Linked List Operations

CMPT 125
Mo Chen
SFU Computing Science
14/2/2020

Lecture 18

Today

- Linkable Nodes
- `LLcreate(...)`
- `LLappend(...)`
- `LLprint(...)`
- `LLsearch(...)`

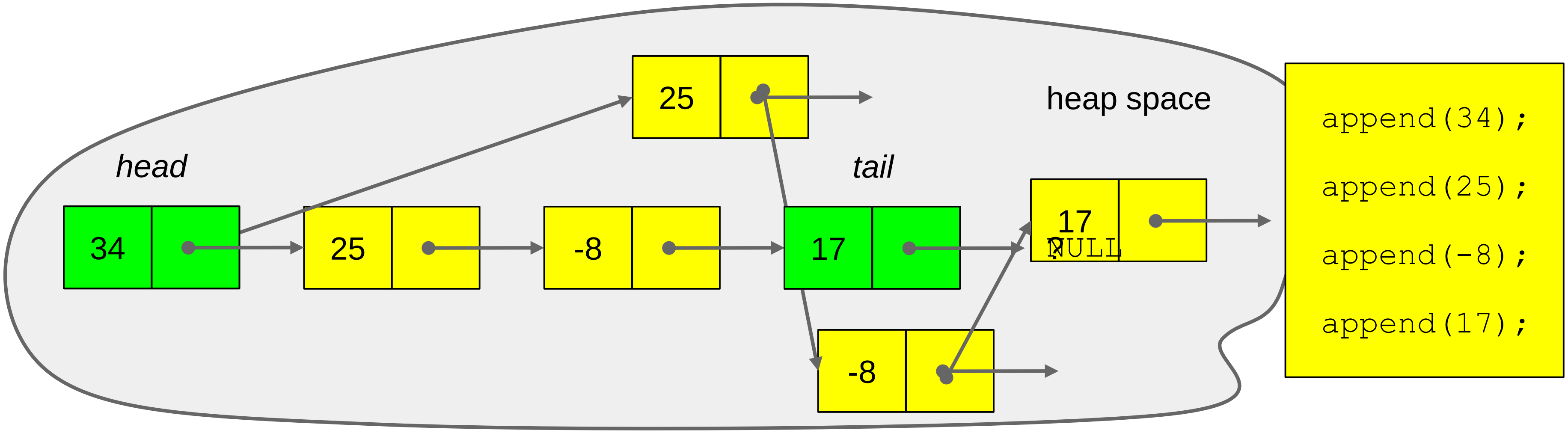
Linked Lists (Review)

On each append, malloc one new element

- keep a pointer to find the next element in the sequence

Coding Idea: parcel the element with the pointer

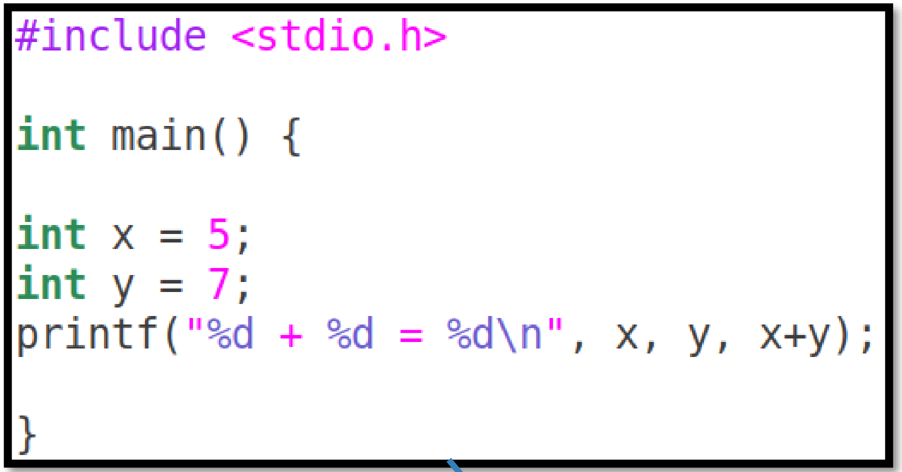
- use a `struct` for convenience
- called a *node*



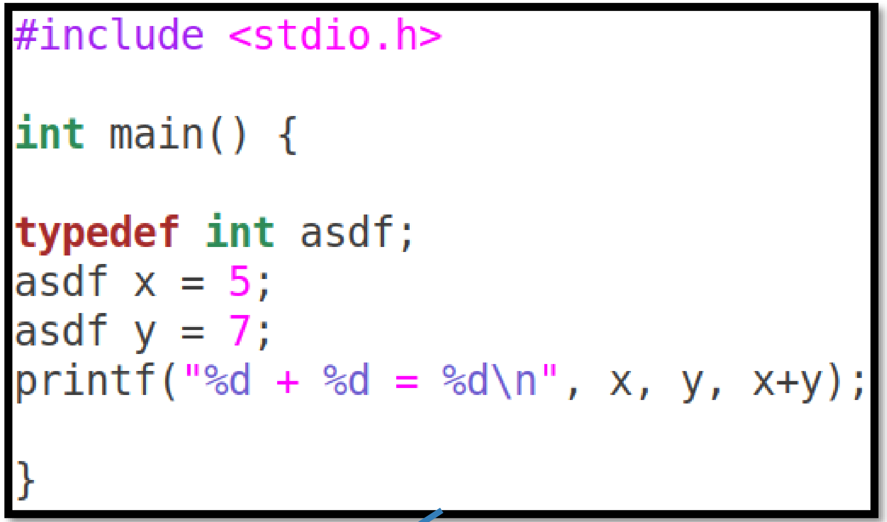
```
append(34);
append(25);
append(-8);
append(17);
```

typedef

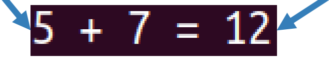
- Rename variable types



```
#include <stdio.h>

int main() {

int x = 5;
int y = 7;
printf("%d + %d = %d\n", x, y, x+y);

}
```



