PURDUE UNIVERSITY®

CS 240: Programming in C

Lecture 13: Linked Lists

Prof. Jeff Turkstra

Midterm Exam

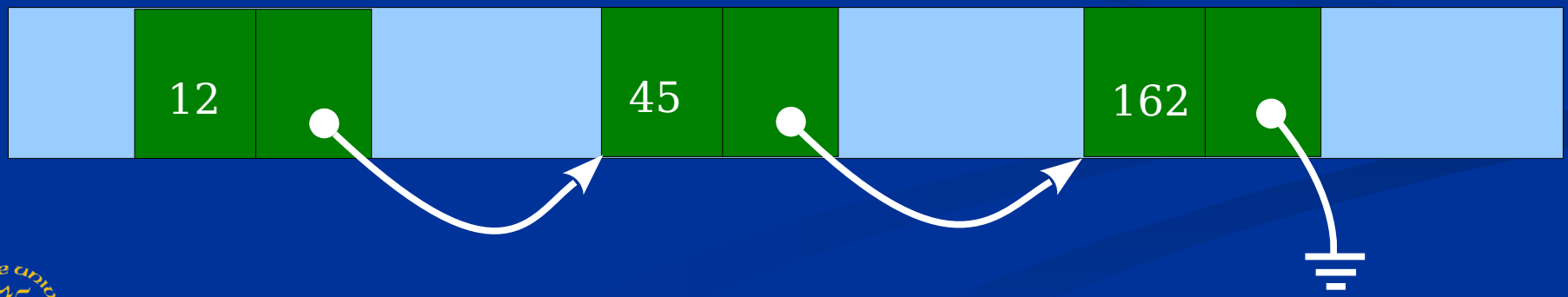
- Tonight!
  - Seating charts on course website
  - 8pm - 10pm
  - Bring a pen/pencil
    - Probably a pencil – you'll be writing a fair amount of code
  - Anything else goes up front
  - Sample questions and exam available on website
  - Note on Ed Discussion regarding question hints

Linked lists

- Consider this structure:

```
struct node {
  int value;
  struct node *next_ptr;
};
```

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



Code for previous linked list

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

one_ptr = malloc(sizeof(struct node));
one_ptr->value = 12;
two_ptr = malloc(sizeof(struct node));
two_ptr->value = 45;
three_ptr = malloc(sizeof(struct node));
three_ptr->value = 162;

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

Too many pointers!

- In practice, we would 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.,
  - What is `one_ptr->next_ptr->value`?

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 every new structure to tail->next_ptr
- 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

Special case for first node

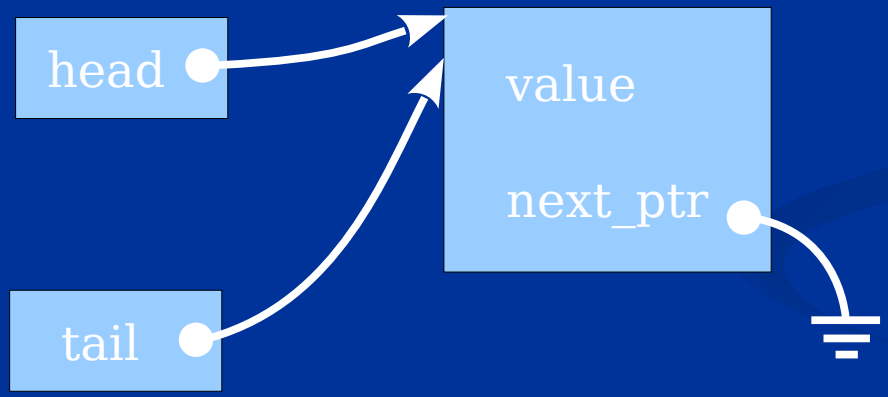
- The first thing we must do in either case is allocate the head:

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

- In the forward growing case, we then set the tail to the head

```
tail = head;
```

