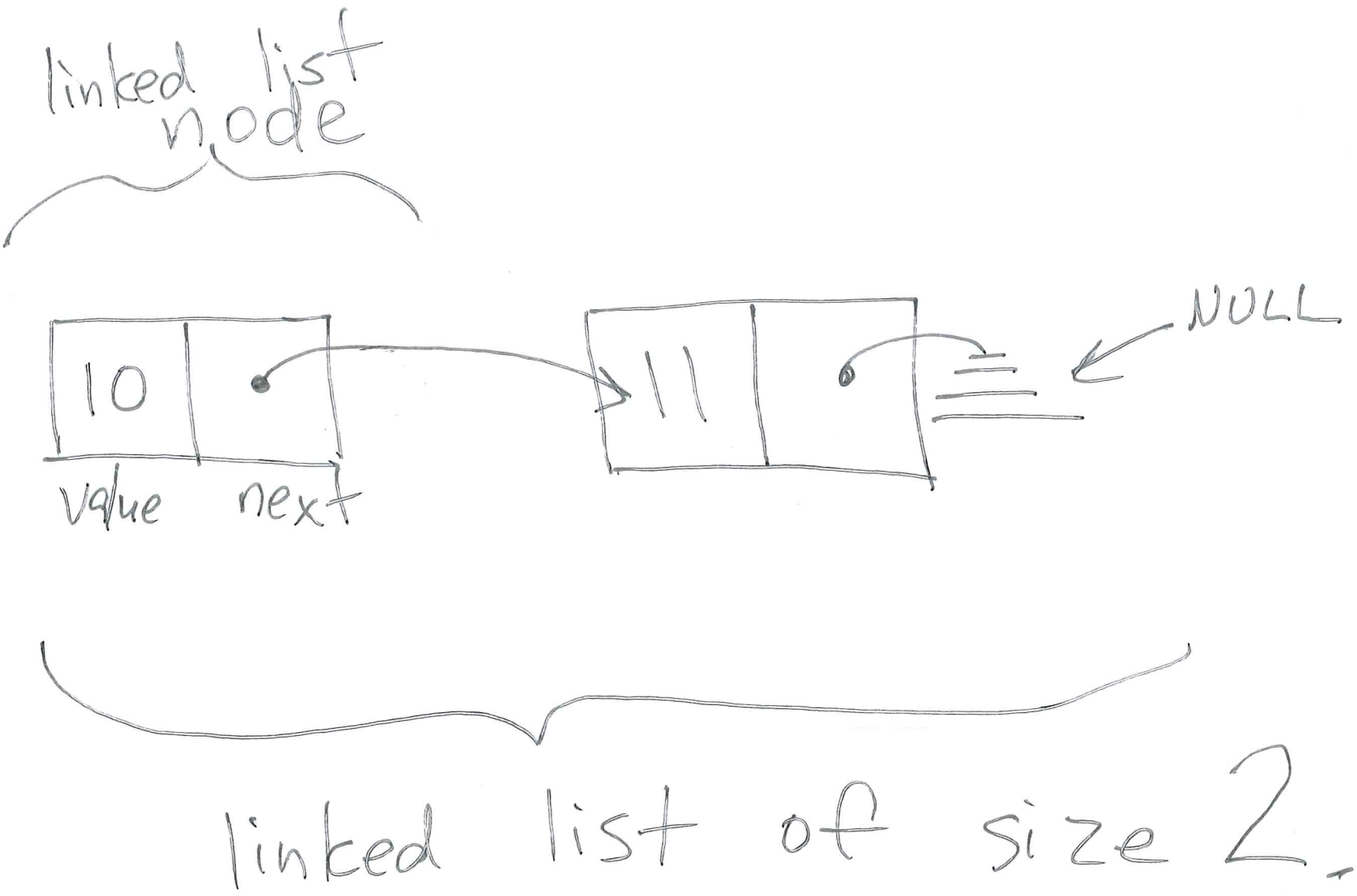
Tue

2-19-2019

Some of this is from Thu 2/14/2019 (last week)

Linked lists

- struct syntax review
- struct address syntax (->)


linked list node
10
11
NULL
value
next
linked list of size 2.

Linked list	Array
Container	Container
almost always on heap (in practice)	stack or heap (or ....)
Can insert or delete elements	fixed size
all elements same type	all elements same type
sequential access only	Random access es. array [idx]

(this page from Fall 2017)

Struct objects (using typedef syntax for here)

Declare + initialize

```
Point p = {
    .x = 5,
    .y = 6
};
```

Access field

```
int w = p.x;
int q = p.y;
```

Assign struct object

```
Point p2 = p1;
```

Assign to field

```
p.x = 5;  // assign 5 to the 'x' field of p
p.y = 6;  //  "  6  "  "  "  y  "  "
```

All of this is on the reference sheet.