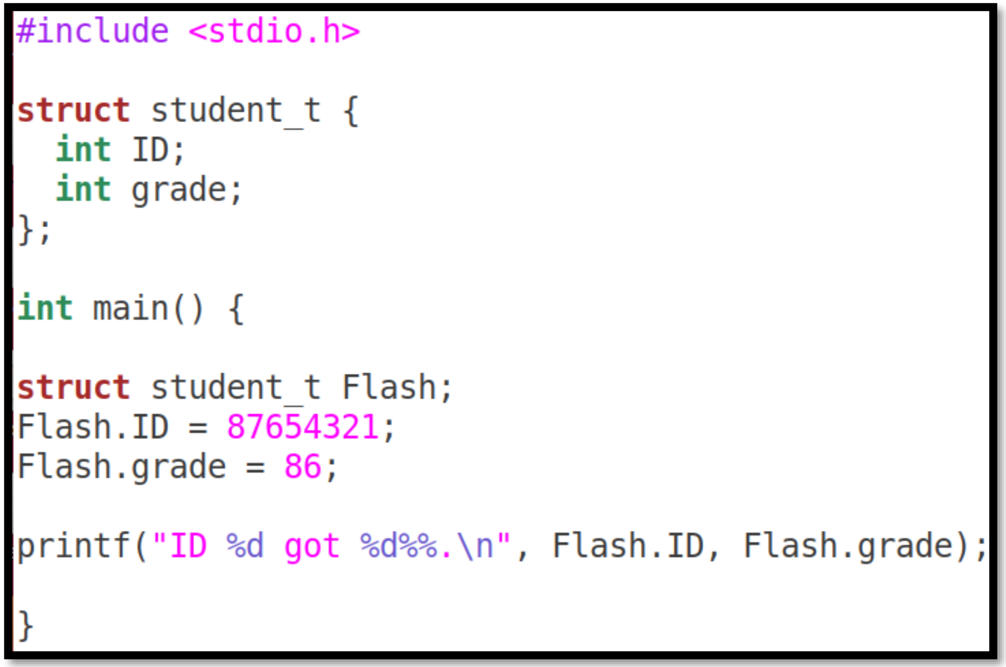
```
#include <stdio.h>

int main() {

typedef int asdf;
asdf x = 5;
asdf y = 7;
printf("%d + %d = %d\n", x, y, x+y);

}
```



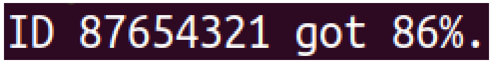
```
5 + 7 = 12
```

struct

- Structure: custom data types that contain other data
  - Can hold any data type, include pointers and other structures



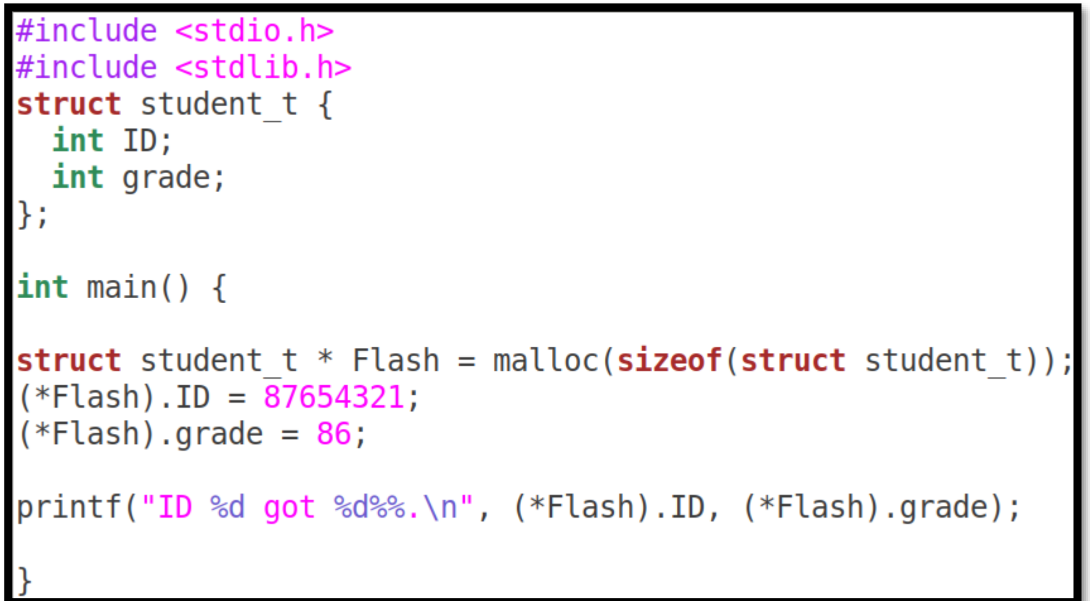
```c
#include <stdio.h>

struct student_t {
  int ID;
  int grade;
};

int main() {

struct student_t Flash;
Flash.ID = 87654321;
Flash.grade = 86;

printf("ID %d got %d%%.\n", Flash.ID, Flash.grade);

}
```