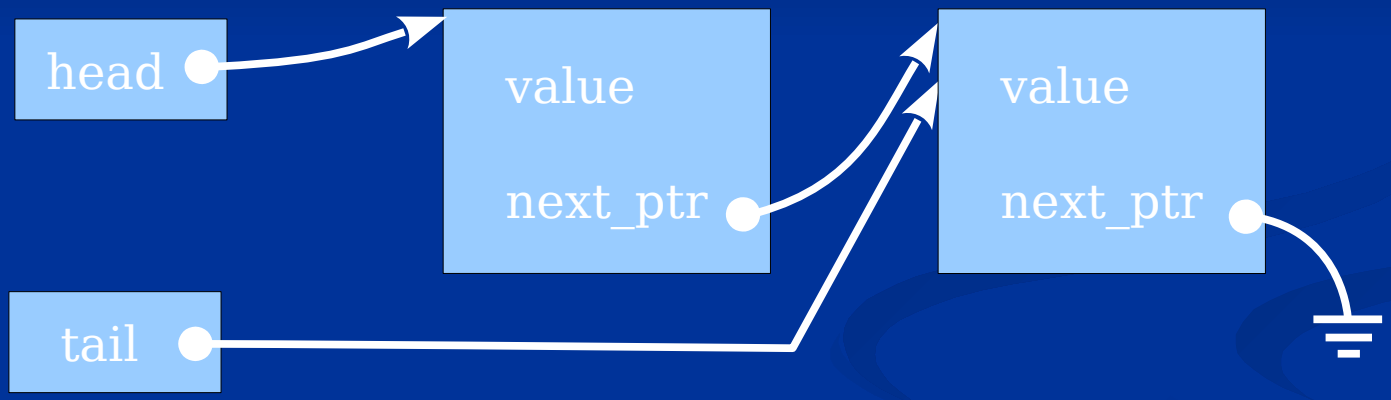
Forward growing (initial setup)



Forward growing (step two)


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

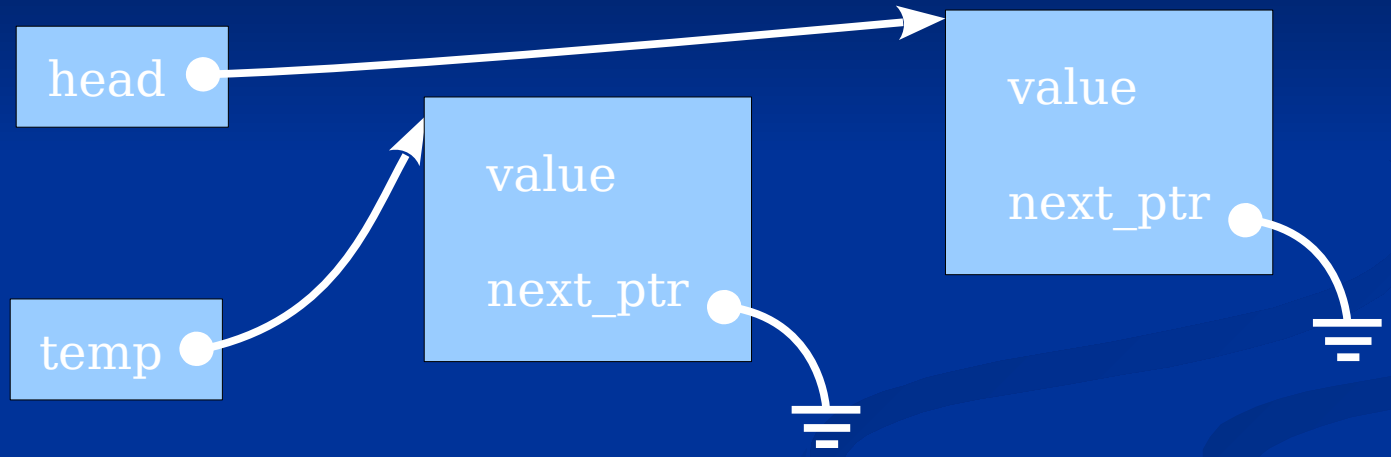
Forward growing (step three)




```
tail = tail->next_ptr;
```

