` might have an `album` payload
- `calendar` might have a `date` payload

Linked lists



Linked lists

A linked-list node will be a `struct`

Two components:

- Payload
- Pointer to next node

```
struct Node {
    // ?
};
```

Linked lists

```
struct Album {
    const char *name;
    const char *artist;
    double length;
};

typedef struct _Node {
    struct Album payload;
    struct _Node *next;
} Node;
```

Linked lists

We will use this definition for Node:

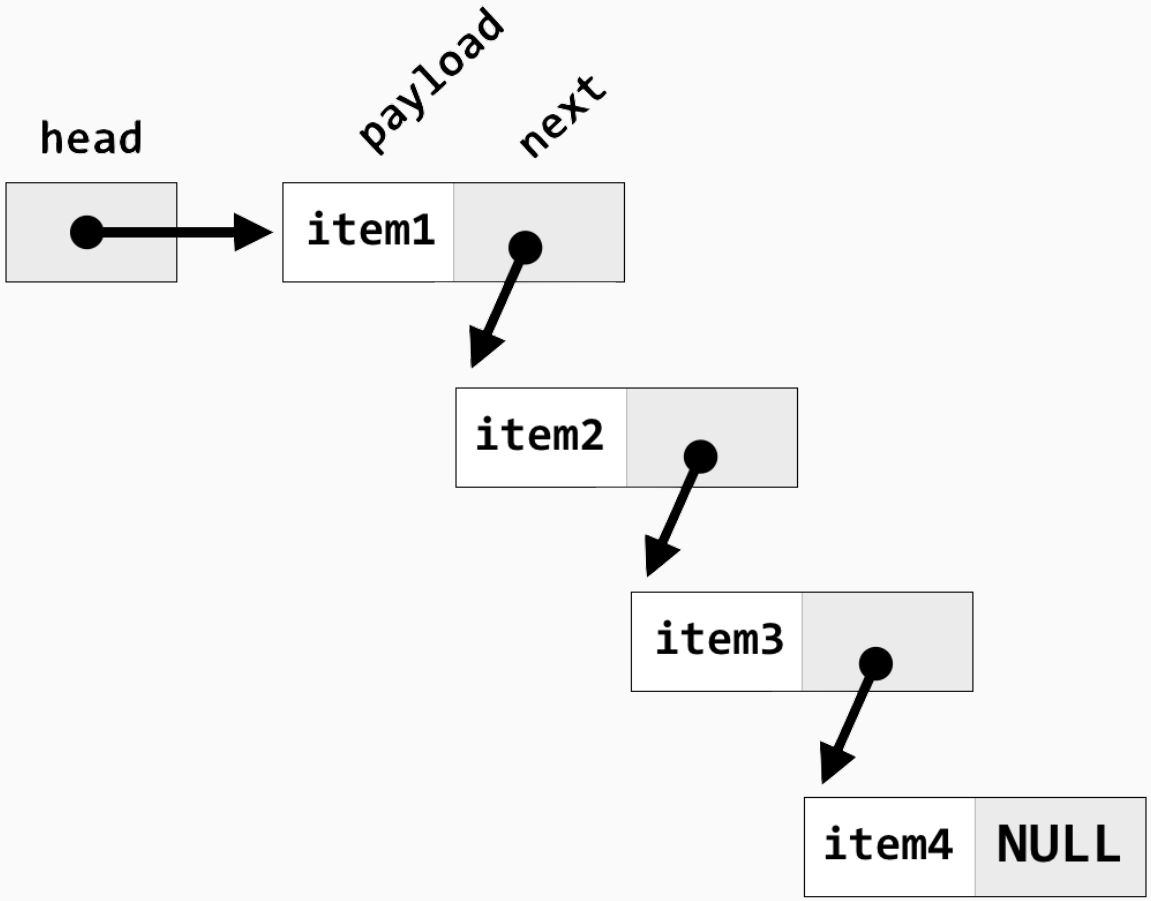
```
typedef struct _Node {
    struct Album payload;
    struct _Node *next;
} Node;
```

Has both a long name (`struct _Node`) and a typedef'ed short name (Node)

- Usually, we can simply use Node (thanks to typedef)
- When declaring the type for the next pointer, typedef isn't active yet, so we have to use `struct _Node`