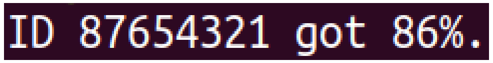
```
ID 87654321 got 86%.
```

struct

- Structure: custom data types that contain other data
  - Can hold any data type, include pointers and other structures



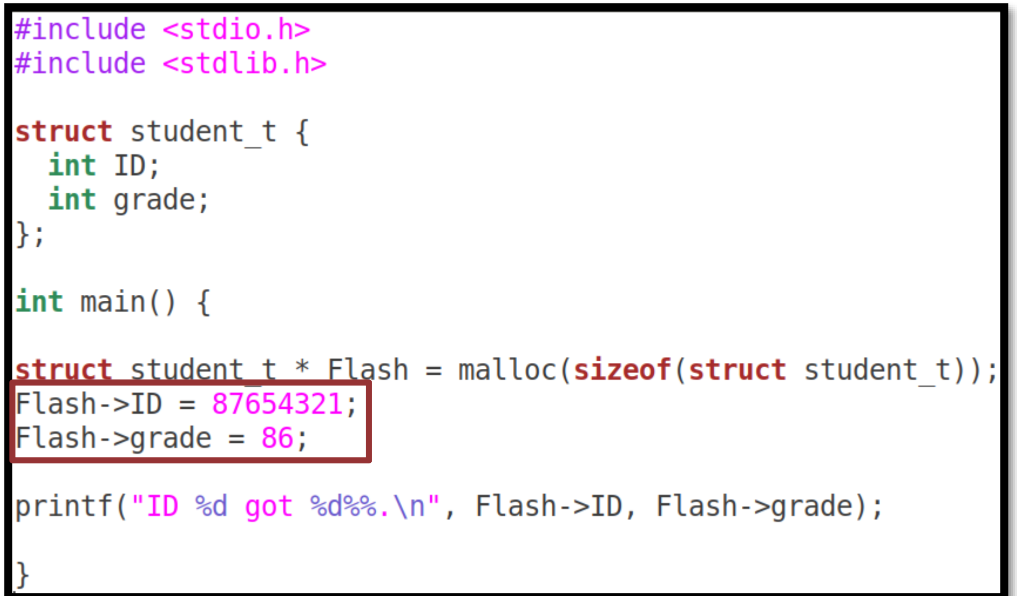
```
#include <stdio.h>
#include <stdlib.h>
struct student_t {
  int ID;
  int grade;
};

int main() {

struct student_t * Flash = malloc(sizeof(struct student_t));
(*Flash).ID = 87654321;
(*Flash).grade = 86;

printf("ID %d got %d%%.\n", (*Flash).ID, (*Flash).grade);

}
```



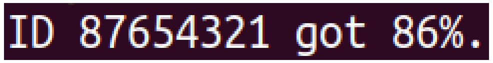
```
ID 87654321 got 86%.
```

struct

- Structure: custom data types that contain other data
    - Can hold any data type, include pointers and other structures



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

struct student_t {
  int ID;
  int grade;
};

int main() {

struct student_t * Flash = malloc(sizeof(struct student_t));
Flash->ID = 87654321;
Flash->grade = 86;

printf("ID %d got %d%%.\n", Flash->ID, Flash->grade);

}
```



```
ID 87654321 got 86%.
```

Linkable Nodes

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

struct node_t x1, x2;
```

Linkable Nodes

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

struct node_t x1, x2;
```

```
typedef struct {
      int data;
      node_t * next;
} node_t;

node_t x1, x2;
```

Can declare a pointer within a struct of the same type

- but would prefer `node_t x1, x2;` over `struct node_t x1, x2;`
- try `typedef`

Linkable Nodes

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

struct node_t x1, x2;
```

✗

```
typedef struct {
       int data;
       node_t * next;
} node_t;


node_t x1, x2;
```

Can declare a pointer within a struct of the same type

- but would prefer `node_t x1, x2;` over `struct node_t x1, x2;`
- try `typedef`

Forward reference is no good

- a *prototype* is required

Linkable Nodes

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

struct node_t x1, x2;
```

✗

```
typedef struct {
        int data;
        node_t * next;
} node_t;

node_t x1, x2;
```

Can declare a pointer within a struct of the same type

- but would prefer `node_t x1, x2;` over `struct node_t x1, x2;`
- try `typedef`

Forward reference is no good

- a *prototype* is required

```
typedef struct _node {
        int data;
        struct _node * next;
} node_t;

node_t x1, x2;
```

Linkable Nodes

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

struct node_t x1, x2;
```

