CS 240: Programming in C

Lecture 13: malloc() and free(), Linked Lists

Announcements

- Homework 6 extended to Friday
- Homework 7 released next week

Pointer example from last time

```
int main() {
  int ctr = 0;
  int *ptr = 0;
  int int4 = 18;
  int int3 = 11;
  int int2 = 10;
  int int1 = 7;

  ptr = &int1;

  for (ctr = 0; ctr < 7; ctr++) {
    printf("Value at address %p: 0x%x (%d)\n",
           ptr, *ptr, *ptr);
    ptr++;
  }

  return 0;
}
```

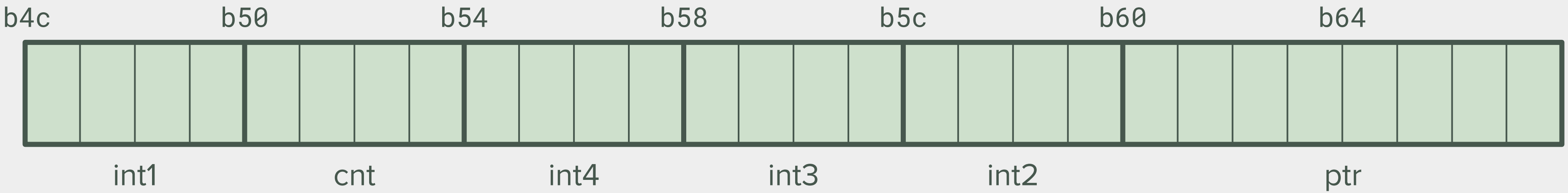
Pointer example from last time

```
Value at address 0x7ffe41adeb4c: 0x7 (7)
Value at address 0x7ffe41adeb50: 0x1 (1)
Value at address 0x7ffe41adeb54: 0x12 (18)
Value at address 0x7ffe41adeb58: 0xb (11)
Value at address 0x7ffe41adeb5c: 0xa (10)
Value at address 0x7ffe41adeb60: 0x41adeb60 (1101917024)
Value at address 0x7ffe41adeb64: 0x7ffe (32766)
```

- What is happening here?

Pointer example from last time

```
Value at address 0x7ffe41adeb4c: 0x7 (7)
Value at address 0x7ffe41adeb50: 0x1 (1)
Value at address 0x7ffe41adeb54: 0x12 (18)
Value at address 0x7ffe41adeb58: 0xb (11)
Value at address 0x7ffe41adeb5c: 0xa (10)
Value at address 0x7ffe41adeb60: 0x41adeb60 (1101917024)
Value at address 0x7ffe41adeb64: 0x7ffe (32766)
```



Pointer example from last time

Iteration 0:

b4c	b50	b54	b58	b5c	b60	b64
07 00 00 00	00 00 00 00	12 00 00 00	0b 00 00 00	0a 00 00 00	4c eb ad 41	fe 7f 00 00
int1	cnt	int4	int3	int2	ptr	

```
Value at address 0x7ffe41adeb4c: 0x7 (7)
```

Iteration 1:

b4c	b50	b54	b58	b5c	b60	b64
07 00 00 00	01 00 00 00	12 00 00 00	0b 00 00 00	0a 00 00 00	50 eb ad 41	fe 7f 00 00
int1	cnt	int4	int3	int2	ptr	

```
Value at address 0x7ffe41adeb50: 0x1 (1)
```

Pointer example from last time

Iteration 2:

b4c				b50				b54				b58				b5c				b60				b64			
07	00	00	00	02	00	00	00	12	00	00	00	0b	00	00	00	0a	00	00	00	54	eb	ad	41	fe	7f	00	00
int1				cnt				int4				int3				int2				ptr							

```
Value at address 0x7ffe41adeb54: 0x12 (18)
```