Linked lists

Apart from the nodes, a *head pointer* to the first (*head*) node

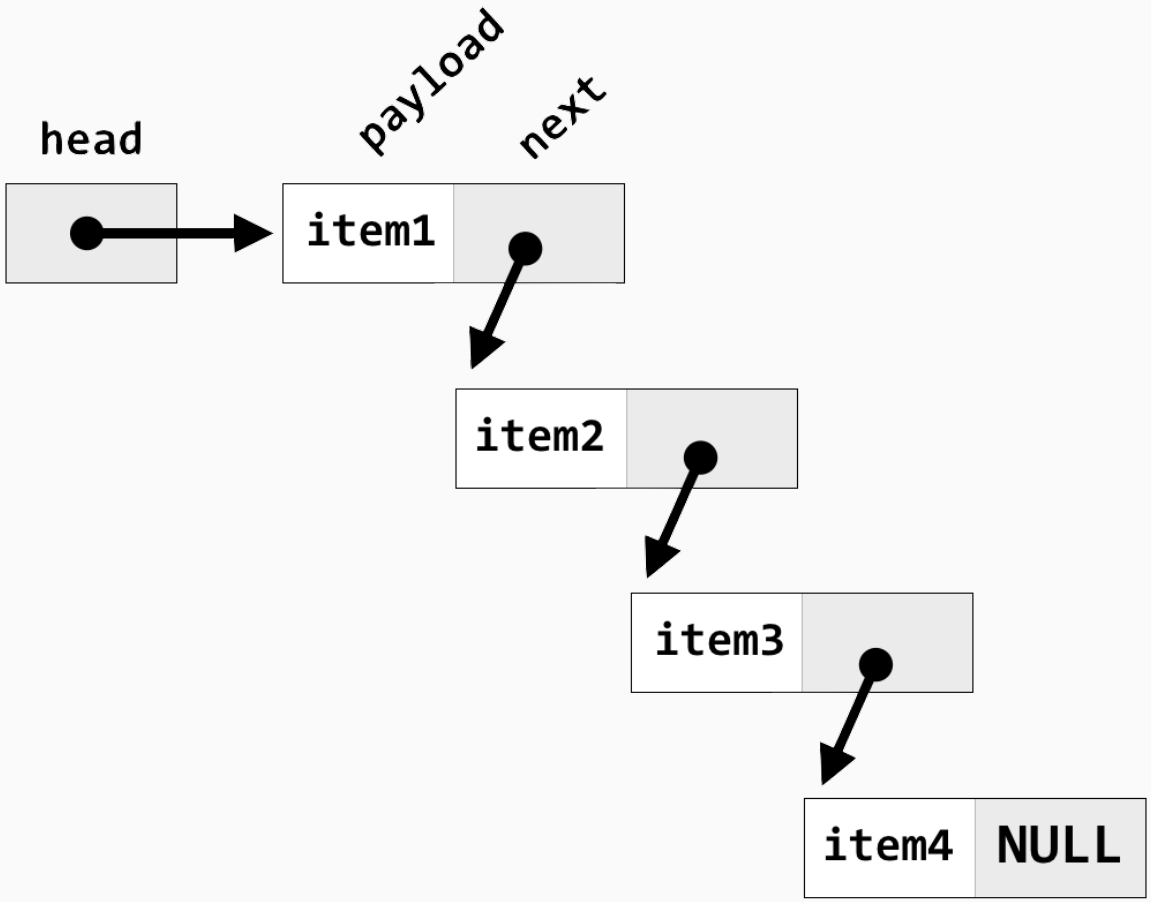


What is the head pointer's type?

Linked lists

Answer: `Node *`

Linked lists



How do we know when we've reached the end (*tail*) of the list?

Linked lists

Answer: tail's `next` pointer should equal `NULL`

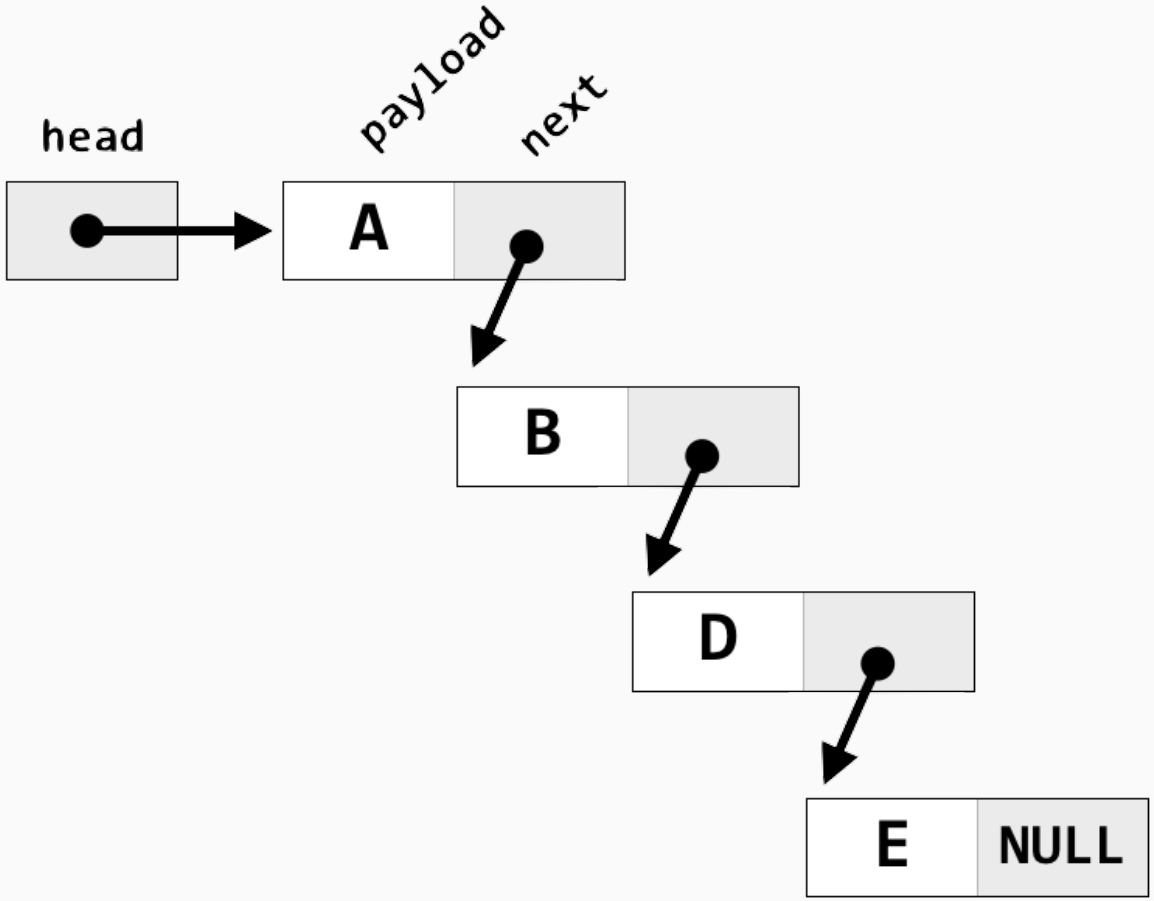
Linked lists

We'll write functions for three basic linked-list tasks:

