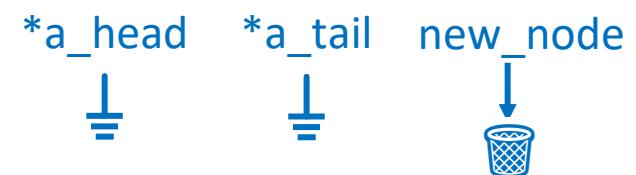


Appending to a linear linked list with the `append(...)` function.

Step 1. `main(...)` calls `append(3, &head, &tail)`. The variables `head` and `tail` (in `main(...)`) are `NULL` because the list is empty.

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4     ►AS OF HERE◄
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node;                            //      Set head to new_node
13    }
14    else {                                              // If list is not empty...
15        (*a_tail) -> next = new_node; //      Connect tail to new_node
16    }
17    *a_tail = new_node;                                // Set the tail to new_node.
18 }
```



Stack					
addr	type*	name*	value	part	fn
200	int	argc	1		main(...)
204	char**	argv	→ {"./foo"}	args	
212	void*			ret addr	
220	struct Node*	head	NULL	locals	
228	struct Node*	tail	NULL		
236	struct Node**	a_head	220	args	append(...)
244	struct Node**	a_tail	228		
252	int	value	3		
256	void*			ret addr	
264	struct Node*	new_node	trash bin	locals	

Heap		
addr	value	lock
400		lock icon
.....		
.....		
.....		
.....		
.....		
.....		
.....		
.....		
.....		

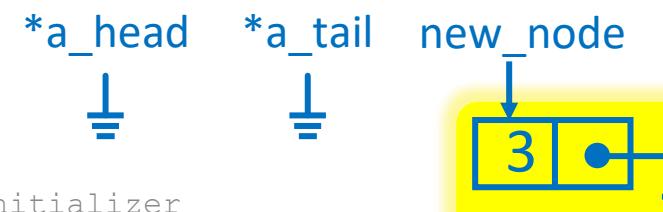
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`.

Appending to a linear linked list with the append (...) function.

Step 2. The first node in the list is allocated (line 6) and initialized (line 9).

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10    ▶AS OF HERE◀
11    if(*a_tail == NULL) {
12        *a_head = new_node;
13    }
14    else {
15        (*a_tail) -> next = new_node;
16    }
17    *a_tail = new_node;
18 }
```



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	locals	
228	struct Node*	tail	NULL		
236	struct Node**	a_head	220	args	append(...)
244	struct Node**	a_tail	228		
252	int	value	3		
256	void*			ret addr	
264	struct Node*	new_node	400	locals	

Heap		
addr	value	fn
400	.value = 3	lock
	.next = NULL	unlock
412		
...		
...		
...		
...		
...		

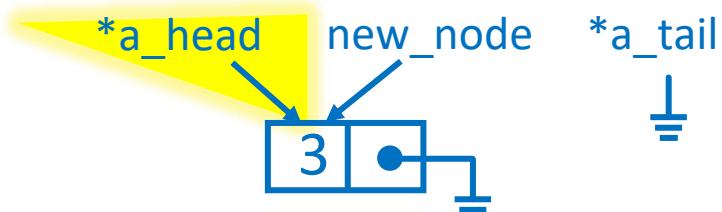
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume sizeof(int)==4, sizeof(char)==1, sizeof(void*)==8.

Appending to a linear linked list with the `append(...)` function.

Step 3. Since the list is currently empty (`*a_tail == NULL`), the head (in `main(...)`'s stack frame) is initialized by way of `a_head`.