Iteration 3:

b4c				b50				b54				b58				b5c				b60				b64			
07	00	00	00	03	00	00	00	12	00	00	00	0b	00	00	00	0a	00	00	00	58	eb	ad	41	fe	7f	00	00
int1				cnt				int4				int3				int2				ptr							

```
Value at address 0x7ffe41adeb58: 0xb (11)
```

Pointer example from last time

Iteration 4:

Address	Bytes	Variable
b4c	07 00 00 00	int1
b50	04 00 00 00	cnt
b54	12 00 00 00	int4
b58	0b 00 00 00	int3
b5c	0a 00 00 00	int2
b60	5c eb ad 41	ptr
b64	fe 7f 00 00	ptr

```
Value at address 0x7ffe41adeb5c: 0xa (10)
```

Iteration 5:

Address	Bytes	Variable
b4c	07 00 00 00	int1
b50	05 00 00 00	cnt
b54	12 00 00 00	int4
b58	0b 00 00 00	int3
b5c	0a 00 00 00	int2
b60	60 eb ad 41	ptr
b64	fe 7f 00 00	ptr

```
Value at address 0x7ffe41adeb60: 0x41adeb60 (1101917024)
```

Pointer example from last time

Iteration 6:

b4c				b50				b54				b58				b5c				b60				b64			
07	00	00	00	06	00	00	00	12	00	00	00	0b	00	00	00	0a	00	00	00	64	eb	ad	41	fe	7f	00	00
int1				cnt				int4				int3				int2				ptr							

```
Value at address 0x7ffe41adeb64: 0x7ffe (32766)
```

Pointer example from last time

Iteration 6:

b4c	b50	b54	b58	b5c	b60	b64
07 00 00 00	06 00 00 00	12 00 00 00	0b 00 00 00	0a 00 00 00	64 eb ad 41	fe 7f 00 00
int1	cnt	int4	int3	int2	ptr	

```
Value at address 0x7ffe41adeb64: 0x7ffe (32766)
```

- Remember: ptr is an `int *`
- Whatever is stored at that address is interpreted as a 4-byte integer
- Incrementing an `int *` adds 4 bytes
- If ptr was a `char *` instead, this would look very different

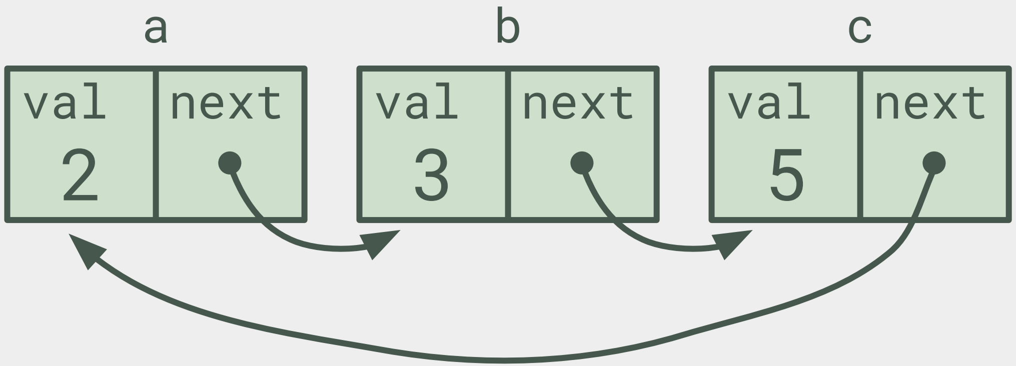
PURDUE UNIVERSITY

Nodes in a ring

```
struct node a;
struct node b;
struct node c;

void setup() {
  a.val = 2;
  b.val = 3;
  c.val = 5;

  a.next = &b;
  b.next = &c;
  c.next = &a;
}
```