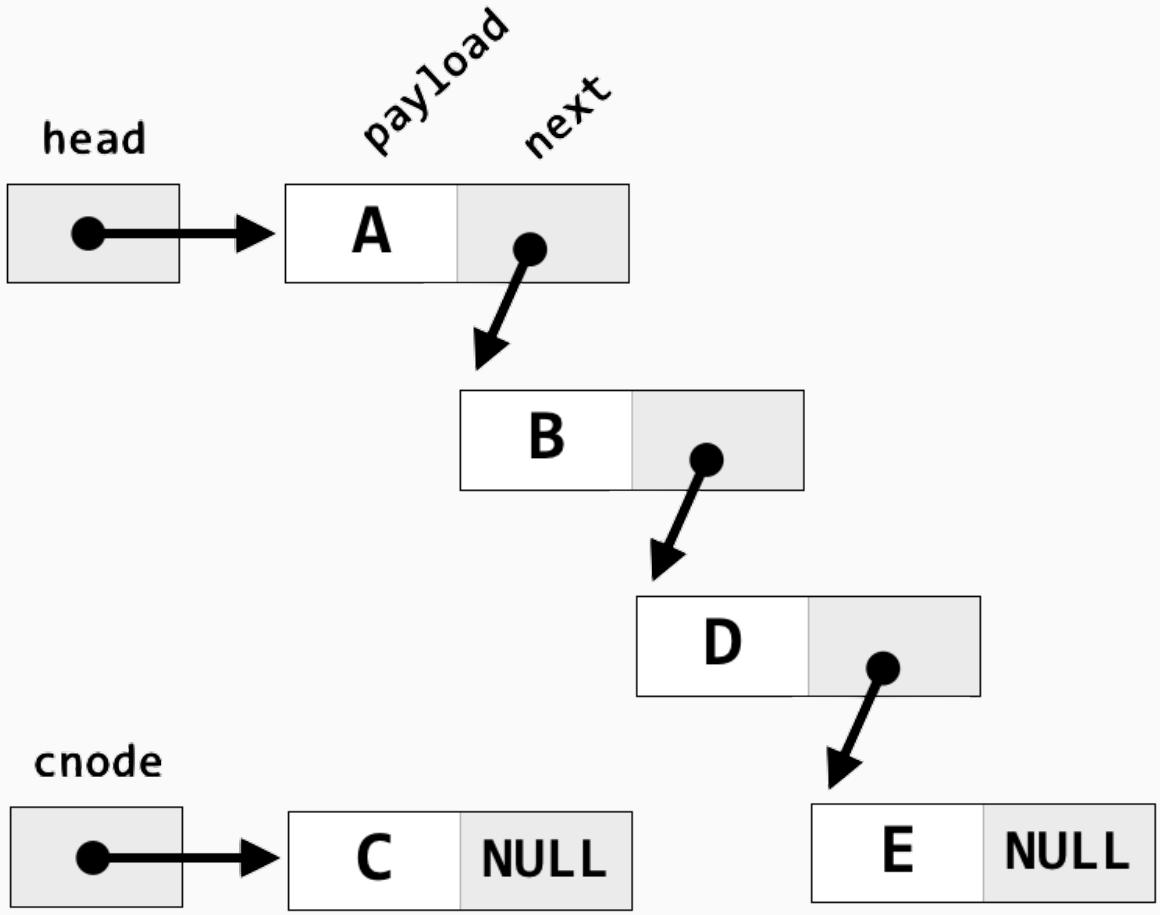
- Inserting an item
- Deleting an item
- "Walking along" ("traversing") the list

Linked lists: inserting

How to insert a new node?