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node; >AS OF HERE<           //      Set head to new_node
13    }
14    else {                                              // If list is not empty...
15        (*a_tail) -> next = new_node;                //      Connect tail to new_node
16    }
17    *a_tail = new_node;                                // Set the tail to new_node.
18 }
```



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		
236	struct Node**	a_head	220	args	append(...)
244	struct Node**	a_tail	228		
252	int	value	3		
256	void*			ret addr	
264	struct Node*	new_node	400	locals	

Heap		
addr	value	fn
400	.value = 3	lock
	.next = NULL	unlock
412		
...		
...		
...		
...		
...		
...		

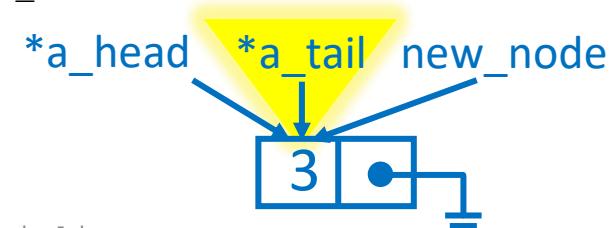
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`.

Appending to a linear linked list with the `append(...)` function.

Step 4. The tail (in `main(...)`'s stack frame) is initialized by way of `a_tail`. The list now has size 1.

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node;                            //      Set head to new_node
13    }
14    else {                                              // If list is not empty...
15        (*a_tail) -> next = new_node;                //      Connect tail to new_node
16    }
17    *a_tail = new_node;    ►AS OF HERE◀           // Set the tail to new_node.
18 }
```



Stack					
addr	type*	name*	value	part	fn
200	int	argc	1		
204	char**	argv	→ {"./foo"}	args	
212	void*			ret addr	main(...)
220	struct Node*	head	NULL 400	locals	
228	struct Node*	tail	NULL 400		
236	struct Node**	a_head	220	args	
244	struct Node**	a_tail	228		
252	int	value	3		
256	void*			ret addr	append(...)
264	struct Node*	new_node	400	locals	

Heap		
addr	value	
400	.value = 3	locked
	.next = NULL	locked
412		
...		
...		
...		
...		
...		
...		
...		

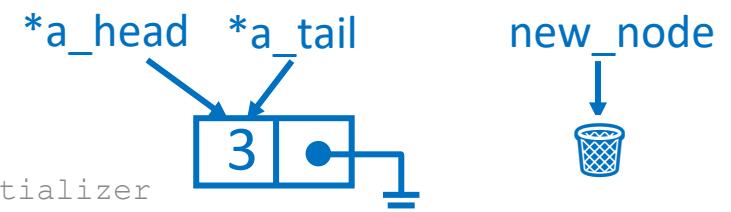
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`.

Appending to a linear linked list with the append (...) function.

Step 5. main(...) calls append(4, &head, &tail). The variables head and tail (in main(...)) both refer to the one and only node in the list.

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4     ▶AS OF HERE◀
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node;                            //      Set head to new_node
13    }
14    else {                                            // If list is not empty...
15        (*a_tail) -> next = new_node; //      Connect tail to new_node
16    }
17    *a_tail = new_node;                                // Set the tail to new_node.
18 }
```



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	locals	
228	struct Node*	tail	NULL		
236	struct Node**	a_head	220	args	append(...)
244	struct Node**	a_tail	228		
252	int	value	4		
256	void*			ret addr	
264	struct Node*	new_node	▀	locals	

Heap		
addr	value	fn
400	.value = 3	▀
	.next = NULL	▀
412		

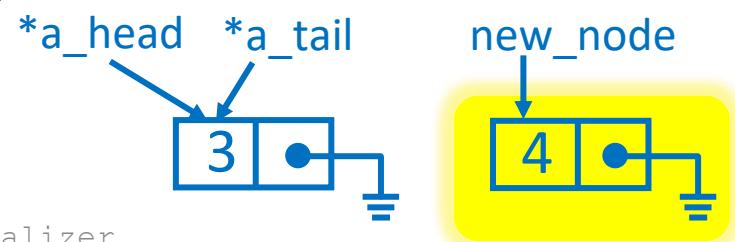
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume sizeof(int)==4, sizeof(char)==1, sizeof(void*)==8.

Appending to a linear linked list with the append (...) function.

Step 6. The second node in the list is allocated (line 6) and initialized (line 9).

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10    ▶AS OF HERE◀
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node;                            //      Set head to new_node
13    }
14    else {                                               // If list is not empty...
15        (*a_tail) -> next = new_node;                //      Connect tail to new_node
16    }
17    *a_tail = new_node;                                // Set the tail to new_node.
18 }
```