Struct address syntax

a->p->x is the same as (*a.p).x

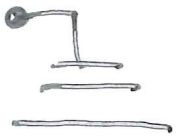
(&p)->x is the same as p.x

Linked list operations

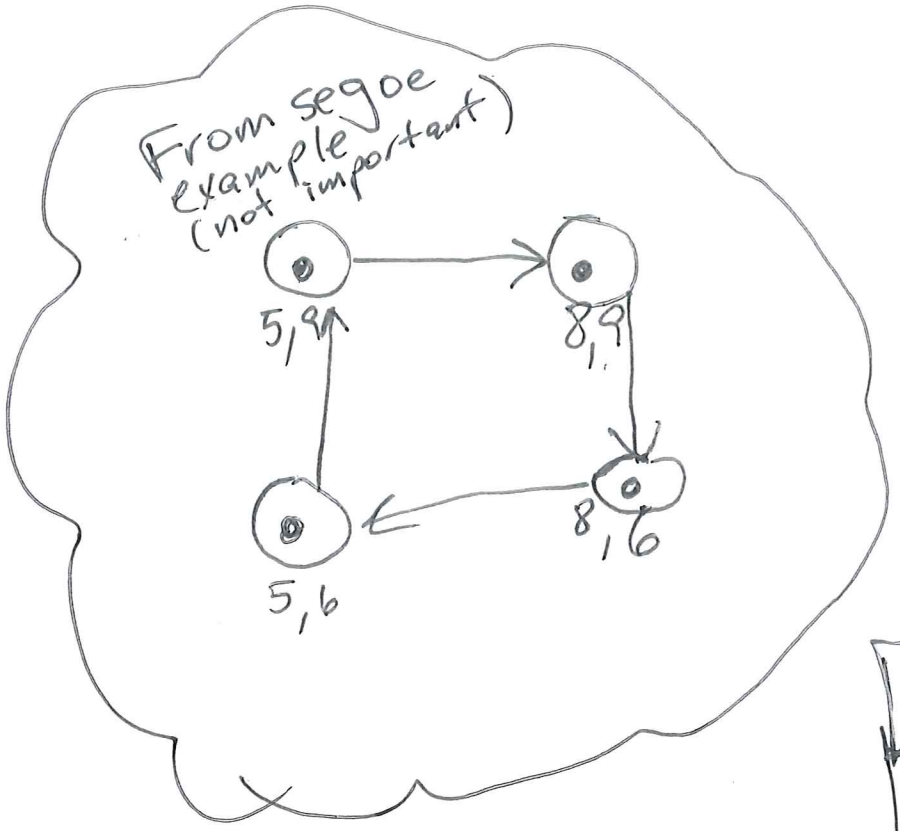
create_node	(in isolation)
append	to end
insert	
destroy	

Empty linked list:

NULL



NULL

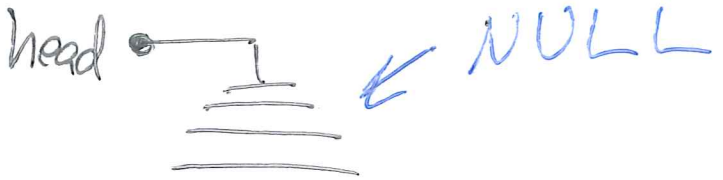


NULL is a constant that we use in place of a memory address.

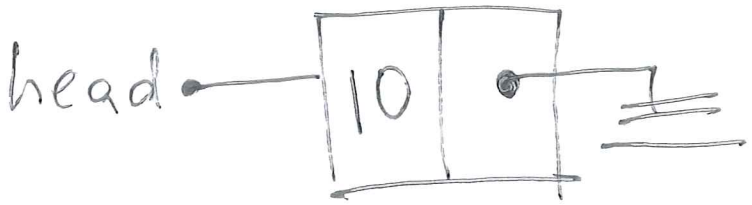
Internally, value of NULL is 0.

append(...)