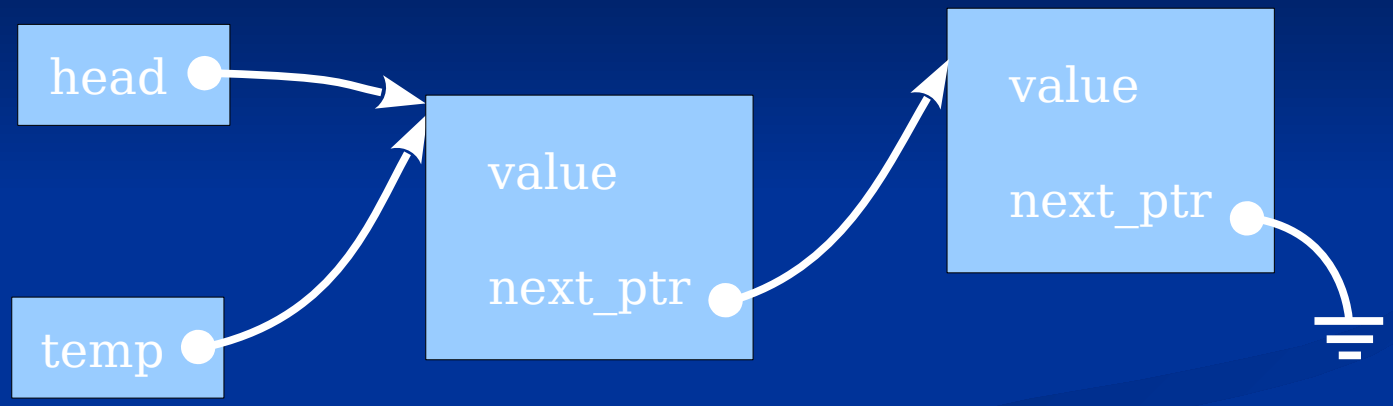
Reverse growing (initial setup)

Reverse growing (step two)




```
temp = malloc(sizeof(struct node));
assert (temp != NULL);
temp->next_ptr = NULL;
```

Step three




```
temp->next_ptr = head;
head = temp;
```

Traversing a linked list

- Usually, you do not know how many structures 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. E.g.:

```
p = head;

while (p != NULL) {
  p->value++;
  p = p->next_ptr;
}
```

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_ptr;
}
```

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) {
   struct node *tmp = p->next;
   free(p);
   p = tmp;
}
```

Functions to simplify list management

- 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->value = new_value;
  temp->next_ptr = 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. E.g.:

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

¿Usted tiene alguna pregunta?

- Sometimes linked lists make C look like a completely different language

Purdue Trivia

- Dr. David N. Crosthwait Jr. was an African-American mechanical and electrical engineer, inventor, and writer
  - BS 1913, MS 1920
  - Honorary Doctorate 1975
- Expertise was on air ventilation, central air conditioning, and heat transfer systems
  - Radio City Music Hall
- 39 US patents, 80 international patents