Stack						
addr	type*	name*	value	part	fn	
200	int	argc	1	args	main(...)	
204	char**	argv	→ {"./foo"}			
212	void*			locals		
220	struct Node*	head	400			
228	struct Node*	tail	400	args		
236	struct Node**	a_head	220			
244	struct Node**	a_tail	228			
252	int	value	4			
256	void*			locals		
264	struct Node*	new_node	412			

Heap		
addr	value	lock
400	.value = 3	locked
	.next = NULL	locked
412	.value = 4	locked
	.next = NULL	locked
424		

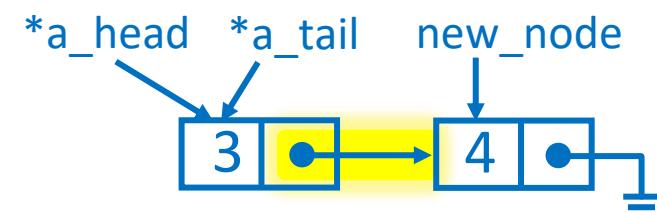
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`.

Appending to a linear linked list with the `append(...)` function.

Step 7. Since the list is *not empty* (`*a_tail != NULL`), the `tail` (in `main(...)`'s stack frame) is connected to the `new_node` by way of `a_tail`.

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node;                            //      Set head to new_node
13    }
14    else {                                              // If list is not empty...
15        (*a_tail) -> next = new_node; ►AS OF HERE◄ // Connect tail to new_node
16    }
17    *a_tail = new_node;                                // Set the tail to new_node.
18 }
```



Stack					
addr	type*	name*	value	part	f
200	int	argc	1		
204	char**	argv	→ {"./foo"}	args	main(...)
212	void*			ret addr	
220	struct Node*	head	400	locals	
228	struct Node*	tail	400		
236	struct Node**	a_head	220	args	
244	struct Node**	a_tail	228		
252	int	value	4		
256	void*			ret addr	append(...)
264	struct Node*	new_node	412	locals	

Heap		
addr	value	lock
400	.value = 3	l
	.next = NULL 412	l
412	.value = 4	l
	.next = NULL	l
424		

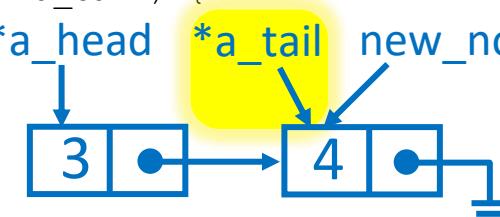
Types and names are not stored in memory or executable. Addresses shown are fictional. Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`.

Appending to a linear linked list with the `append(...)` function.

Step 8. The tail (in `main(...)`'s stack frame) is set to the new tail (`new_node`) by way of `a_tail`. The list now has size 2.