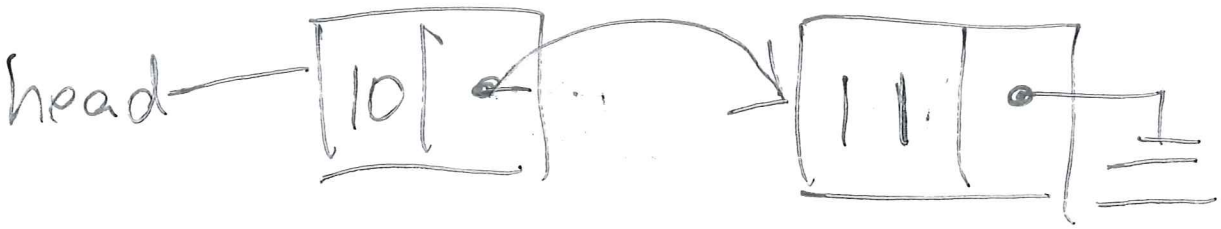
Size 0



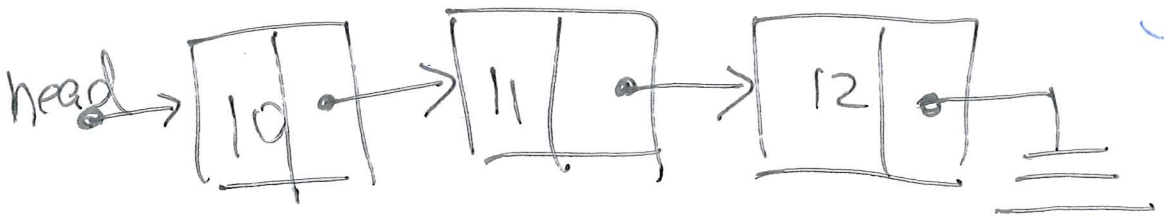
Size 1



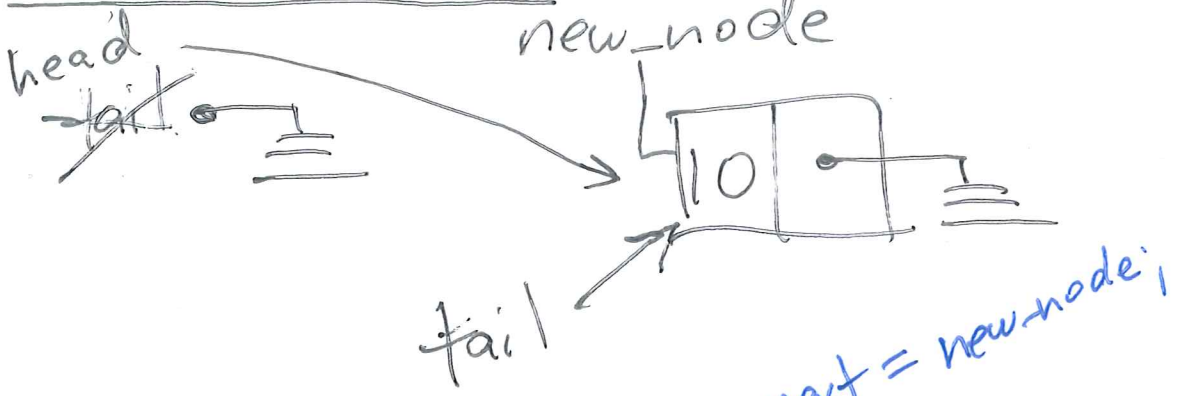
Size 2



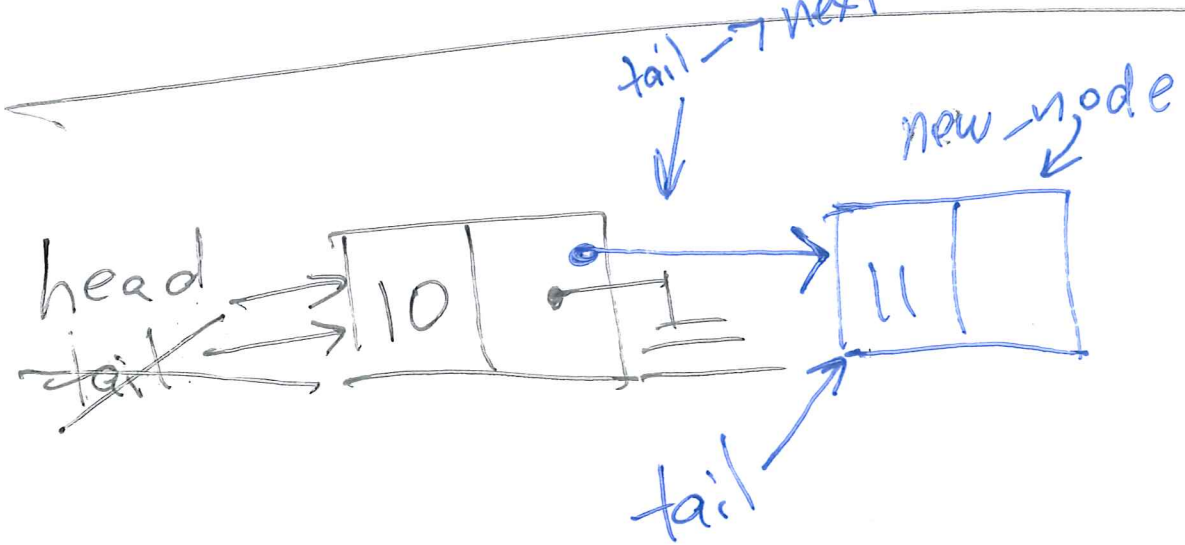
Size 3



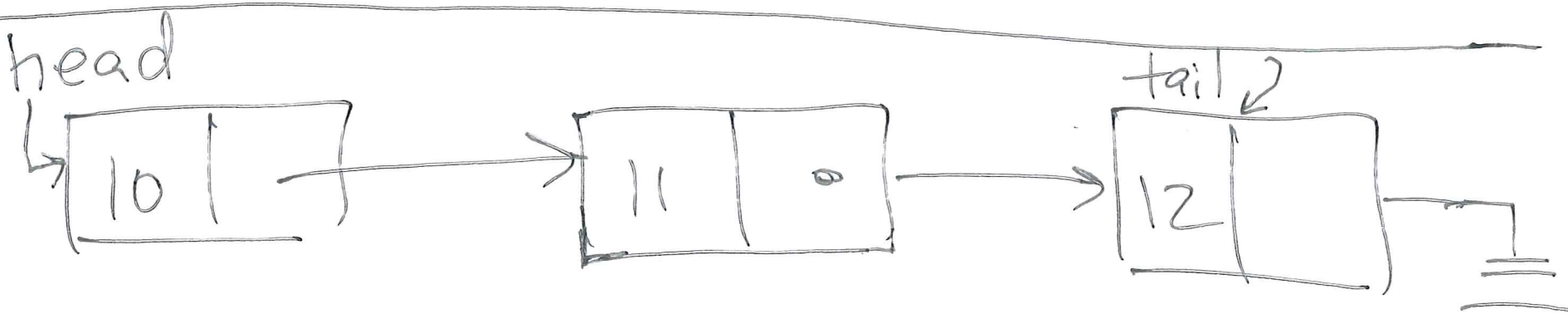
Linked lists

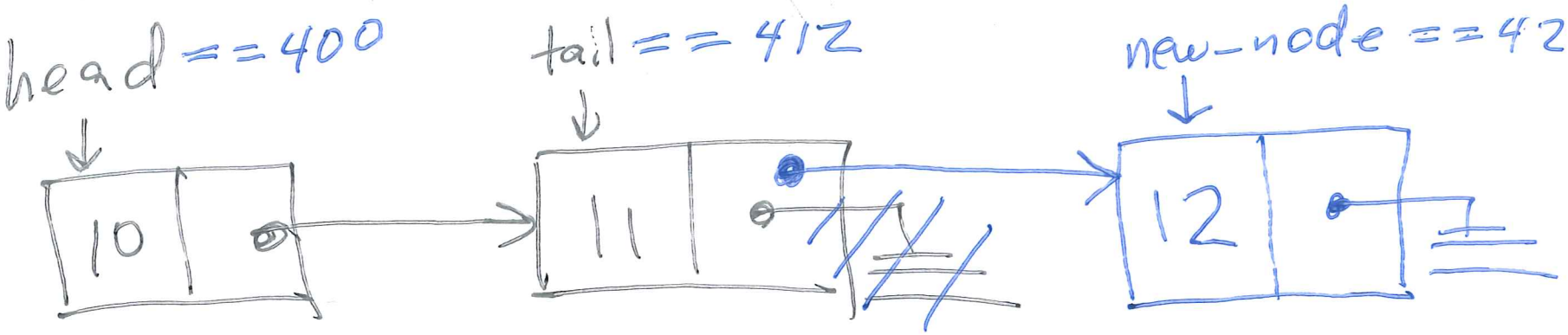


size 0 to 1



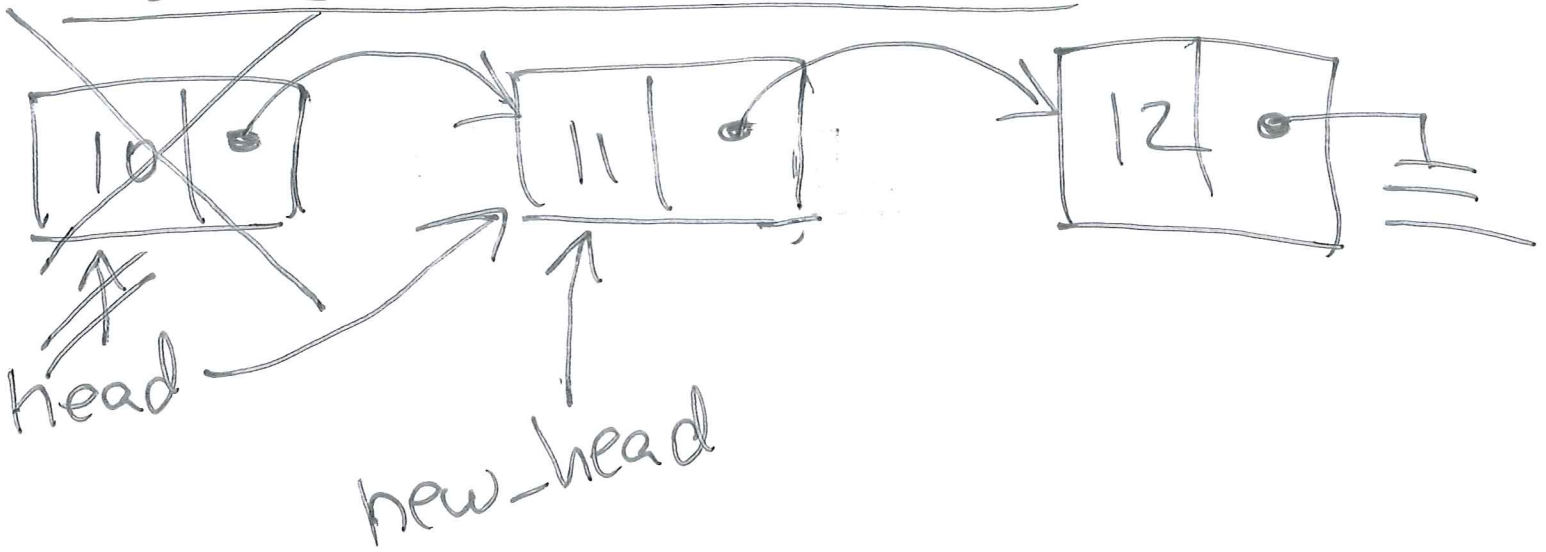
size 1 to 2

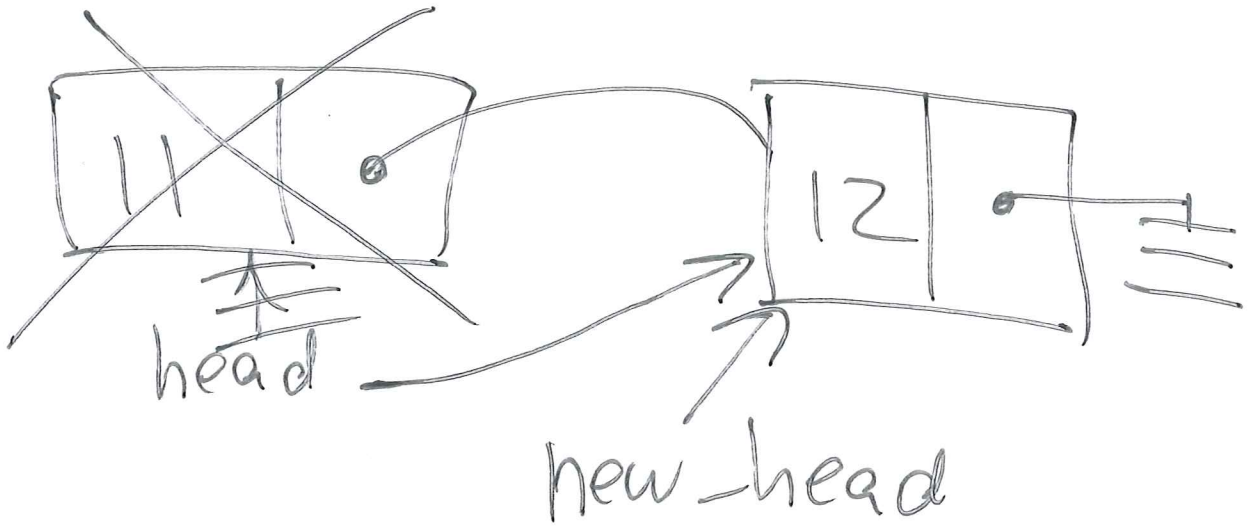


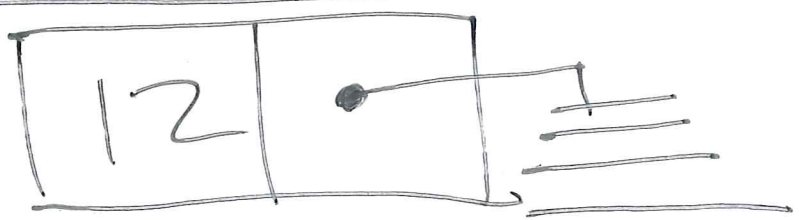


Append to linked list

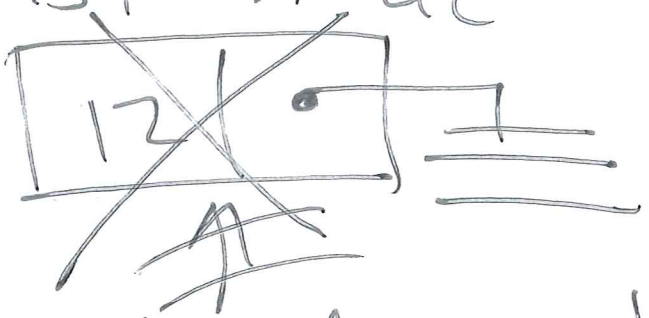
Destroy linked list