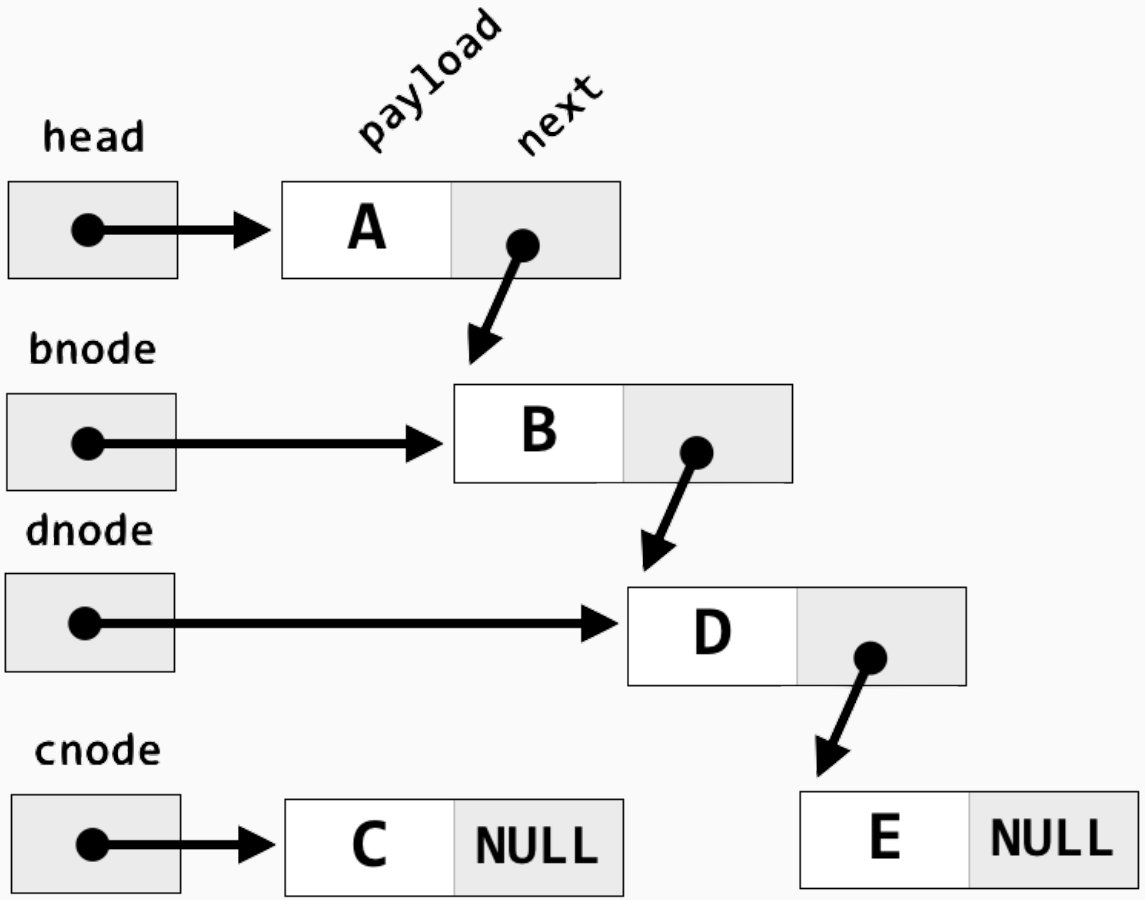


Linked lists: inserting



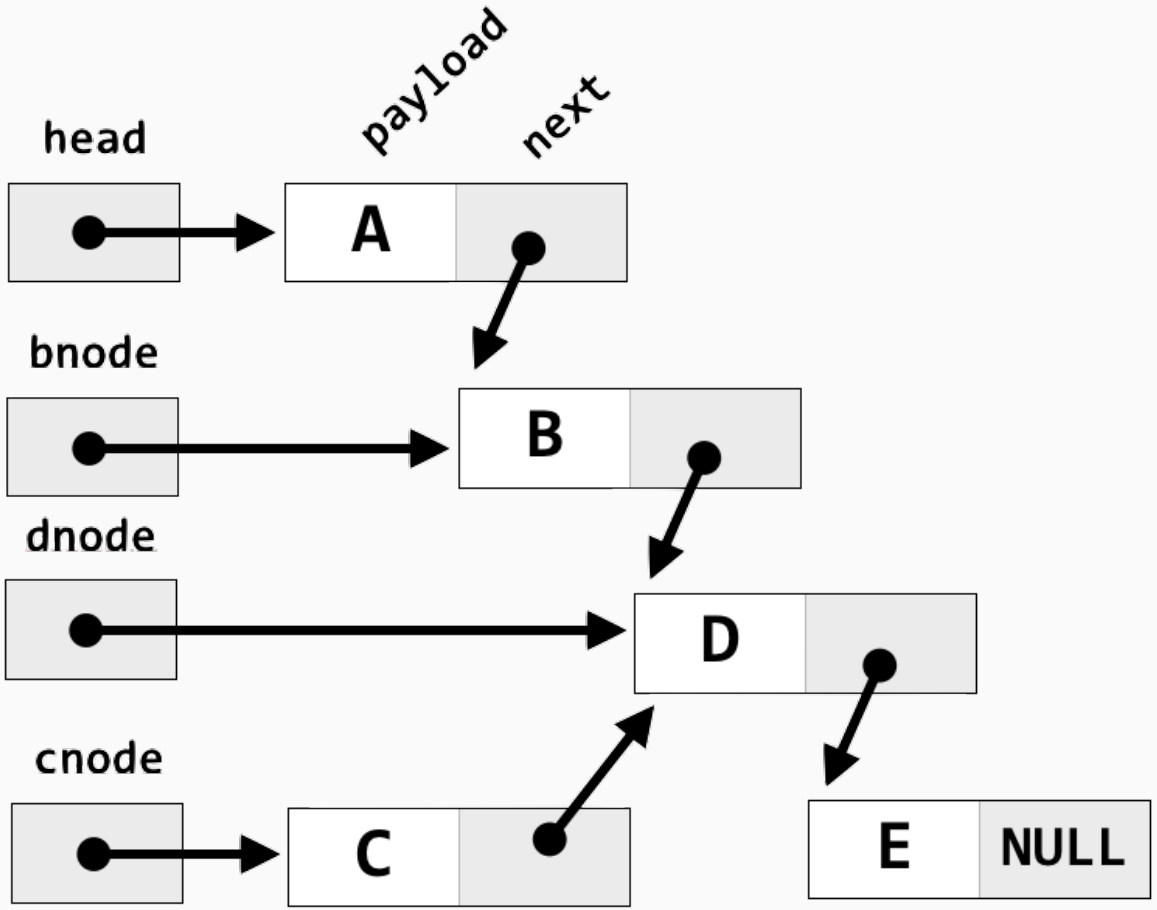

```
Node *cnode = malloc(sizeof(Node));
cnode->payload = 'C';
cnode->next = NULL;
```

Linked lists: inserting



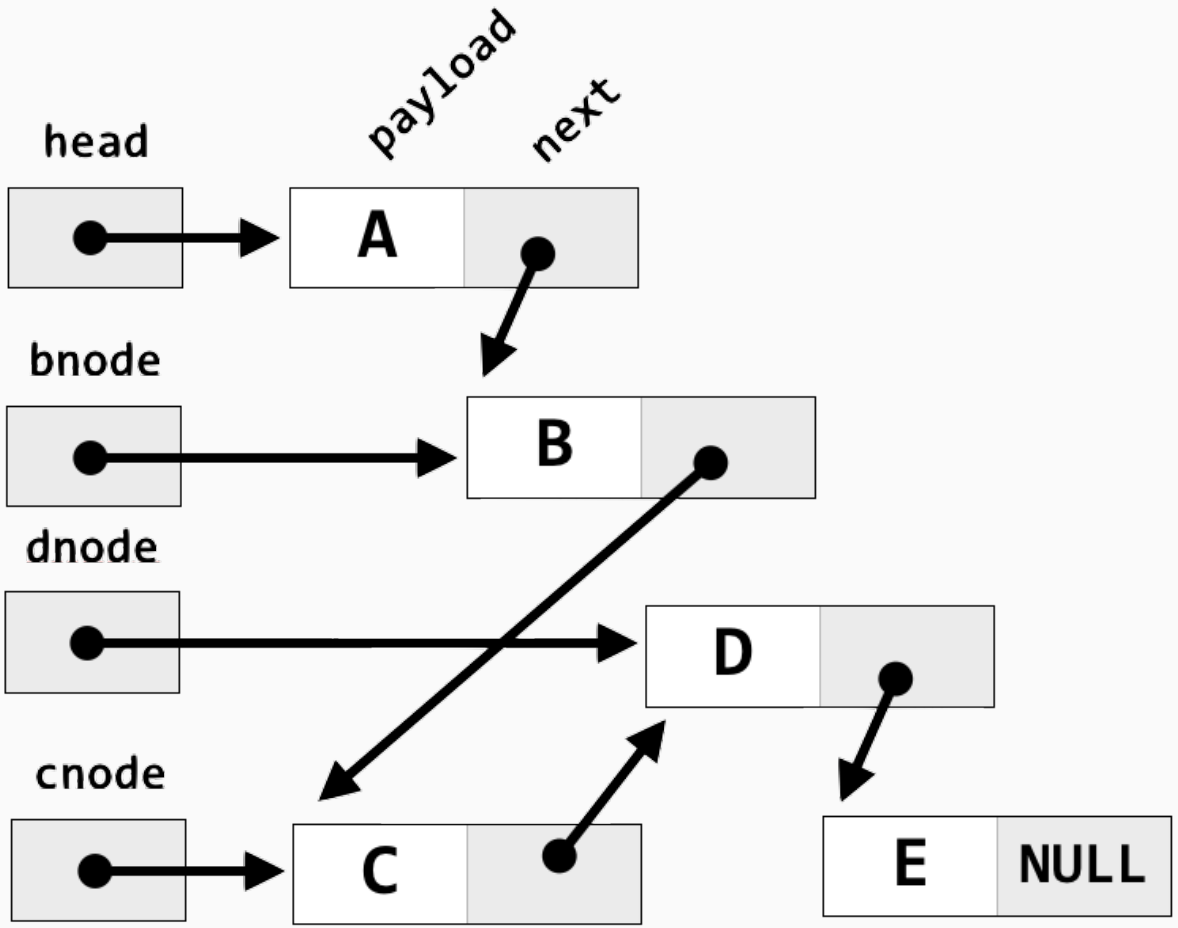

```
Node *bnode = head->next;
Node *dnode = bnode->next;
```