```

1 void append(int value, struct Node** a_head, struct Node** a_tail) {
2     // Assert: If head is NULL then tail must be NULL
3     assert( (*a_tail == NULL) == (*a_head == NULL) );
4
5     // Allocate memory on heap for new_node.
6     struct Node* new_node = malloc(sizeof(*new_node));
7
8     // Initialize all fields of new_node using compound initializer
9     *new_node = (struct Node) { .value=value, .next=NULL };
10
11    if(*a_tail == NULL) {                                // If list is empty...
12        *a_head = new_node;                            //      Set head to new_node
13    }
14    else {                                              // If list is not empty...
15        (*a_tail) -> next = new_node;                //      Connect tail to new_node
16    }
17    *a_tail = new_node;    ►AS OF HERE◀          // Set the tail to new_node.
18 }
```



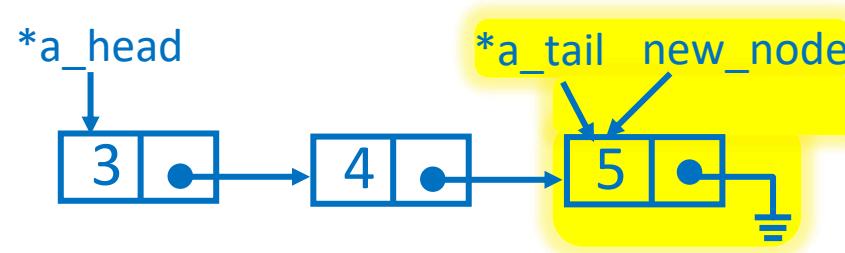
Stack					
addr	type*	name*	value	part	fn
200	int	argc	1	args main(...)	
204	char**	argv	→ {"./foo"}		
212	void*				
220	struct Node*	head	400		
228	struct Node*	tail	412		
236	struct Node**	a_head	220	args	
244	struct Node**	a_tail	228		
252	int	value	4		
256	void*				
264	struct Node*	new_node	412	locals	

Heap		
addr	value	fn
400	.value = 3	lock
	.next = NULL 412	
412	.value = 4	lock
	.next = NULL	
424		

Types and names are not stored in memory or executable. Addresses shown are fictional. Assume `sizeof(int)==4`, `sizeof(char)==1`, `sizeof(void*)==8`.

Appending to a linear linked list with the append (...) function.

Step 9. main(...) calls append(5, &head, &tail) to add a third node. This is as of line 17 (just before returning from append(...)).



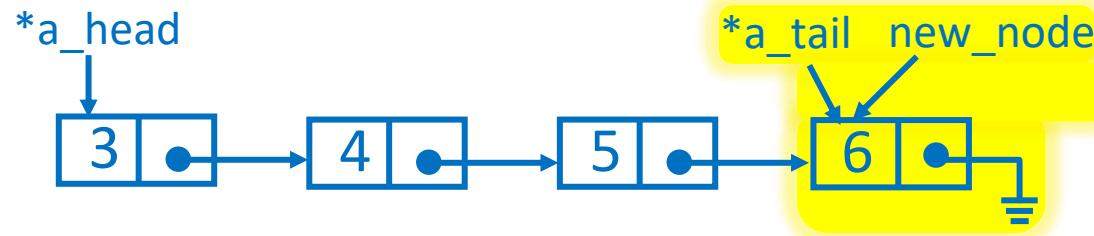
Stack						Heap	
addr	type*	name*	value	part	f	addr	value
200	int	argc	1		main(...)	400	.value = 3
204	char**	argv	→ {"./foo"}	args		412	.next = 412
212	void*			ret addr	424	.value = 4	
220	struct Node*	head	400	locals		.next = NULL 424	
228	struct Node*	tail	424		436	.value = 5	
236	struct Node**	a_head	220	args		.next = NULL	
244	struct Node**	a_tail	228				
252	int	value	5				
256	void*			ret addr			
264	struct Node*	new_node	424	locals			

Types and names are not stored in memory or executable. Addresses shown are fictional. Assume sizeof(int)==4, sizeof(char)==1, sizeof(void*)==8.

Appending to a linear linked list with the append (...) function.

10

Step 10. main(...) calls append(6, &head, &tail) to add a fourth node. This is as of line 17 (just before returning from append(...)).



Stack					
addr	type*	name*	value	part	fn
200	int	argc	1	args main(...)	main(...)
204	char**	argv	→ {"./foo"}		
212	void*				
220	struct Node*	head	400		
228	struct Node*	tail	436		
236	struct Node**	a_head	220		
244	struct Node**	a_tail	228		
252	int	value	6		
256	void*				
264	struct Node*	new_node	436	locals	append(...)

Heap		
addr	value	lock
400	.value = 3	locked
	.next = 412	locked
412	.value = 4	locked
	.next = 424	locked
424	.value = 5	locked
	.next = NULL 436	locked
436	.value = 6	locked
	.next = NULL	locked
448		

Types and names are not stored in memory or executable. Addresses shown are fictional. Assume sizeof(int)==4, sizeof(char)==1, sizeof(void*)==8.