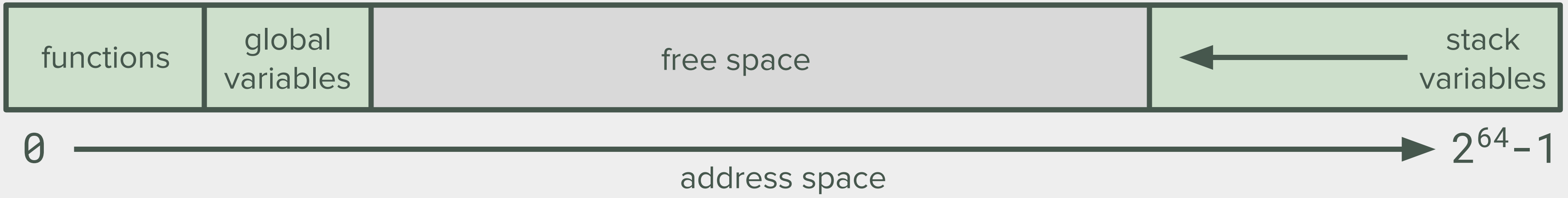


What’s the point?

- Still not much use for this…
- We still have the same number of node structures
- What if we don’t know the number of nodes we need ahead of time?
- We can create new node structures dynamically

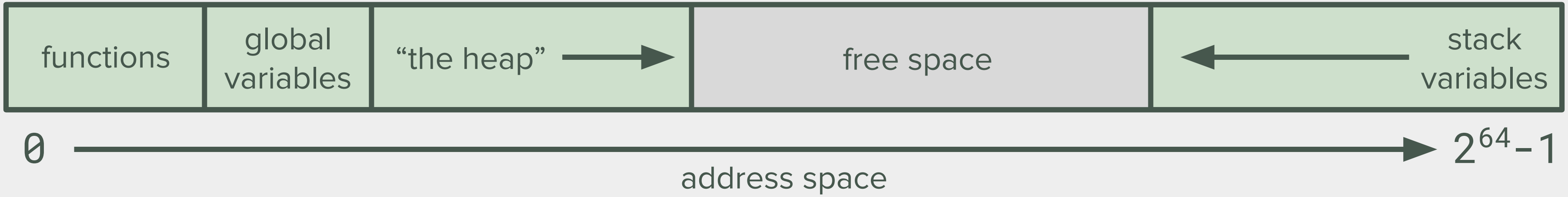
Memory layout revisited

- Here's a macroscopic view of memory for your application.
- Local variables (defined inside functions) appear on the stack
- The stack starts at the highest address and grows downwards
- What about all of that unused memory?



Let’s use that memory

- We can allocate memory in what we call “the heap”
  - Not related to the binary heap data structure
  - A more apt name would be “the pool”
- Unlike the stack, the heap grows upwards
- Use the standard library to manage allocation for us



Stack vs. Heap

- Why not just use the stack?
- The stack is used for function calls
- Variables within a function have a specific lifetime
  - They are destroyed when their function returns
  - i.e., the “stack frame” is “popped” from the stack
- Sometimes we want a variable to live longer than that
- The heap is much more flexible

malloc() and free()

- The malloc() function is used to allocate a chunk of the heap

```
address malloc(int size);
```

- The free() function tells the system that we’re done with that chunk.

```
void free(address);
```

Example of malloc()

```
#include <stdio.h>
#include <malloc.h>

void get_some_memory() {
  int *int_arr = NULL;

  int_arr = malloc(40 * sizeof(int));
  for (int i = 0; i < 40; i++)
    int_arr[i] = 15;
  free(int_arr);
  int_arr = NULL;
}
```

Allocating a struct

```
#include <stdio.h>
#include <malloc.h>

struct node { int val;
              struct node *next; };
void alloc_a_struct() {
  struct node *node_ptr = NULL;

  node_ptr = malloc(sizeof(struct node));
  node_ptr->val = 42;
  node_ptr->next = NULL;
  free(node_ptr);
  node_ptr = NULL;
}
```

Things to remember

- When using malloc(), always double check that you specify the proper size.
  - Otherwise, chaos will ensue
- Always check the return value from malloc()
- After free(ptr); ptr still points to the same chunk of memory
  - But we no longer have it reserved
  - A subsequent malloc() may reuse it!
- Always say ptr = NULL; after a free(ptr); call
  - That way we do not try to use that memory again

malloc(), calloc()

- malloc(int size) reserves a chunk of memory
  - What does that chunk contain?
- calloc(int n, int s) reserves n chunks of memory of size s
  - and sets all of the bytes to zero
- free(void *ptr) will cancel the reservation for memory from either source
  - What happens to the contents of that memory?