Linked lists: inserting



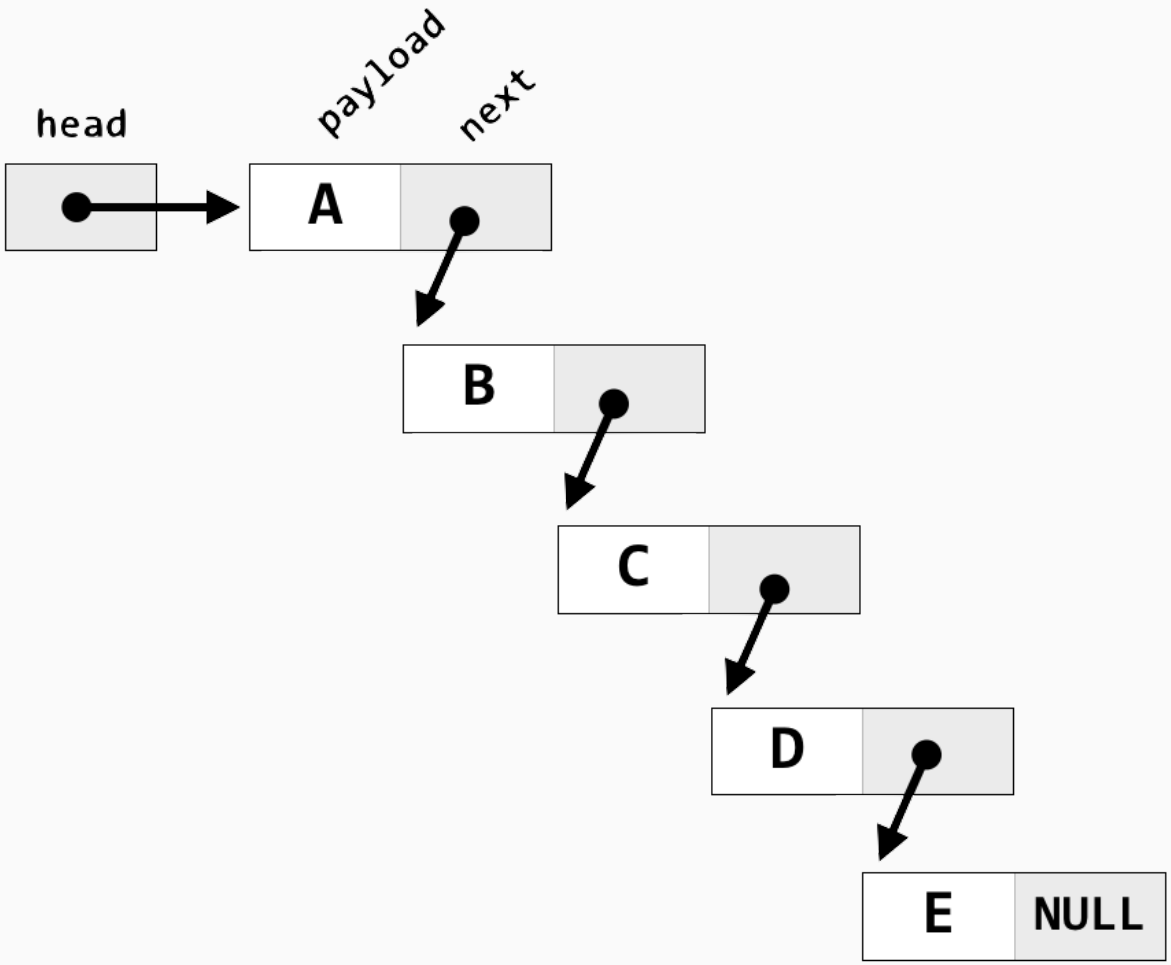

```
cnode->next = dnode; // make new node point to successor
```

Linked lists: inserting




```
bnode->next = cnode; // make predecessor point to new node
```

Linked lists: inserting



Linked lists: inserting

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct _Node {
    char payload;
    struct _Node *next;
} Node;

int insert_after(Node *before, char newPayload) {
    Node *newNode = malloc(sizeof(Node));
    if(newNode == NULL) {
        return -1; // error
    }
    newNode->payload = newPayload;
    newNode->next = before->next; // make new node point to successor
    before->next = newNode;       // make predecessor point to new node
    return 0;
}
```

Linked lists: inserting

```
// *Traverses* list, as we discuss later
void print_list(Node *cur) {
    while(cur != NULL) {
        printf("%c ", cur->payload);
        cur = cur->next;
    }
    putchar('\n');
}

int main() {
    // Make a little list
    Node *head = malloc(sizeof(Node));
    head->payload = 'A';
    head->next = NULL;
    insert_after(head, 'B');
    insert_after(head->next, 'C');
    // check for errors (omitting for brevity)
    print_list(head);
    // free everything (we'll see how later)
    return 0;
}
```

Linked lists: inserting