Can declare a pointer within a struct of the same type

- but would prefer `node_t x1, x2;` over `struct node_t x1, x2;`
- try `typedef`

Forward reference is no good

- a *prototype* is required

✗

```
typedef struct {
       int data;
       node_t * next;
} node_t;

node_t x1, x2;
```

✓

```
typedef struct _node {
       int data;
       struct _node * next;
} node_t;

node_t x1, x2;
```

Linkable Nodes

✓

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

typedef struct node_t node_t;
node_t x1, x2;
```

✗

```
typedef struct {
       int data;
     node_t * next;
} node_t;


node_t x1, x2;
```

Can declare a pointer within a struct of the same type

- but would prefer `node_t x1, x2;` over `struct node_t x1, x2;`
- try `typedef`

Forward reference is no good

- a *prototype* is required

✓

```
typedef struct _node {
       int data;
     struct _node * next;
} node_t;


node_t x1, x2;
```

Node structure and typedef

- Node used in linked lists

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

Declaring a node_t: `struct node_t node1;`

- Use typedef reduce annoyance

```
typedef struct node_t node_t;
```

Declaring a node_t: `node_t node1;`

- “Shortcut”

```
typedef struct _node {
    int data;
    struct _node * next;
} node_t;
```

Declaring a node_t: `node_t node1;`

Building a Linked List

Strategy: Maintain a pointer to the head element and a pointer to the tail.

- Q. What types are these?
- Q. When declared, with what values are `head`, `tail` initialized?