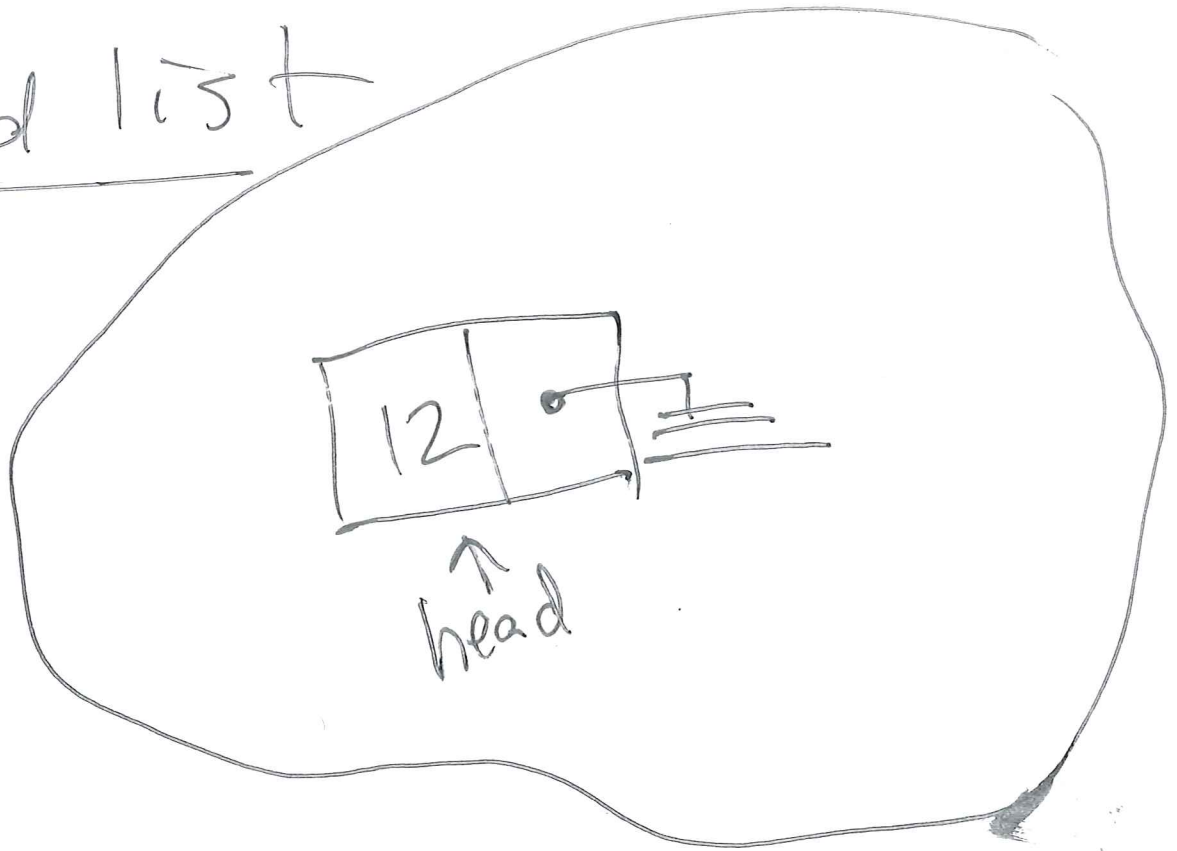




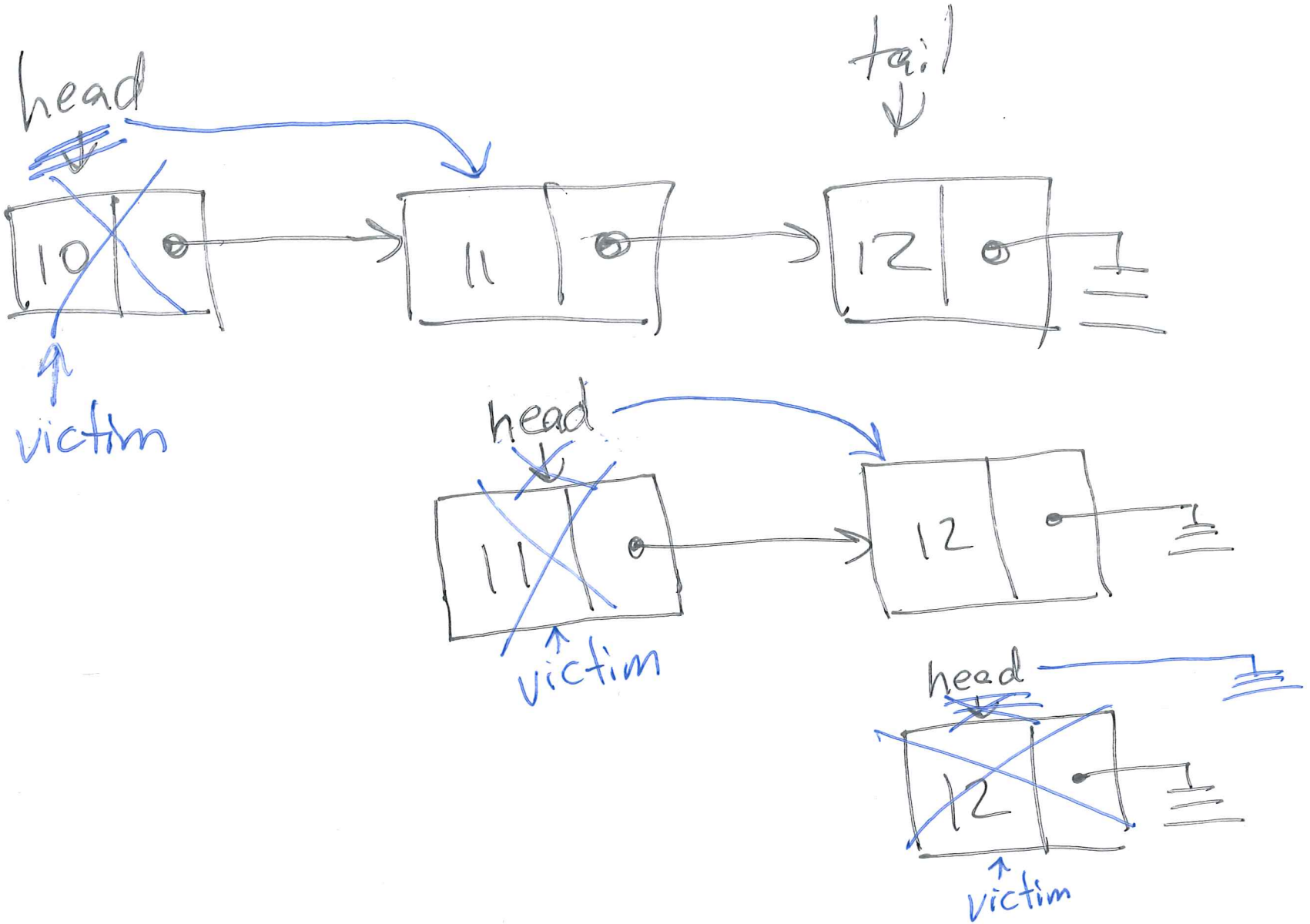
Delete last node



Destroy linked list

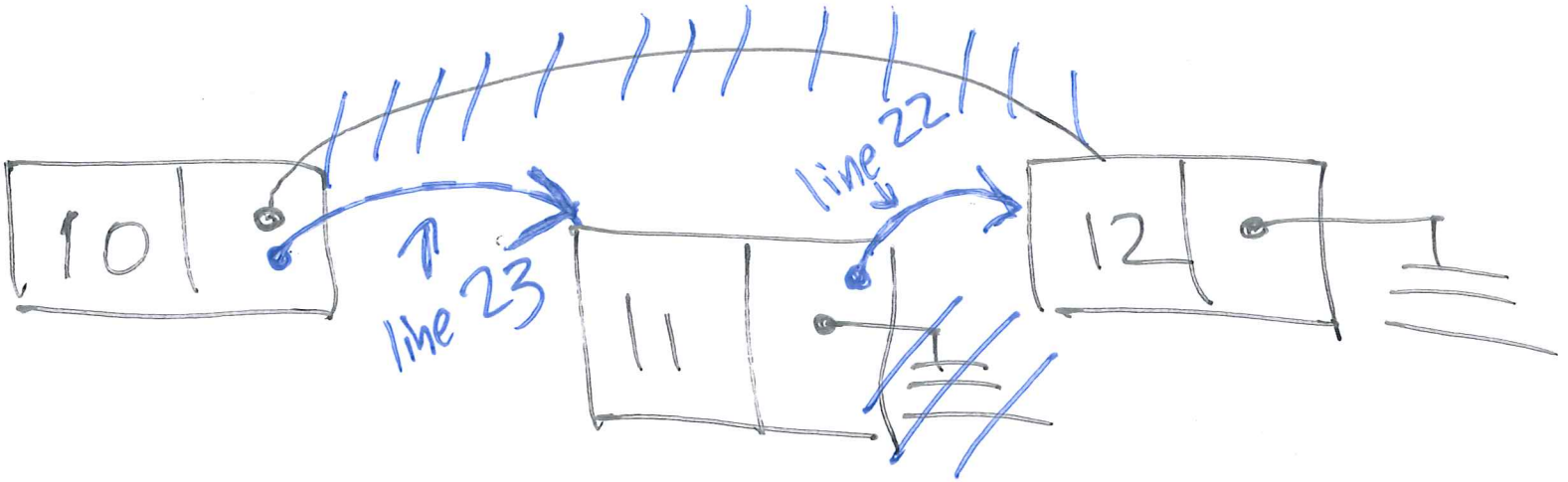


Destroy linked list



Free every block that you malloc.
↓
exactly once

Insert node between two existing nodes



~~Ex~~ From append(..) example

Section 3

Stack

addr	type*	name*	value	part	fn
200	int	argc	1	args	main(...)
204	char**	argv	→ {"./foo"}		
212	void*			ret addr	
220	struct Node*	head	~~NULL~~ 400	locals	
228	struct Node*	tail	~~NULL~~ 400		
~~236~~	int	value	10	args	append(..)