```
$ gcc -c ll_insert.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ll_insert ll_insert.o
$ ./ll_insert
A B C
```

Linked lists: inserting

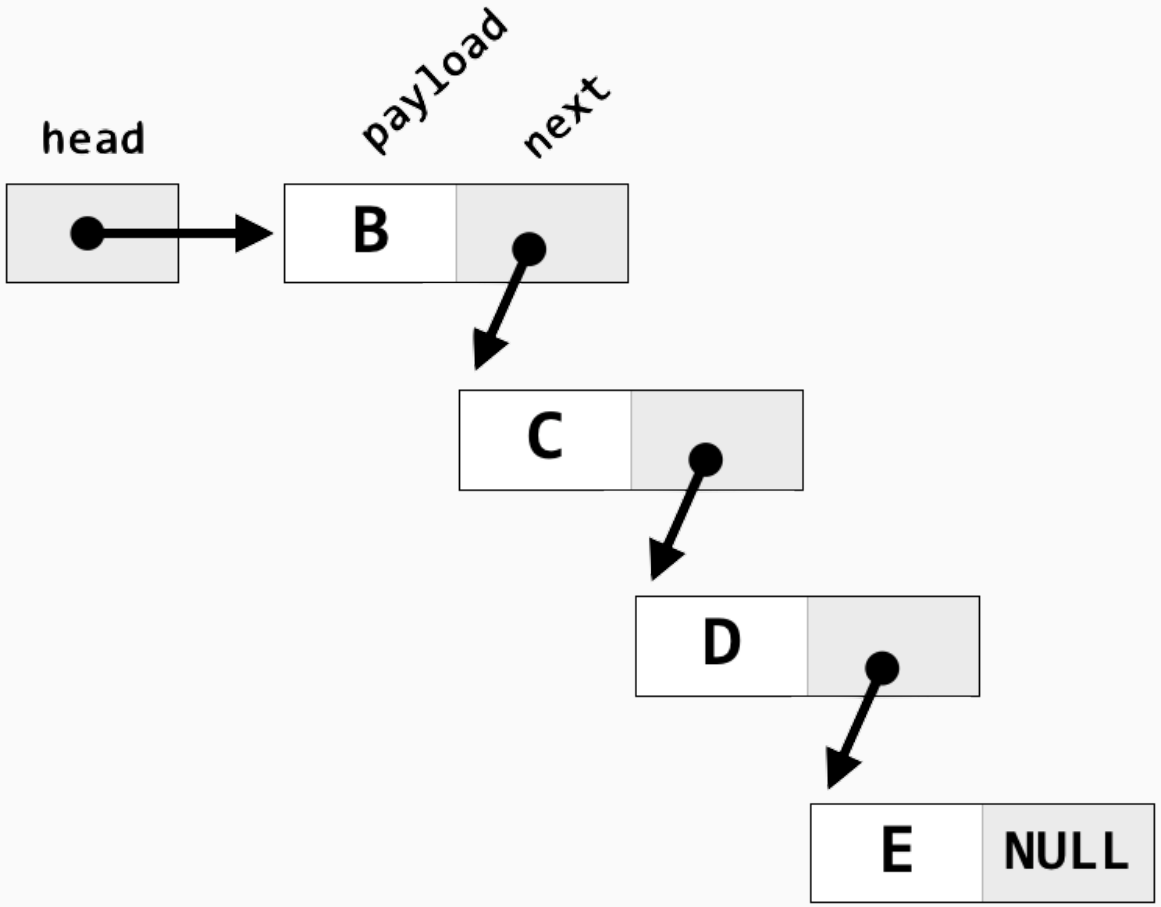
Insertion steps:

- Allocate new node
- Make new node's `->next` point to successor
- Make predecessor's `->next` point to new node

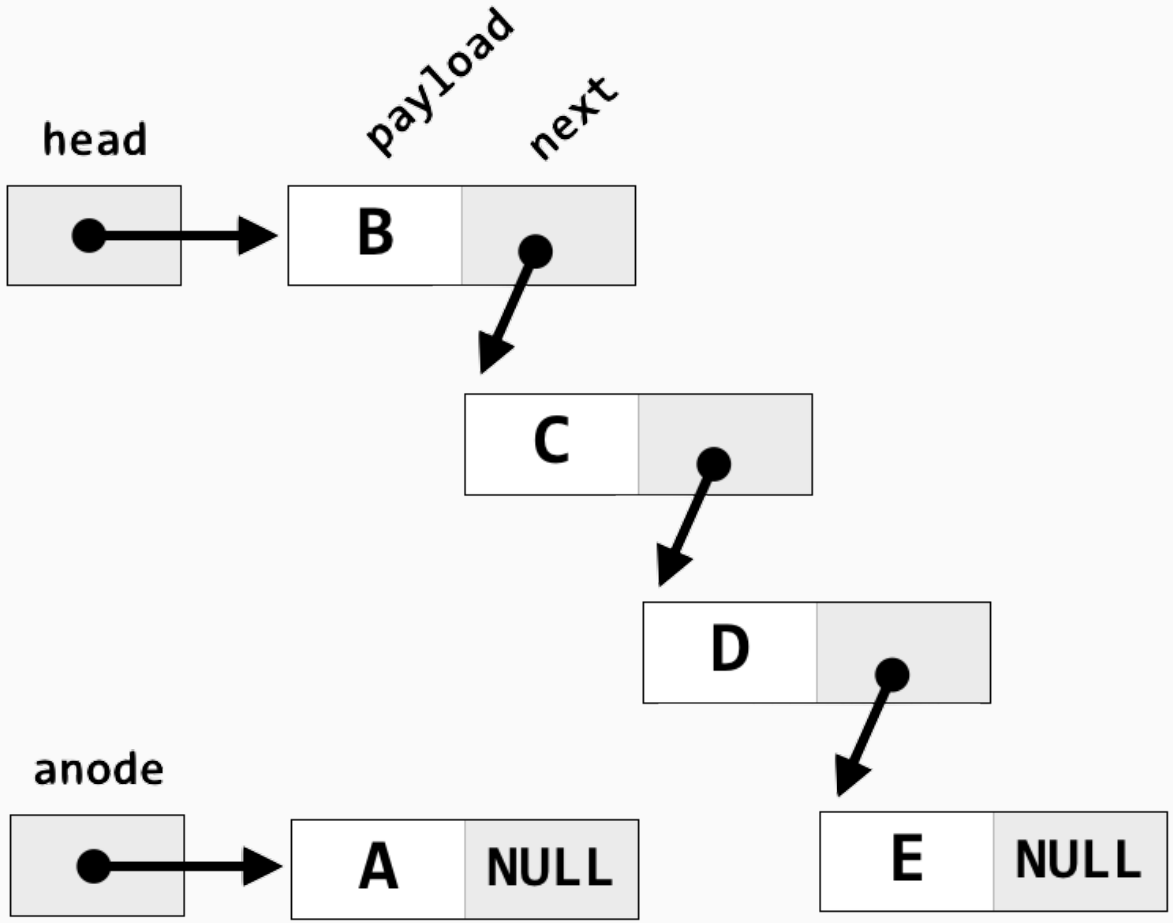
Special cases:

- Inserting at the end (new node's `->next` gets `NULL`)
- Inserting at the beginning...

Linked lists: inserting

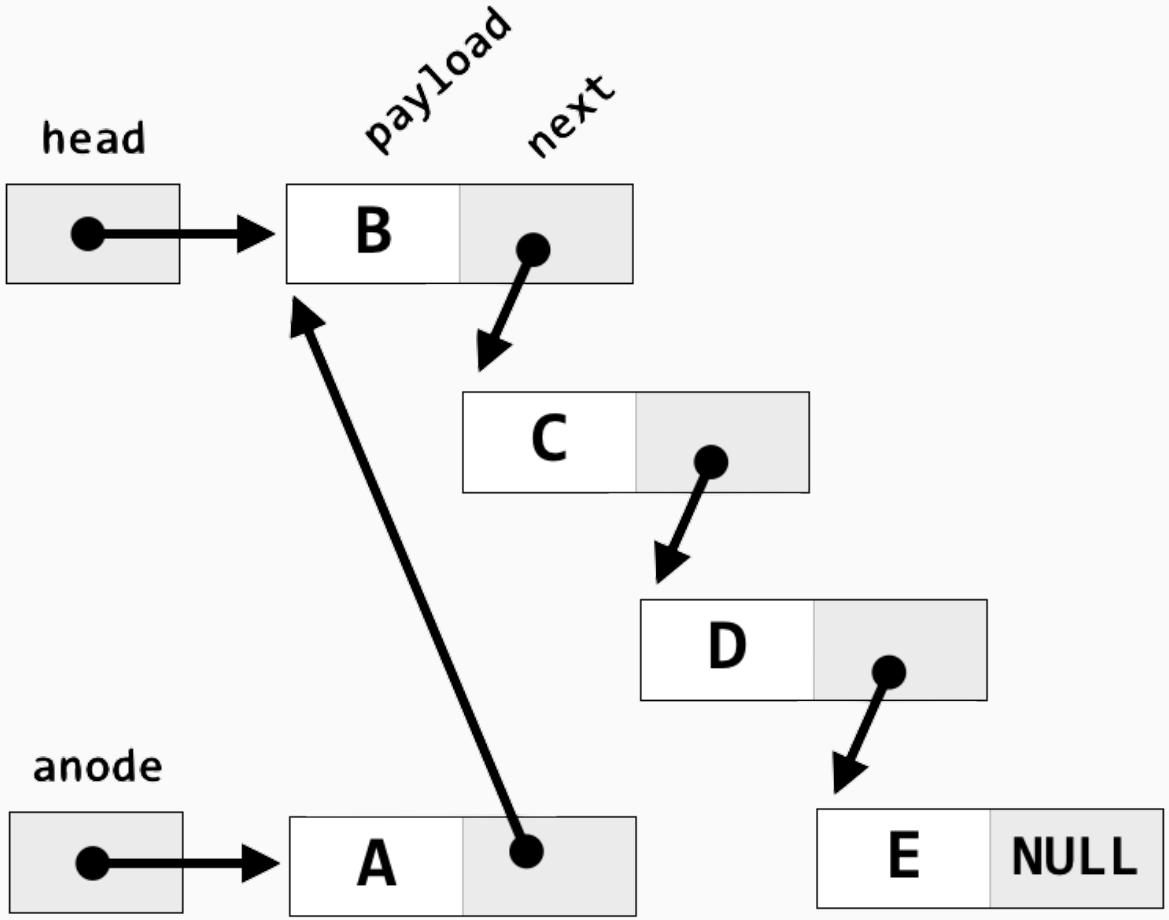


Linked lists: inserting



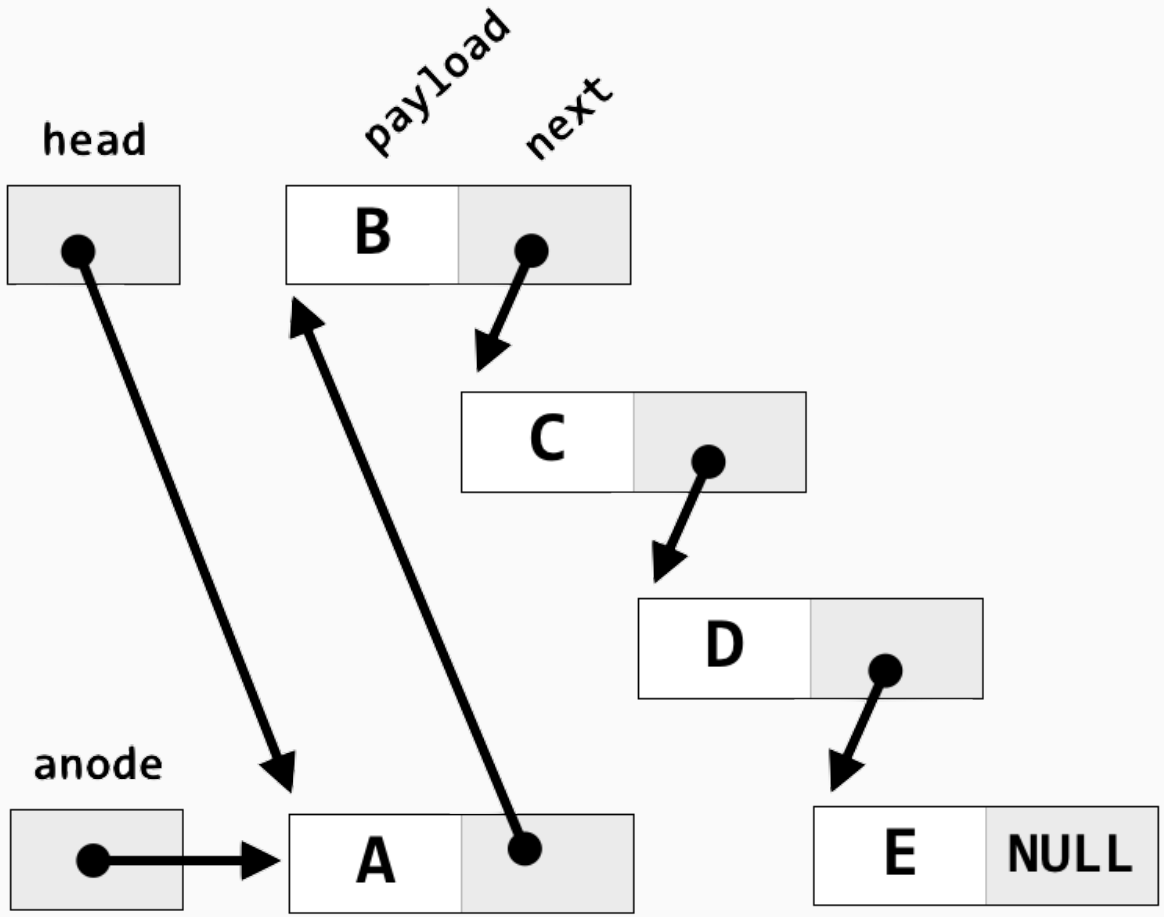