What’s wrong with this?

```
#include <stdio.h>

typedef struct node {
  int val;
  struct node *next;
} node_t;

node_t *alloc_a_struct() {
  node_t my_node;

  my_node.val = 42;
  my_node.next = NULL;
  return &my_node;
}
```

What's wrong with this?

```
#include <stdio.h>

typedef struct node {
  int val;
  struct node *next;
} node_t;

node_t *alloc_a_struct() {
  node_t my_node;

  my_node.val = 42;
  my_node.next = NULL;
  return &my_node;
}
```

<u>Never</u> return a pointer to something that is stack-allocated

Let’s fix it

```
#include <stdio.h>

typedef struct node {
  int val;
  struct node *next;
} node_t;

node_t *alloc_a_struct() {
  node_t *my_node = malloc(sizeof(node_t));

  my_node->val = 42;
  my_node->next = NULL;
  return my_node;
}
```

Linked lists

- Consider this structure:

```
struct node {
  int val;
  struct node *next;
};
```

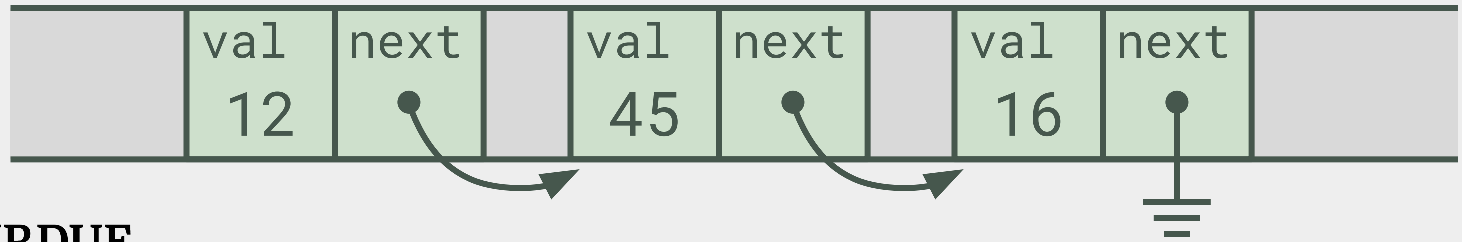
PURDUE UNIVERSITY

Linked lists

- Consider this structure:

```
struct node {
  int val;
  struct node *next;
};
```

- Create three of them somewhere in memory
- Let each one point to the next, and the last have a NULL pointer



Code for the previous example

```
struct node *one_ptr = NULL;
struct node *two_ptr = NULL;
struct node *three_ptr = NULL;

one_ptr = malloc(sizeof(struct node));
one_ptr->val = 12;
two_ptr = malloc(sizeof(struct node));
two_ptr->val = 45;
three_ptr = malloc(sizeof(struct node));
three_ptr->val = 16;

one_ptr->next = two_ptr;
two_ptr->next = three_ptr;
three_ptr->next = NULL;
```

Too many pointers!