A linked list can be uniquely specified by its head pointer.

- keep tail pointer around for convenience

Building The Interface

Put all declarations in the header file

- `typedef LL_t`
- function prototypes

Put implementation in a corresponding .c file

- keep details hidden from other programs

Q. What sort of operations would you perform on a list?

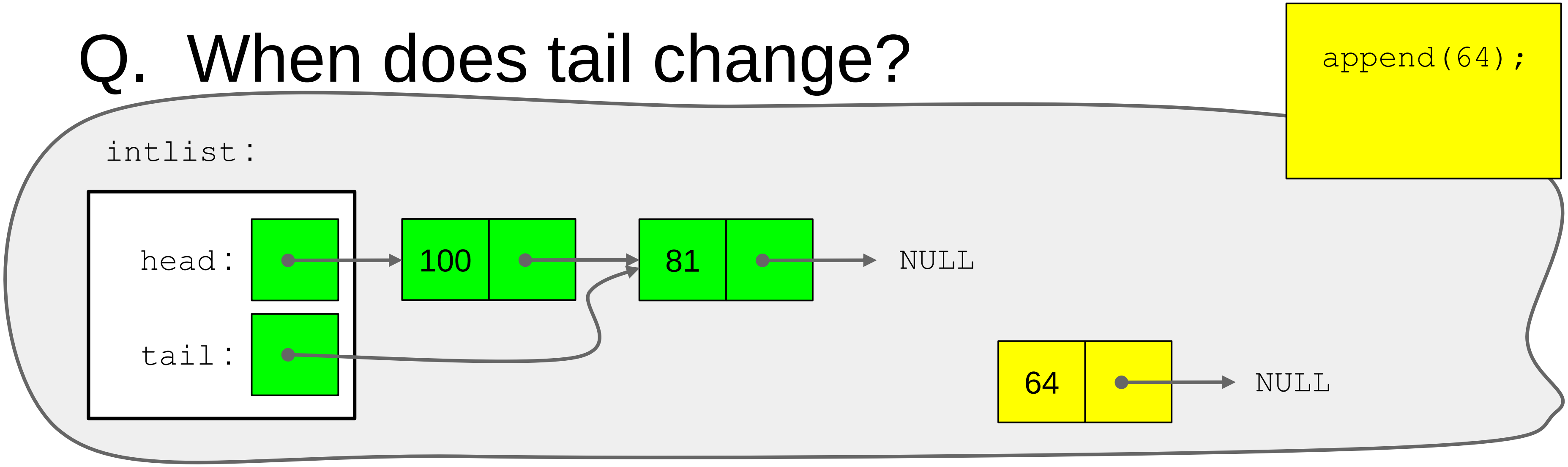
Linked List: `append(x)`

Two big steps:

- allocate new node
- maintain head, tail

Q. When does head change?

Q. When does tail change?



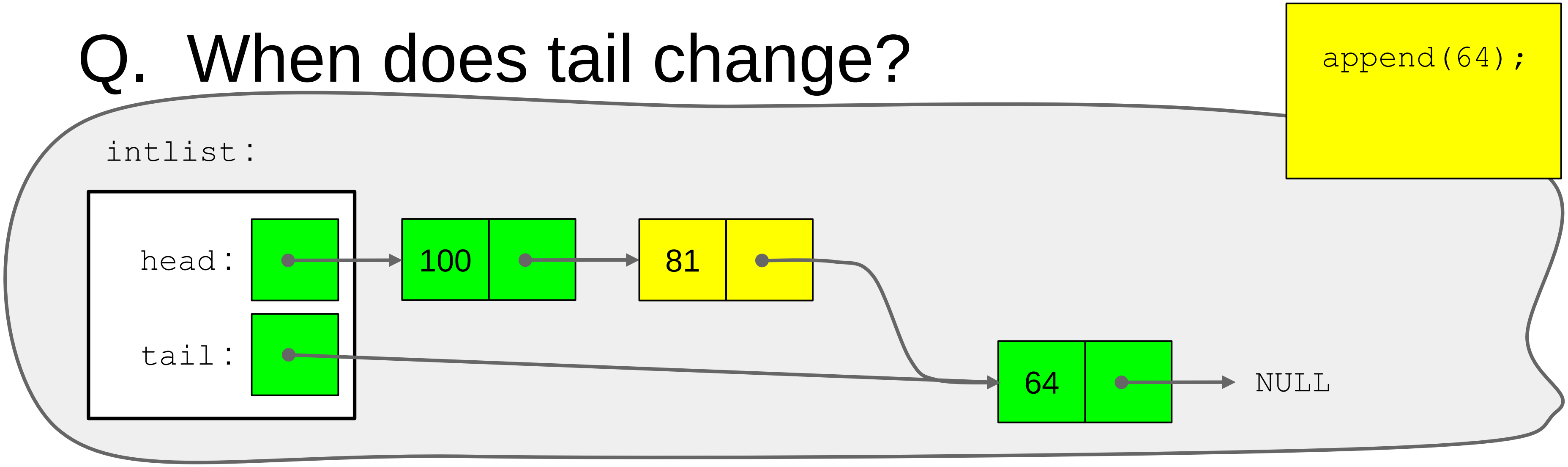
Linked List: **`append(x)`**

Two big steps:

- allocate new node
- maintain head, tail

Q. When does head change?

Q. When does tail change?



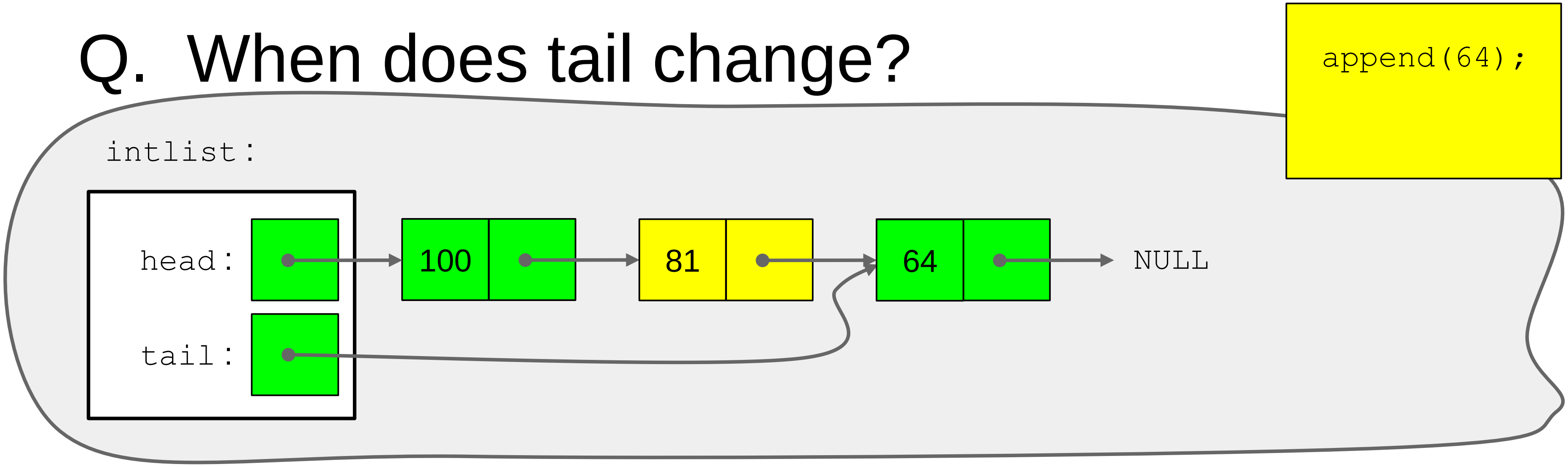
Linked List: `append(x)`

Two big steps:

- allocate new node
- maintain head, tail

Q. When does head change?

Q. When does tail change?



Linked List: `append(x)`

Two big steps:

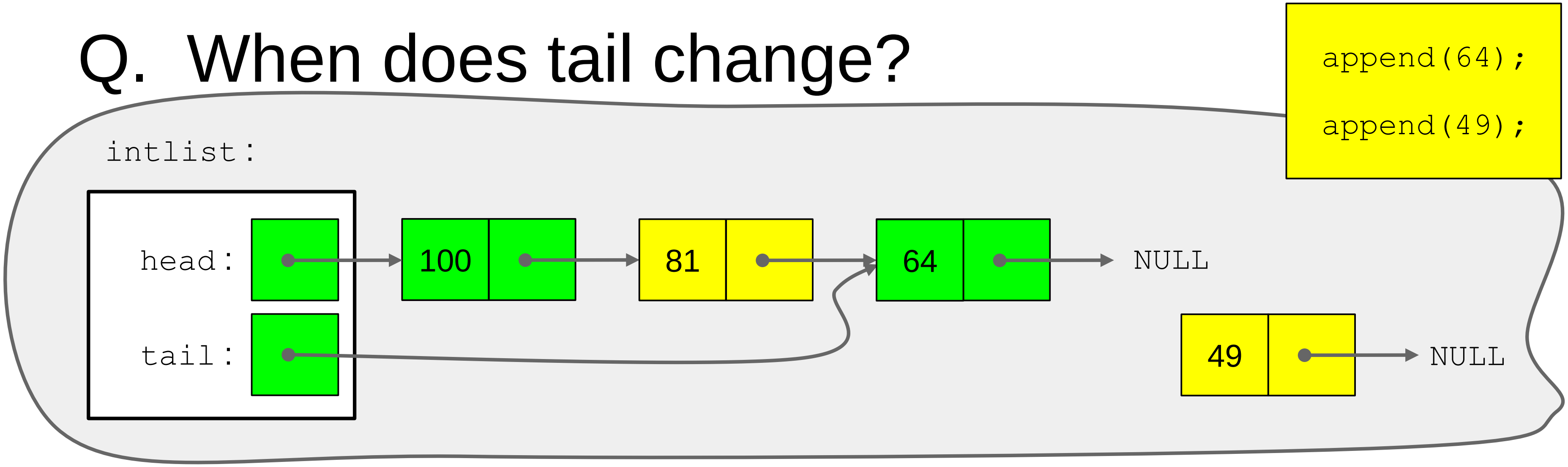
- allocate new node
- maintain head, tail

All the steps:

- malloc a new `node_t`
- fill in the fields of the new node
- `tail->next = newNode;`
- `tail = newNode;`

Q. When does head change?

Q. When does tail change?



Linked List: `append(x)`

Two big steps:

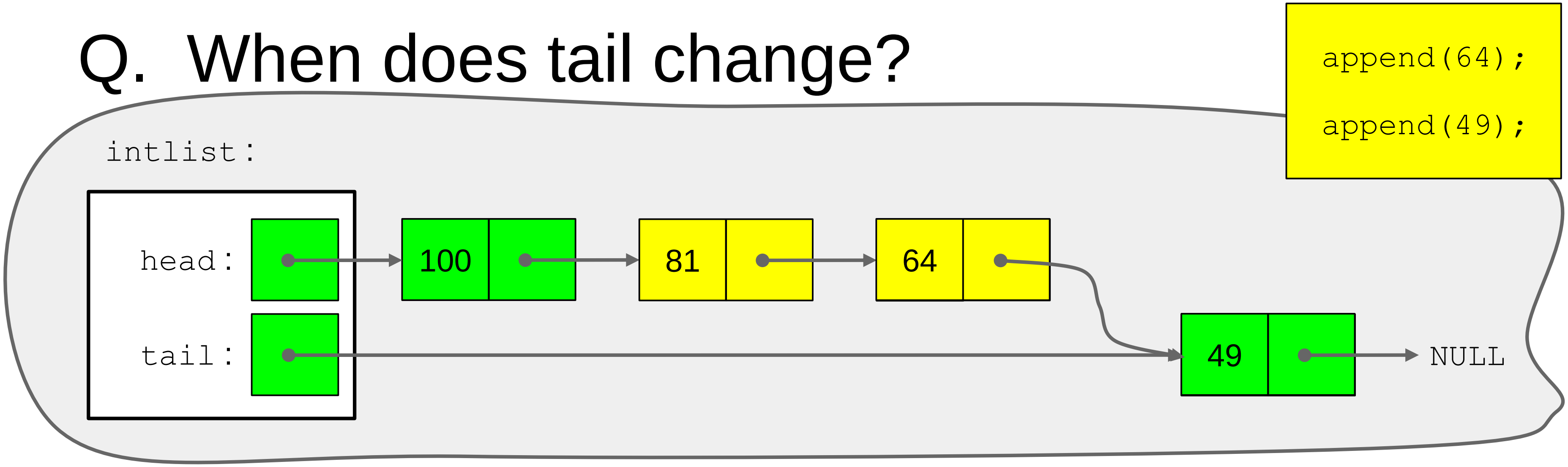
- allocate new node
- maintain head, tail

All the steps:

- malloc a new `node_t`
- fill in the fields of the new node
- `tail->next = newNode;`
- `tail = newNode;`

Q. When does head change?

Q. When does tail change?