240	struct Node**	a_head	220		
248	struct Node**	a_newtail	228		
~~256~~	struct Node*	new_node	400		
236		value	11	args	
		a_head	220		
		a_tail	228		
		new_node	412	locals	

Heap

addr	value		
400	10	~~NULL~~ 412	
412	11	NULL	
424			

* a_head

Data segment

addr	type*	value
600		

- Type and name are not actually stored in memory or executable. Addresses shown are fictional.
- Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`
- To show struct types with fields, split the type and name fields. In value field, just write the value of the field. Example →

type	name	value
Point : int, : int	p . x . y	5 6

Linked list of size 1

Stack

From append(...) example

addr	type*	name*	value	part	fn
200	int	argc	1	args	main(...)
204	char**	argv	→ {"./foo"}		
212	void*			ret addr	
220	struct Node*	head	~~NULL~~ → 400	locals	
228	struct Node*	tail	~~NULL~~ → 400		
236	int	value	10		append
240	struct Node**	a_head	220		
248	struct Node**	a_tail	228		
256	struct Node*	new_node	400		
264					

append(10, &head, &tail)

Heap

Section 2

addr	value	
400	10 | NULL	
412		

Data segment

addr	type*	value
600		

- Type and name are not actually stored in memory or executable. Addresses shown are fictional.
- Assume `sizeof(int)==4` `sizeof(char)==1` `sizeof(void*)==8`
- To show struct types with fields, split the type and name fields. In value field, just write the value of the field. Example →

type	name	value
Point : int, : int	p . x	5
	. y	6

append(...)

Stack

Heap

addr	type*	name*	value	part	fn
200	int	argc	1	args	main(...)
204	char**	argv	→ {"./foo"}		
212	void*			ret addr	
220	struct Node *	head	400	locals	
228	struct Node *	tail	412		
236	int	value		params	append(...)
240	struct Node **	a_head	220		
	struct Node **	a_tail	228		

addr	value	
400	10 | 412	🔒
412	11 | 424	🔒
424		

append(12, &head, &tail)

Data segment

addr	type*	value
600		

- Type and name are not actually stored in memory or executable. Addresses shown are fictional.
- Assume `sizeof(int)==4` `sizeof(char)==1` `sizeof(void*)==8`
- To show struct types with fields, split the type and name fields. In value field, just write the value of the field. Example →

type	name	value
Point : int,	p . x	5
: int	. y	6