- In practice, we could do the previous example without using so many pointers
- For instance, we can refer to any element within any of the structures via the first structure.
  - E.g., `one_ptr->next->val`

Forming a linked list

- Growing “forward” (adding to the end):
  - Use one pointer to refer to the “head” of the list
  - Use a second pointer to refer to the “tail” of the list
  - Add the new structure to `tail->next`
  - Set `tail = tail->next`
- Growing “backward” (adding to the beginning):
  - Use one pointer to refer to the “head” of the list
  - Use a temporary pointer to refer to a new structure
  - Set `temp->next_ptr = head`
  - Set `head = temp`

PURDUE UNIVERSITY

Special case for first node

- We start with a head pointer and a tail pointer

```
struct node *head = NULL;
struct node *tail = NULL;
```

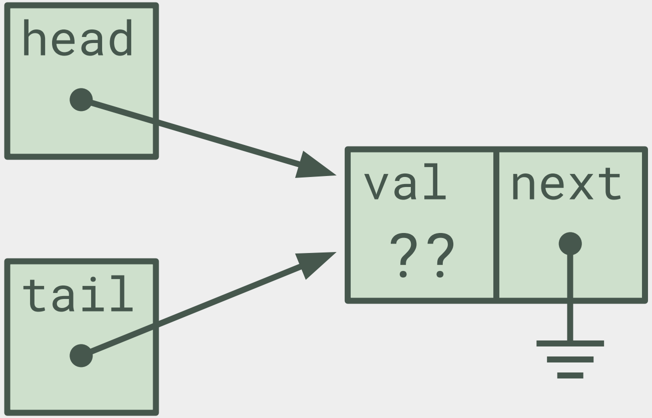
- Then we allocate the head:

```
if (head == NULL) {
  head = malloc(sizeof(struct node));
  assert(head != NULL);
  head->next = NULL;
}
```

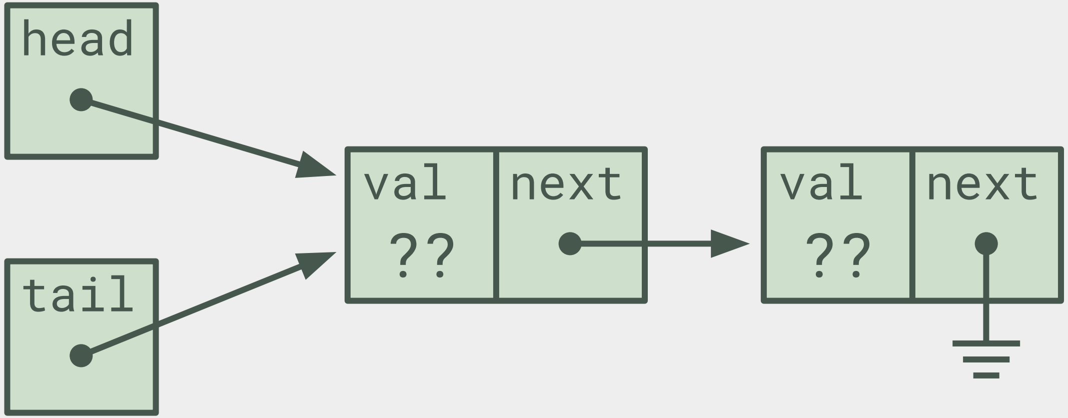
- Then we set the tail to the head

```
tail = head;
```