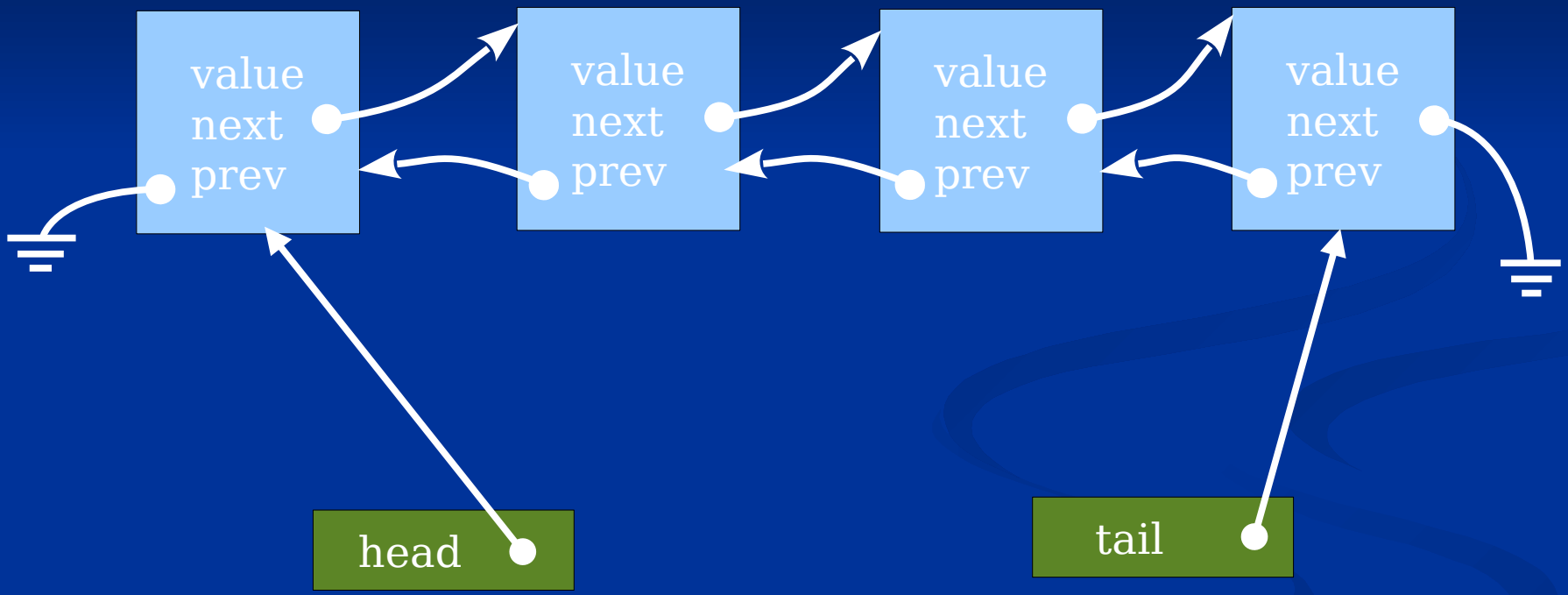
Tough questions

- It's easy to traverse a list from head to tail
  - How about tail to head?
- Can you write a function that will exchange a specified structure in a linked list with the structure that follows it?
  - Without specifying the head of the list?
- Can you write a function that will prepend a structure before an arbitrary node in the list?
  - Without specifying the head of the list?

Doubly-linked list

- Without the head, the answers to the previous questions are ‘no.’
- The lists we’ve looked at so far are called singly-linked lists
- A doubly-linked list contains two pointers:
  - A “next” pointer
  - A “previous” pointer

Example of a doubly-linked list



Example of declaration

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

struct double_l {
   int value;
   struct double_l *next_ptr;
   struct double_l *prev_ptr;
};
```

Example of use

```
int main() {
   struct double_l *ptr = NULL;

   ptr = malloc(sizeof(struct double_l));
   assert(ptr != NULL);

   ptr->next_ptr = NULL;
   ptr->prev_ptr = NULL;

   ptr->value = 15;

   return ptr->value;
}
```

Practice

- Try creating similar functions to above for a doubly-linked list
  - We'll look at some of them next time
- Draw the diagrams first
- Practice on paper
- Then write the code

Important points

- There are four steps.
- When you implement insert, prepend, append, etc you should always have four steps
- It is imperative to put those steps in the right order
  - Some steps are interchangeable; some are not!
- You should practice this on paper

For next lecture

- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Boiler Up!