```
struct Node *anode = malloc(sizeof(struct node));
anode->payload = 'A';
anode->next = NULL;
```

Linked lists: inserting



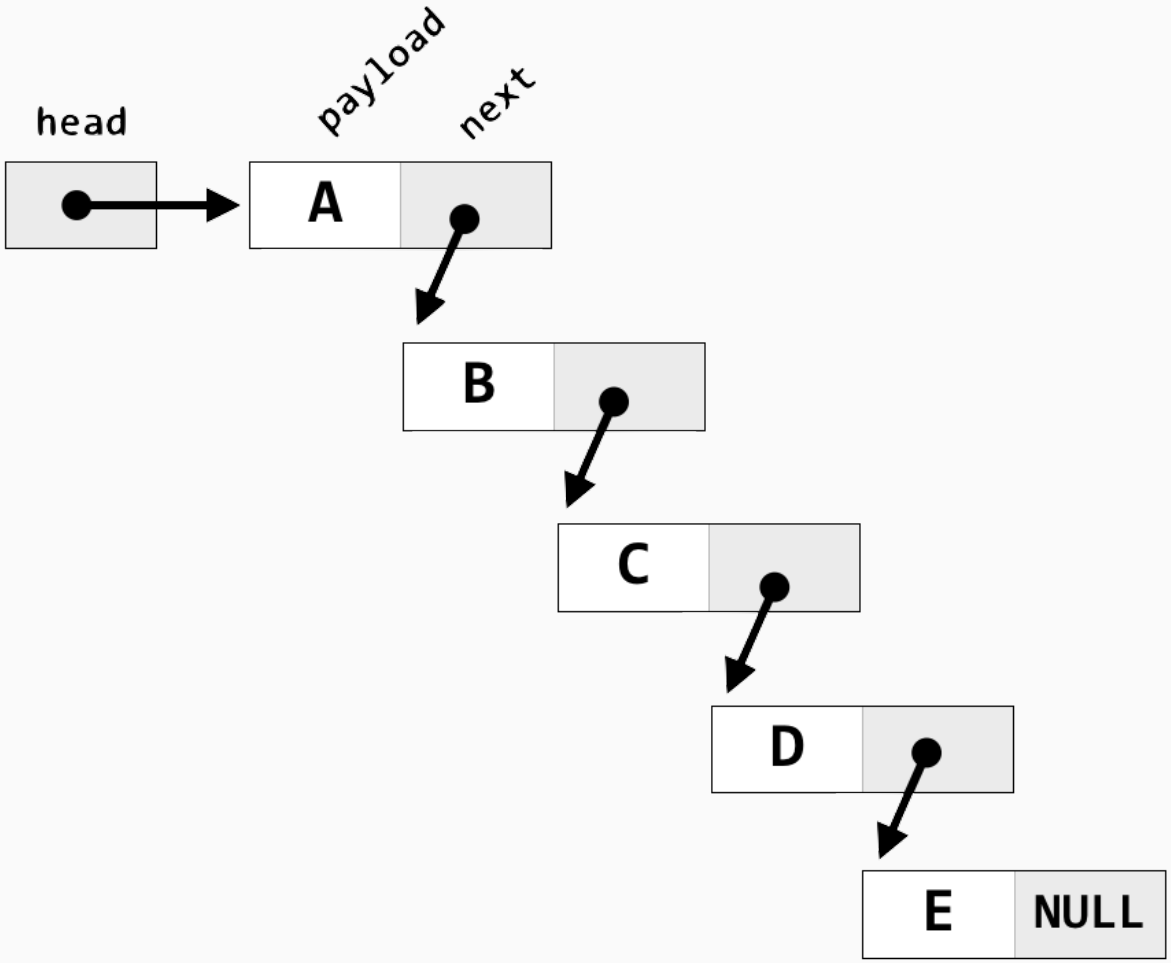