Initial setup

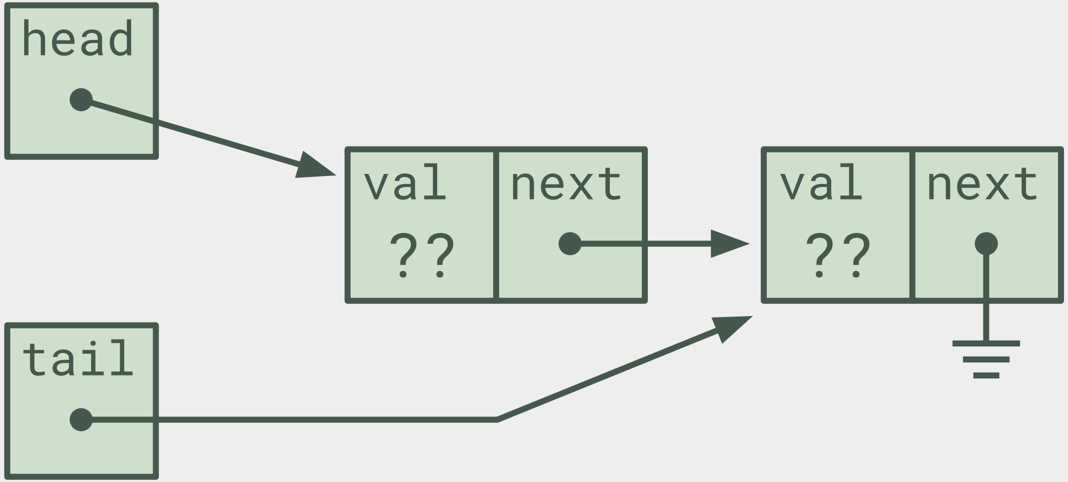


Forward growing (step one)



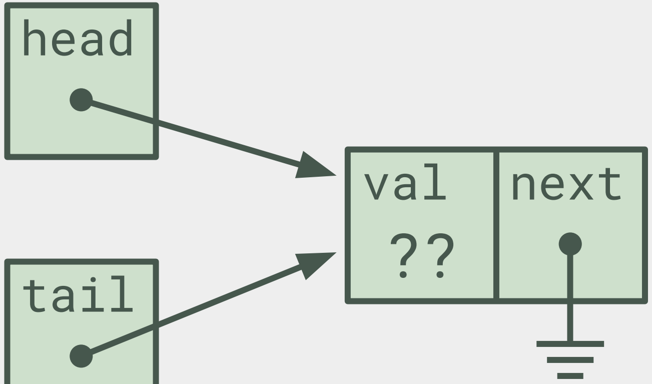

```
tail->next = malloc(sizeof(struct node));
assert(tail->next != NULL);
tail->next->next = NULL;
```

Forward growing (step two)