```
append(64);
append(49);
```

Linked List: `append(x)`

Two big steps:

- allocate new node
- maintain head, tail

All the steps:

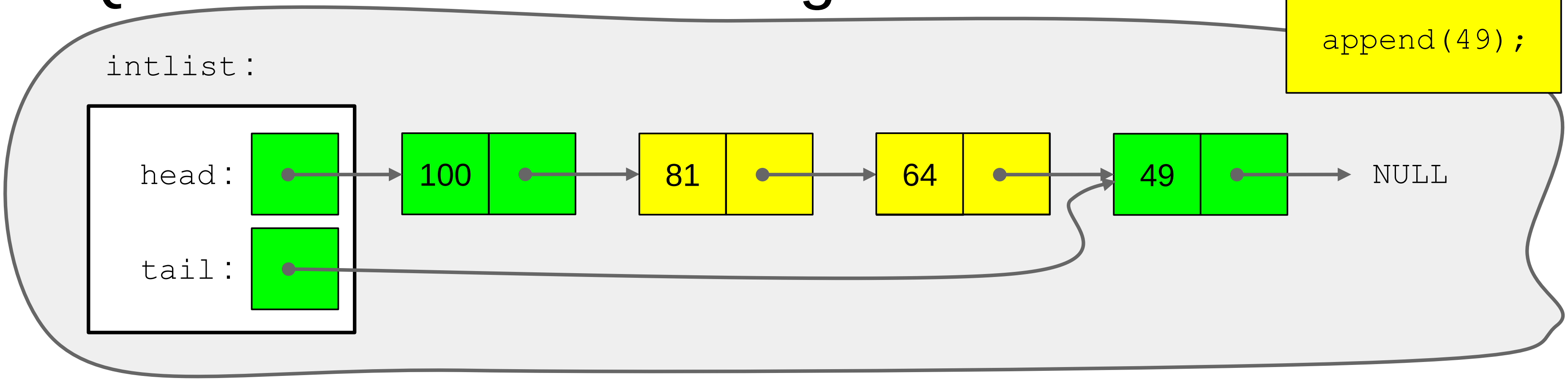
- malloc a new `node_t`
- fill in the fields of the new node
- `tail->next = newNode;`
- `tail = newNode;`

But why does it seg fault?

Q. When does head change?

Q. When does tail change?