```
anode->next = head;
```

Linked lists: inserting




```
head = anode;
```

Linked lists: inserting



Note this involves changing the head pointer. If the caller or any other function maintains a head pointer, they need to update it!

Linked lists: traversing

We already saw a function that *traverses* the list nodes in order, printing each payload as it goes:

```
void print_list(Node *cur) {
    while(cur != NULL) {
        printf("%c ", cur->payload);
        cur = cur->next;
    }
    putchar('\n');
}
```

Linked lists: traversing

Such functions tend to follow this form:

```
??? do_something(Node *cur) {
    ???
    while(cur != NULL) {
        ???
        cur = cur->next;
    }
    return ???;
}
```

Linked lists: traversing

Complete this function

```
int list_length(Node *cur) {
    ???
    while(cur != NULL) {
        ???
        cur = cur->next;
    }
    return ???;
}
```

Linked lists: traversing

Complete this function

```
int list_length(Node *cur) {
    int length = 0;
    while(cur != NULL) {
        length++;
        cur = cur->next;
    }
    return length;
}
```

Linked lists: traversing

Can we complete this function?

```
void free_list(Node *cur) {
    while(cur != NULL) {
        ???
    }
}
```

Note: to free a list, it's not enough to free the head. That "leaks" the non-head nodes.

Linked lists: traversing

Careful: we have to avoid freeing a `Node` *before* we've followed its `next` pointer

We can't dereference already-freed memory

Linked lists: traversing

```
void free_list(Node *cur) {
    while(cur != NULL) {
        Node *next = cur->next; // *first* get `next`
        free(cur);              // *then* free `cur`
        cur = next;
    }
}
```