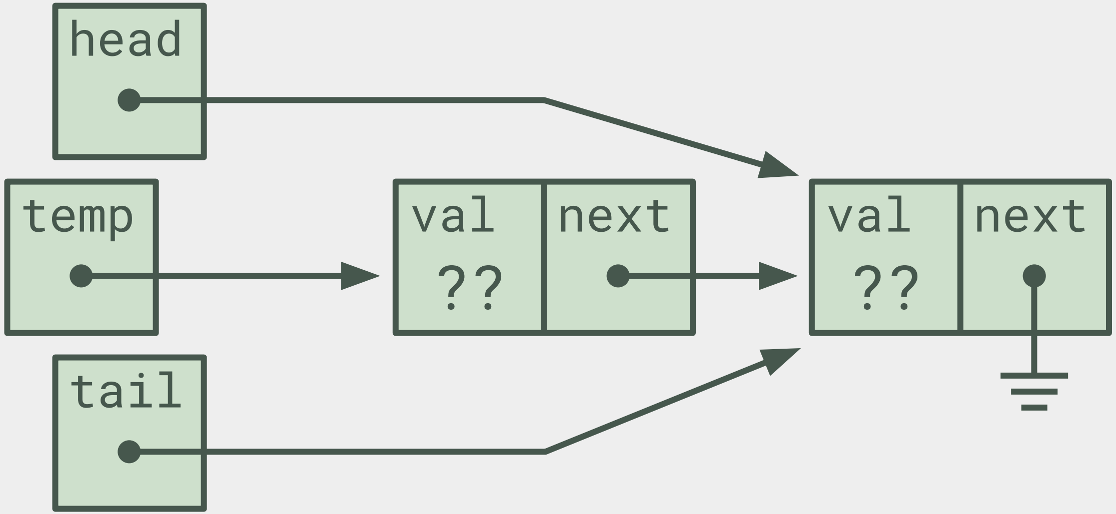



```
tail = tail->next;
```

Initial setup

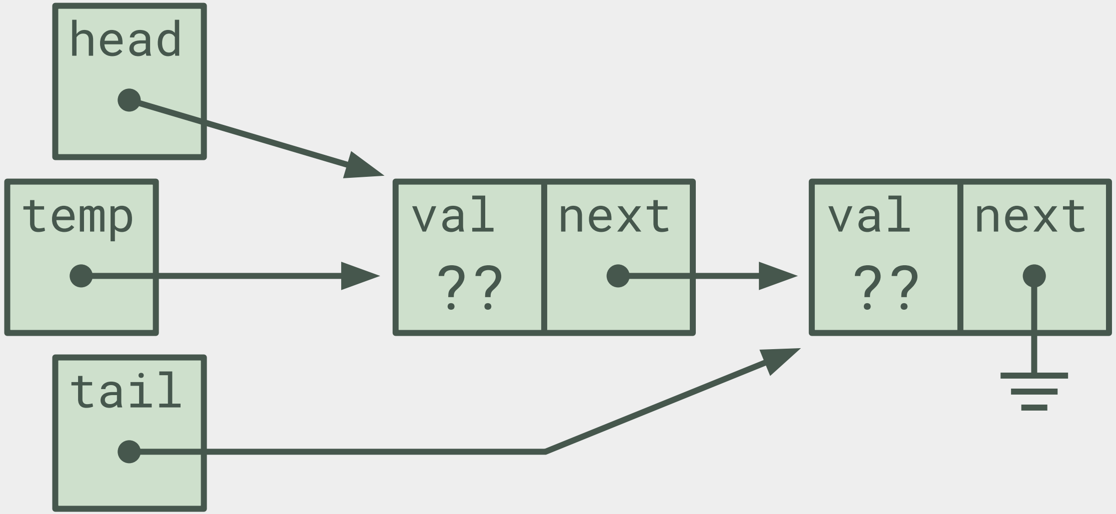


Reverse growing (step one)




```
struct node *temp = malloc(sizeof(struct node));
assert(temp != NULL);
temp->next = head;
```

Reverse growing (step two)




```
head = temp;
```

Traversing a linked list

- Usually, you do not know how many nodes are in a linked list
  - Have to “traverse” it to find an item or do work on the structures
- You can traverse a linked list with one extra pointer

```
struct node *p = head;
while (p != NULL) {
  p->val++;
  p = p->next;
}
```

Deleting a linked list

- Deletion of a linked list is a special case of the traversal process
- What’s wrong with this?

```
p = head;
while (p != NULL) {
  free(p);
  p = p->next;
}
```

PURDUE UNIVERSITY

Deleting a linked list

- Deletion of a linked list is a special case of the traversal process
- What's wrong with this?

```
p = head;
while (p != NULL) {
  free(p);
  p = p->next;
}
```

p has already been freed!

PURDUE UNIVERSITY

Deleting a linked list

- Deletion of a linked list is a special case of the traversal process
- Let's fix it:

```
p = head;
while (p != NULL) {
  struct node *next = p->next;
  free(p);
  p = next;
}
```

Functions to simplify list mgmt

- Writing code to do operations on lists is
  - Repetitive
  - Tedious
  - Error prone
- It is usually a good idea to encapsulate the functionality into functions to create, delete, insert, and append new structures

Example: create_node()

- Allocate a new node, check the malloc() return value, and set the fields:

```
struct node *create_node(int new_value) {
  struct node *temp = NULL;

  temp = malloc(sizeof(struct node));
  assert(temp != NULL);

  temp->val = new_value;
  temp->next = NULL;

  return temp;
}
```

Bigger "payload"

- Normally a structure in a linked list contains many more elements than just a single value and a list pointer:

```
struct big_node {
  struct big_node *next;
  float height;
  float width;
  float weight;
  int angle;
  float age;
};
```

PURDUE UNIVERSITY

For next lecture

- Read K&R Chapter 8.7, Beej Chapter 11
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.