```
append(64);

append(49);
```



Linked List: `append(x)`

Two big steps:

- allocate new node
- maintain head, tail

All the steps:

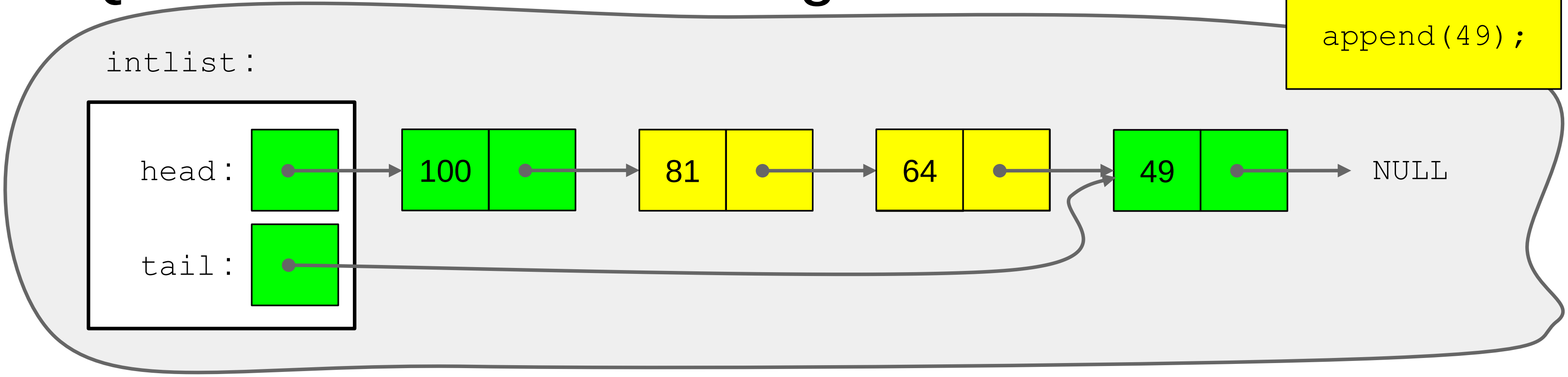
- malloc a new `node_t`
- fill in the fields of the new node
- `tail->next = newNode;`
- `tail = newNode;`

But why does it seg fault?

Q. When does head change?

Q. When does tail change?

```
append(64);

append(49);
```



Linked List: `append(x)`

Two big steps:

- allocate new node
- maintain head, tail

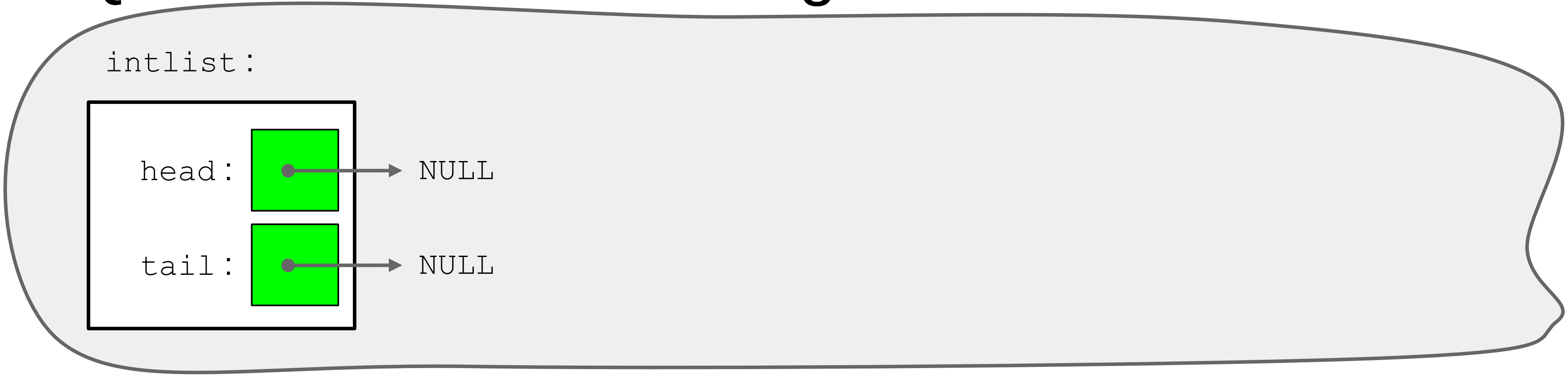
Q. When does head change?

Q. When does tail change?

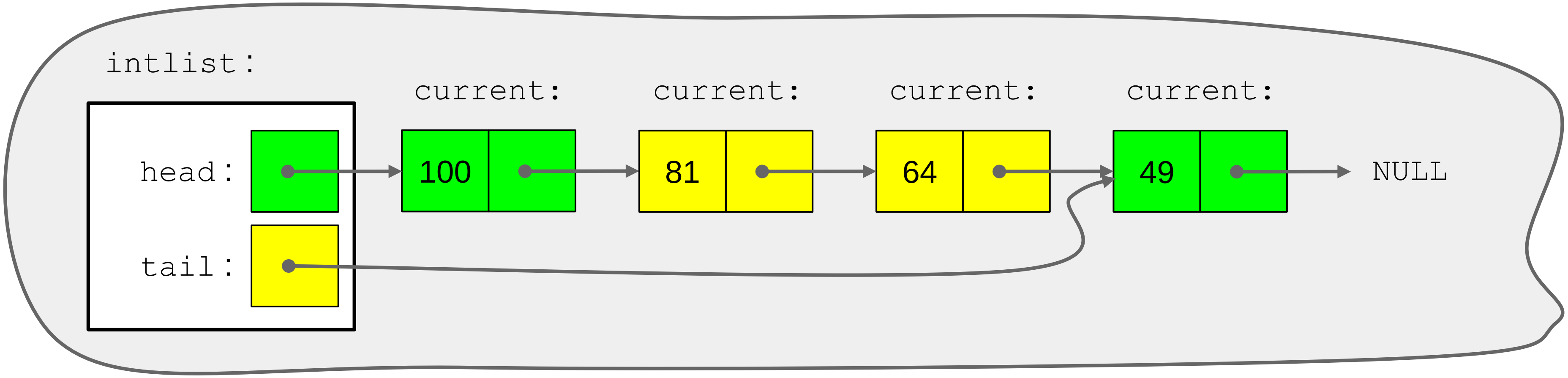
All the steps:

- malloc a new `node_t`
- fill in the fields of the new node
- `tail->next = newNode;`
- `tail = newNode;`

Appending to the empty list is a corner case that must be handled separately.



Linked List: `print()`



Expected output:

```
100 81 64 49
```

Output:

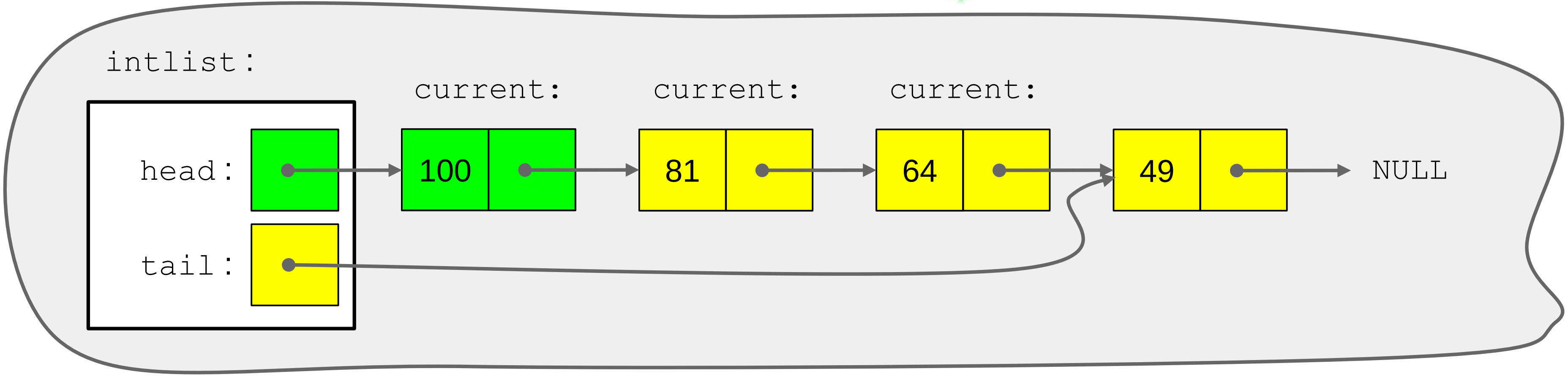
```
100 81 64 49
```

Strategy: Dereference all pointers in sequence starting with head.

- then `head->next`
- then `head->next->next`, etc.
- stop when `NULL` is reached

```
curr = head
while(curr != NULL) {
  print curr->data
  curr = curr->next
}
```

Linked List: `search(target)` ✓



`search(64)` returns `1`

`search(58)` returns `0`

Q. What's the strategy this time?

- similar to `print()`
- instead of print, `return 1` if found

```
curr = head
while(curr != NULL) {
  if equal then
    return 1
  curr = curr->next
}
return 0
```